Hello World pod lupą

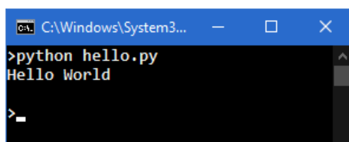
Pierwszym krokiem w klasycznej ścieżce edukacji przyszłych programistów jest stworzenie programu wypisującego – najczęściej w konsoli – tekst „Hello, World!”. Sam program jest z definicji banalny, ale to, co dzieje po jego uruchomieniu – już nie do końca. W tym artykule prześledzimy ścieżkę wykonania mini-programu „Hello World” napisanego w Pythonie, zaczynając od pojedynczego wywołania wysokopoziomowej funkcji print, poprzez kolejne poziomy abstrakcji interpretera, systemu operacyjnego i sterowników graficznych, a kończąc na wyświetleniu odpowiednich pikseli na ekranie. Skupimy się przy tym na systemie Windows. Jak się okaże, ścieżka ta sama w sobie nie jest ani prosta, ani krótka, ale zdecydowanie bardzo ciekawa.

KOD W PYTHONIE

Kod, od którego zaczniemy, jest banalny:

```
print("Hello World")
```

Efekt jego działania jest zarówno przewidywalny, jak i oczywisty:



Co jednak sprawia, że nasz komputer w efekcie wykonania powyższego programu uznaje za stosowne zmienić kolor kilkuset wybranych pikseli na ekranie?

Pierwszym krokiem okazuje się być kompilacja wskazanego pliku zawierającego nasz kod źródłowy (*hello.py*). Niektórzy czytelnicy mogą czuć się zaskoczeni już w tym momencie: „Ale chwila, czy Python – w przeciwieństwie do C czy C++ – nie jest czasem językiem interpretowanym?”. I faktycznie, Python często jest nazywany językiem skryptowym, a te, z definicji, nie powinny być kompilowane, czyż nie?

W praktyce wiele popularnych języków skryptowych, jak na przykład PHP, Ruby, Lua, JavaScript, Perl czy właśnie Python, są kompilowane do swoich własnych wariantów kodu bajtowego (ang. *bytecode*), czyli formy binarnej, która – mimo iż jest niekompatybilna z językiem maszynowym prawdziwych procesorów[1] – jest dużo łatwiejsza do szybkiej interpretacji i wykonania niż czysty kod źródłowy. Python w tym przypadku jest językiem o tyle wdzięcznym, że udostępnia moduły pozwalające na wgląd w poszczególne części tego procesu z poziomu samego języka.

Dla przypomnienia, proces kompilacji – w sporym uproszczeniu – można sprowadzić do trzech kroków:

» **analizy leksykalnej** (wykonywanej przez *lexer*), której wynikiem jest lista tokenów,
» **analizy składniowej** (wykonywanej przez *parser*), której wynikiem jest drzewo wyrażeń (AST, *Abstract Syntax Tree*),
» oraz **generowania kodu** – w naszym przypadku bajtowego.

1. Z drobnym wyjątkiem w postaci Jazelle DBX, czyli dodatkowego trybu w niektórych starszych procesorach z rodziny ARM, który umożliwiał wykonanie kodu bajtowego Java.

Efekt analizy leksykalnej możemy obejrzeć, korzystając z uruchomionego z linii poleceń modułu `tokenize`, czego wynikiem będzie wypisanie listy tokenów, której poszczególne wiersze zawierają pozycję danego tokena w pliku, jego typ oraz ewentualną zawartość tekstową.

```
>python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,5:            NAME           'print'
1,5-1,6:            OP             '('
1,6-1,19:           STRING         '"Hello World"'
1,19-1,20:          OP             ')'
1,20-1,21:          NEWLINE        '\n'
2,0-2,0:            ENDMARKER      ''
```

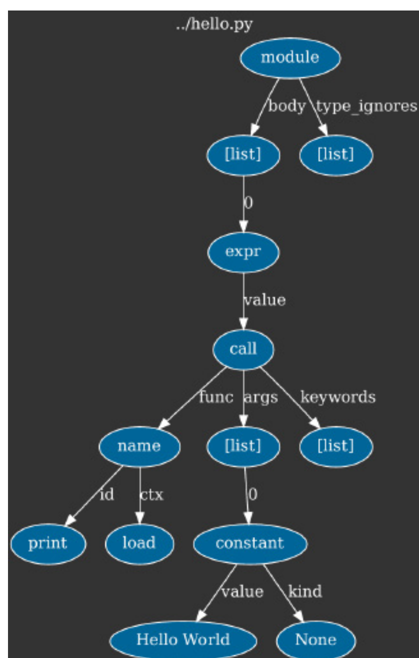
Nasz prosty Hello World składa się z niewielu tokenów: nazwy `print`, operatorów `(` i `)` oraz literału tekstowego `"Hello World"`. Oprócz nich jest jeszcze nieistotny w tym przypadku znak nowej linii, a także tokeny zawierające metadane, takie jak użyte w pliku źródłowym kodowanie czy znacznik końca danych.

Tokeny są następnie przekazywane do parsera, który, korzystając z zasad gramatycznych, generuje drzewo AST. Wynik działania parsera możemy obejrzeć, korzystając z modułu `ast`, który, podobnie jak wcześniej `tokenize`, działa również bezpośrednio z linii poleceń.

```
>python -m ast hello.py
Module(
  body=[
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Constant(value='Hello World')],
        keywords=[]))],
  type_ignores=[])
```

Podobnie jak w przypadku listy tokenów, samo drzewo AST ogranicza się jedynie do kilku węzłów: do korzenia `Module` podczepione jest tylko jedno wyrażenie – typu `Call` (wywołanie funkcji), które z kolei połączone jest jedynie z węzłem zawierającym nazwę (`Name`) funkcji oraz jednym argumentem będącym stałą (`Constant`) o wartości `'Hello World'`.

Warto dodać, że powstało kilka prostych narzędzi umożliwiających wyświetlenie drzewa AST programów napisanych w Pythonie w postaci faktycznego grafu. Efekt działania jednego z nich – AST visualizer autorstwa *quantifiedcode* [1] – znajduje się na Rysunku 1.



Rysunek 1. Drzewo AST programu „Hello World"

W kolejnym kroku na podstawie drzewa AST generowany jest kod bajtowy, który również możemy podejrzeć, tym razem korzystając z modułu `dis`, który wyświetli wszystkie instrukcje w formie tekstowej.

```
>python -m dis hello.py
1       0 LOAD_NAME       0 (print)
        2 LOAD_CONST      0 ('Hello World')
        4 CALL_FUNCTION   1
        6 POP_TOP
        8 LOAD_CONST      1 (None)
       10 RETURN_VALUE
```

Skrótowy opis wszystkich instrukcji można znaleźć w dokumentacji samego modułu `dis` [2], więc na potrzeby tego artykułu ograniczymy się do omówienia jedynie operacji użytych w naszym Hello World:

» **`LOAD_NAME`** ***nazwa*** – umieszcza na stosie wartość zmiennej globalnej o podanej nazwie[2],
» **`LOAD_CONST`** ***stała*** – umieszcza na stosie podaną stałą[3],
» **`CALL_FUNCTION`** ***liczba_parametrów*** – wywołuje zdjętą ze stosu funkcję oraz przekazuje jej podaną liczbę zdjętych ze stosu parametrów,
» **`POP_TOP`** – usuwa jeden element ze szczytu stosu,
» **`RETURN_VALUE`** – wychodzi z funkcji, zwracając element z wierzchu stosu.

Jak można się domyślić z opisu powyższych instrukcji, wzorcowa implementacja Pythona korzysta z maszyny stosowej i – w przeciwieństwie do np. procesorów ARM czy x86 – nie ma rejestrów, a więc kod jest wykonywany poprzez umieszczanie wartości na stosie, a następnie użycie instrukcji, które zdejmują argumenty ze stosu i umieszczają na nim wynik operacji[4].

2. Posłużyliśmy się tu pewnym uproszczeniem, jako że faktycznie w samym kodzie bajtowym w tym miejscu znajdziemy nie ciąg „print", a indeks w tablicy nazw (`co_names`), pod którym można znaleźć ten string.

3. Podobnie jak w przypadku `LOAD_NAME`, w samym kodzie bajtowym znajdziemy jedynie indeks w tablicy stałych (`co_consts`).

4. Pomijając skoki, kod bajtowy Pythona mógłby być w zasadzie porównany do Odwrotnej Notacji Polskiej.

Listing 1. Widok heksadecymalny na plik wynikowy

```
>hexdump hello.cpython-39.pyc
00000000: 61 0D 0D 0A 00 00 00 00 - 83 F2 49 61 15 00 00 00 |a        Ia    |
00000010: E3 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 |                |
00000020: 00 02 00 00 00 40 00 00 - 00 73 0C 00 00 00 65 00 |     @   s    e |
00000030: 64 00 83 01 01 00 64 01 - 53 00 29 02 7A 0B 48 65 |d     d S ) z He|
00000040: 6C 6C 6F 20 57 6F 72 6C - 64 4E 29 01 DA 05 70 72 |llo WorldN)    pr|
00000050: 69 6E 74 A9 00 72 02 00 - 00 00 72 02 00 00 00 FA |int  r    r     |
00000060: 08 68 65 6C 6C 6F 2E 70 - 79 DA 08 3C 6D 6F 64 75 | hello.py  <modu|
00000070: 6C 65 3E 01 00 00 00 F3 - 00 00 00 00             |le>          |
0000007c;
```

Nasz program sprowadza się do sześciu instrukcji, które kolejno:

» umieszczają na stosie funkcję `print`,
» umieszczają na stosie string „Hello World",
» wywołują zdjętą ze stosu funkcję z jednym argumentem (czyli `print('Hello World')`),
» usuwają ze stosu zwróconą przez `print` wartość None,
» umieszczają na stosie wartość None,
» pobierają wartość z góry stosu i ją zwracają (czyli istniejące niejawnie w kodzie `return None`), co powoduje zakończenie programu.

OBIEKT CODE

Efektem kompilacji kodu źródłowego nie jest jednak jedynie kod bajtowy, lecz również cała seria metadanych zapisanych w obiekcie klasy `code`. W trzeciej wersji Pythona klasa `code` nie należy do wąskiego grona podstawowych typów (takich jak `str` czy `int`), ale można się do niej odwołać, korzystając z modułu `types`:

```
>>> import types
>>> types.CodeType
<class 'code'>
```

Inspekcja obiektu `code` naszego programu będzie jednak odrobinę bardziej skomplikowana. Otóż najprostszym sposobem, żeby móc wejść w interakcję z tym obiektem, jest wymuszenie skompilowania naszego skryptu do pliku (`python -m compileall hello.py`), a następnie przyjrzenie się otrzymanemu w ten sposób plikowi wynikowemu *__pycache__/hello.cpython-39.pyc* (Listing 1).

Na szczęście nie musimy opierać naszej analizy bezpośrednio o dane binarne. Zamiast tego skorzystamy ze standardowej biblioteki Pythona o nazwie `marshal` w celu zdeserializowania powyższego pliku (uprzednio pomijając nieistotny dla nas 16-bajtowy nagłówek) i wypiszemy informacje o otrzymanym obiekcie typu `code` (`types.CodeType`). W tym celu posłużymy się następującym krótkim programem pomocniczym:

```
import marshal
with open("__pycache__/hello.cpython-39.pyc", "rb") as f:
  marshaled_obj = f.read()[16:]  # Zignoruj 16 bajtów nagłówka.
obj = marshal.loads(marshaled_obj)  # Zwróci obiekt typu code.

print("-=- Code object")
for field in dir(obj):
  if not field.startswith("co_"):
    continue
  print("  %-20s: %s" % (field, getattr(obj, field)))
```

Po jego uruchomieniu powinniśmy otrzymać całkiem sporo ciekawych informacji:

```
-=- Code object
  co_argcount         : 0
  co_cellvars         : ()
  co_code             : b'e\x00d\x00\x83\x01\x01\x00d\x01S\x00'
  co_consts           : ('Hello World', None)
  co_filename         : hello.py
  co_firstlineno      : 1
  co_flags            : 64
  co_freevars         : ()
  co_kwonlyargcount   : 0
  co_lnotab           : b''
  co_name             : <module>
  co_names            : ('print',)
  co_nlocals          : 0
  co_posonlyargcount  : 0
  co_stacksize        : 2
  co_varnames         : ()
```

Analizując wynik działania naszego skryptu, od razu można zauważyć, że większość pól jest albo pusta (jak `co_cellvars`, `co_freevars`, `co_lnotab`, `co_varnames`), albo wyzerowana (jak `co_argcount`, `co_kwonlyargcount`, `co_nlocals`, `co_posonlyargcount`). Wpływ na to ma zarówno prostota kodu źródłowego, jak i to, że mamy do czynienia z kodem znajdującym się w przestrzeni globalnej modułu – a więc wszelkie pola związane z metadanymi funkcji pozostają niewykorzystane.

Pełen opis wszystkich pól można znaleźć w dokumentacji modułu `inspect` [3], niemniej jednak chcielibyśmy zwrócić uwagę na kilka z nich:

- **`co_code`** – tablica bajtów (formalnie obiekt typu `bytes`) zawierająca wcześniej omówiony kod w formie skompilowanej, tj. binarnej,
- **`co_consts`** – krotka zawierająca wszystkie stałe używane przez kod; w przypadku naszego kodu jest to jedynie string „Hello World", oraz używane niejawnie `None`; w przypadku większych programów i modułów znalazłyby się tutaj obiekty `code` wszystkich funkcji globalnych, w tym tych odpowiedzialnych za faktyczne utworzenie zdefiniowanych w kodzie klas;
- **`co_names`** – krotka z używanymi w kodzie globalnymi „nazwami", a więc odniesieniami do globalnych funkcji, nazw, typów itd.,
- **`co_name`**, **`co_filename`**, **`co_firstlineno`** oraz (puste w naszym przypadku) **`co_lnotab`** – dane mapujące kod binarny na konkretne linie w kodzie źródłowym, co jest bardzo przydatne w momencie, gdy trzeba wyświetlić informacje o błędzie.

Gdybyśmy chcieli uruchomić nasz kod „Hello World", rekonstruując obiekt `code` z informacji wyświetlanych przez nasz pomocniczy program, musielibyśmy stworzyć najpierw sam obiekt klasy `code`, a następnie obiekt klasy `function`, który przywiązuje obiekt `code` do konkretnego zestawu zmiennych globalnych:

```python
import types
# Kolejność pól konstruktora CodeType może być inna
# w innych wersjach CPython.
# W razie czego: help(types.CodeTypes)
c = types.CodeType(
    0, 0, 0, 0, 5, 64,
    b'e\x00d\x00\x83\x01\x01\x00d\x01S\x00',
    ('Hello World', None),
    ('print',), (), 'hello.py',
    '<module>', 1, b'')
f = types.FunctionType(c, globals())
f()
```

Efekt działania podanego programu wygląda tak samo jak w jego pierwotnej wersji:

```
>python rec.py
Hello World
```

MASZYNA WIRTUALNA CPYTHON I FUNKCJA PRINT

Sercem maszyny wirtualnej wykonującej kod bajtowy jest funkcja `_PyEval_EvalFrameDefault`, którą można odnaleźć w źródłach Pythona w pliku *Python/ceval.c*. Sama funkcja jest dosyć długa – ma ponad 3000 linii – ale zawarte w niej implementacje poszczególnych operacji są dość krótkie i do pewnego stopnia czytelne nawet dla osób nie znających niuansów projektu CPython. Na przykład implementacja używanego w naszym skrypcie opkodu `POP_TOP` wygląda następująco:

```c
case TARGET(POP_TOP): {
    PyObject *value = POP();
    Py_DECREF(value);
    FAST_DISPATCH();
}
```

Szczegółową analizę kodu instrukcji jednak pominiemy – wykracza ona poza zakres tego artykułu – i zamiast tego przejdziemy do samej funkcji `print`.

Funkcję `print` – a w zasadzie jej implementację w języku C o nazwie `builtin_print` – można znaleźć w pliku *Python/bltinmodule.c*. Jej działanie można sprowadzić do następujących punktów:

1. Pobierz obiekt `sys.stdout` (lub inny przekazany w opcjonalnym argumencie `file`), z którego będą korzystać wszystkie operacje wypisywania.
2. Dla każdego nienazwanego argumentu funkcji:
   a. Jeśli to jest drugi lub kolejny argument, wypisz separator (domyślnie znak spacji).
   b. Wypisz argument.
3. Opcjonalnie wypisz znak końca linii.
4. Opcjonalnie opróżnij bufor, tj. wykonaj operację `flush()`.

Operacje wypisania są wykonywane za pomocą jednej z dwóch funkcji: `PyFile_WriteObject` oraz, dla literałów tekstowych jak spacja czy znak nowej linii, `PyFile_WriteString`. Ponieważ ta druga funkcja sprowadza się ostatecznie i tak do wywołania tej pierwszej, skupimy się na `PyFile_WriteObject`. Funkcję tę można znaleźć w pliku *Objects/fileobject.c*, a jej działanie, w uproszczeniu i pseudokodzie, wygląda następująco:

```
s = repr(obiektDoWypisania)
sys.stdout.write(s)
```

Jak można wywnioskować z powyższego kodu, `print` ostatecznie i tak wywołuje metodę `write` należącą do obiektu `sys.stdout`. Co za tym idzie, jeśli byśmy nadpisali ten obiekt i wprowadzili własną metodę `write`, to możemy przechwycić wszystko, co jest wypisywane za pomocą `print` – ilustruje to następujący eksperyment:

```
#!/usr/bin/python3
import sys
class BetterStdOut:
  def __init__(self, org):
    self._org = org

  def write(self, v):
    self._org.write(v.upper())

  def __getattr__(self, name):
    return getattr(self._org, name)

sys.stdout = BetterStdOut(sys.stdout)
print("wszędzie wielkie litery")
```

Po wykonaniu powinniśmy zobaczyć następujący wynik:

```
>python over.py
WSZĘDZIE WIELKIE LITERY
```

Listing 2. Pełny stos wywołań od print() w Pythonie do WriteConsoleW w WinAPI

```
KERNELBASE!WriteConsoleW
python39!_io__WindowsConsoleIO_write_impl+0x148 [\Modules\_io\winconsoleio.c @ 1004]
python39!_io__WindowsConsoleIO_write+0x7f [\Modules\_io\clinic\winconsoleio.c.h @ 319]
python39!method_vectorcall_O+0x9d [\Objects\descrobject.c @ 463]
python39!_PyObject_VectorcallTstate+0x2c [\Include\cpython\abstract.h @ 118]
python39!PyObject_VectorcallMethod+0x82 [\Objects\call.c @ 828]
python39!PyObject_CallMethodOneArg+0x2f [\Include\cpython\abstract.h @ 208]
python39!_bufferedwriter_raw_write+0x99 [\Modules\_io\bufferedio.c @ 1822]
python39!_bufferedwriter_flush_unlocked+0x7c [\Modules\_io\bufferedio.c @ 1871]
python39!buffered_flush_and_rewind_unlocked+0x12 [\Modules\_io\bufferedio.c @ 799]
python39!buffered_flush+0x77 [\Modules\_io\bufferedio.c @ 826]
python39!method_vectorcall_NOARGS+0xa0 [\Objects\descrobject.c @ 435]
python39!_PyObject_VectorcallTstate+0x2c [\Include\cpython\abstract.h @ 118]
python39!PyObject_VectorcallMethod+0x82 [\Objects\call.c @ 828]
python39!PyObject_CallMethodNoArgs+0x1c [\Include\cpython\abstract.h @ 198]
python39!_io_TextIOWrapper_write_impl+0x373 [\Modules\_io\textio.c @ 1724]
python39!_io_TextIOWrapper_write+0x3c [\Modules\_io\clinic\textio.c.h @ 411]
python39!cfunction_vectorcall_O+0xaa [\Objects\methodobject.c @ 513]
python39!_PyObject_VectorcallTstate+0x2e [\Include\cpython\abstract.h @ 119]
python39!PyObject_CallOneArg+0x2d [\Include\cpython\abstract.h @ 189]
python39!PyFile_WriteObject+0x5d [\Objects\fileobject.c @ 141]
python39!PyFile_WriteString+0x42 [\Objects\fileobject.c @ 165]
python39!builtin_print+0x132 [\Python\bltinmodule.c @ 1892]
```

SYSTEM NACZYŃ POŁĄCZONYCH I SYS.STDOUT

Wracając do naszego tytułowego programu, wiemy już, że „Hello World" zostanie ostatecznie przekazane do metody `write` obiektu `sys.stdout`. W wersji trzeciej Pythona `sys.stdout` jest w zasadzie stosem złożonym z trzech poziomów pomocniczych klas wejścia/wyjścia, odpowiedzialnych za konwersję i kodowanie, buforowanie oraz ostatecznie przekazanie danych do – w naszym przypadku – konsoli. Konkretniej, na stos ten składają się klasy:

» `io.TextIOWrapper` odpowiadającej za kodowanie podawanych metodzie `write` Unicode'owych stringów na UTF-8, opcjonalną konwersję sekwencji końca linii (w przypadku Windowsa będzie to przetłumaczenie `"\n"` na `"\r\n"`) oraz buforowanie na poziomie linii[5],
» `io.BufferedWriter` (widocznej w polu `sys.stdout.buffer`) odpowiadającej za buforowanie na poziomie bajtów,
» `io._WindowsConsoleIO` (widocznej w polu `sys.stdout.buffer.raw`) odpowiadającej za bezpośredni kontakt z API konsoli.

Warto zaznaczyć, że zestaw ten może być inny w przypadku, gdy wynik wykonania programu przekażemy do innego procesu (np. `python hello.py | more`) lub zapiszemy do pliku (np. `python hello.py > plik.txt`).

Samo przekazanie danych do konsoli odbywa się ostatecznie w implementacji metody `write` klasy `io._WindowsConsoleIO`, czyli w funkcji `_io__WindowsConsoleIO_write_impl` w pliku *Modules/_io/winconsoleio.c*. W skrócie, funkcja ta wykonuje dwie czynności:

1. Konwertuje otrzymywany strumień bajtów z UTF-8 na używany wewnętrznie w WinAPI UTF-16.
2. Wywołuje funkcję WinAPI `WriteConsoleW`, która eksportuje dane z naszego procesu do konsoli i której działaniu poświęcimy kolejną sekcję.

Wywołanie funkcji `WriteConsoleW` jest jednocześnie miejscem, w którym ostatecznie opuszczamy kod zarówno naszego skryptu w Pythonie, jak i samego środowiska wykonania CPython. Dotychczasową wędrówkę można zobaczyć w postaci stosu wywołań w Listingu 2.

5. Operacja `flush()` jest wykonywana po napotkaniu przynajmniej jednego znaku nowej linii w buforze, lub gdy liczba bajtów w buforze przekroczy 8192.

KONSOLE W SYSTEMIE WINDOWS

Na tym etapie wiemy już, że użycie funkcji `print` w języku Python prowadzi do wywołania przez interpreter systemowej funkcji `WriteConsoleW` na podanym ciągu znaków. Pozostaje jednak pytanie, w jaki sposób realizuje ona swoje zadanie. Aby zrozumieć zasadę działania tej funkcji, musimy najpierw cofnąć się o kilka, a nawet kilkadziesiąt lat wstecz.

Tryb tekstowy i wiersz poleceń od samego początku były nieodłączną częścią systemów operacyjnych rozwijanych przez firmę Microsoft. W MS-DOS służyły one za podstawowy sposób interakcji z komputerem, a w momencie wprowadzenia interfejsu graficznego w Windows 1.0 tekstowe komendy wciąż były jedynym sposobem na wykonanie wielu operacji. Wiersz poleceń nie odszedł w zapomnienie nawet po wydaniu pierwszego Windowsa z rodziny NT, który całkowicie uniezależnił się od środowiska DOS i oferował w pełni okienkowy tryb pracy. Podstawowa linia komend o nazwie *command.com* a później *cmd.exe* przetrwała w Windowsach przez kolejne 20 lat – aż do dziś, wciąż ciesząc się dużą popularnością wśród programistów, administratorów i power userów. Można śmiało powiedzieć, że jest to jeden z najstarszych, jeśli nie najdłużej istniejący Windowsowy program.

Warto zaznaczyć, że wiersz poleceń a tryb konsolowy to dwie bardzo blisko związane, ale jednak różne rzeczy. Linia komend opiera się na konsoli jako interfejsie wejścia/wyjścia, skąd pobiera polecenie lub serię poleceń do wykonania, a następnie wypisuje ich wynik. Może ona również realizować bardziej skomplikowane zadania, takie jak obsługa języków skryptowych. Przykładowe wiersze poleceń w systemie Windows to wspomniany już Command Prompt (*cmd.exe*) oraz PowerShell. Z kolei terminal to wewnętrzna część systemu operacyjnego, która odpowiada za graficzną otoczkę trybu tekstowego, czyli rysowanie okna, znaków tekstowych, umożliwianie zaznaczania/kopiowania tekstu itp. W jego skład można również zaliczyć implementację tzw. Console API [4], czyli zbioru funkcji systemowych umożliwiających operacje na konsoli (np. `GetConsoleFontSize`, `ReadConsole`, `SetConsoleTitle` itp.). Dzięki wbudowanemu wsparciu dla konsoli w Windowsie proste programy działające w trybie tekstowym nie muszą zajmować się graficznym

aspektem komunikacji z użytkownikiem – mogą wywołać kilka prostych funkcji API (lub wręcz funkcji biblioteki standardowej), a resztą zajmuje się system. W tym artykule skupimy się właśnie na szczegółach działania domyślnej implementacji konsol, dla której alternatywą jest od niedawna rozwijany przez Microsoft projekt Windows Terminal [5].

Aby dla danego procesu zostało przydzielone okno konsoli (nowe lub istniejące), musi wystąpić jeden z warunków:

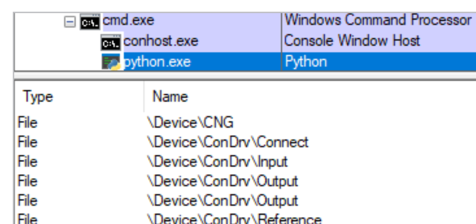
» Program został skompilowany jako konsolowy. W środowisku Visual Studio odpowiada za to flaga `/SUBSYSTEM:CONSOLE`, a w przypadku kompilatorów gcc i clang jest to opcja domyślna (o ile nie zostanie użyty parametr `-Wl,--subsystem,windows`, lub w skrócie `-mwindows`). W wynikowym pliku wykonywalnym świadczy o tym pole `Subsystem` w strukturze `IMAGE_OPTIONAL_HEADER`, które może być ustawione na stałą `IMAGE_SUBSYSTEM_WINDOWS_GUI` (program okienkowy) lub `IMAGE_SUBSYSTEM_WINDOWS_CUI` (program konsolowy).
» Program wywołał funkcję `AllocConsole` do utworzenia nowej konsoli, lub `AttachConsole` do przyłączenia się do konsoli istniejącego procesu.

Płynie z tego kilka ciekawych wniosków. Po pierwsze, kwestia tego, czy dany program jest konsolowy czy nie, jest w dużym stopniu umowna, gdyż zarówno aplikacja oznaczona jako konsolowa może wyświetlać okna, jak i aplikacja teoretycznie graficzna może stworzyć i korzystać z konsoli. Po drugie, obowiązuje zasada, że jeden program może w danym momencie operować na jednej konsoli, natomiast do jednej konsoli może być przypisanych wiele procesów. Dzieje się tak na przykład, kiedy wiersz poleceń *cmd.exe* uruchamia inny tekstowy program (jak choćby *python.exe*) – oba mają wtedy dostęp do wspólnego interfejsu tekstowego.

Przez długie lata wygląd i działanie konsoli w Windowsie nie zmieniały się – aż do czasów Windowsa 8.1 (włącznie) dostępne okna oferowały tylko najprostsze opcje dostosowywania. Użytkownik mógł m.in. zmieniać rozmiar okna, wybrać jeden z kilku predefiniowanych krojów i rozmiarów czcionki czy ustawić kolor tekstu i tła na jeden z 16 kolorów. Jedną z bardziej uciążliwych dolegliwości był brak możliwości kopiowania ciągłego tekstu rozbitego na dwie lub więcej linii – konsola pozwalała wyłącznie na zaznaczanie prostokątnych obszarów terminala. Archaiczna była również implementacja na poziomie systemu. Za obsługę i rysowanie okien odpowiedzialny był proces systemowy *csrss.exe*, komunikujący się ze wszystkimi innymi aplikacjami za pośrednictwem tzw. portów LPC. Nie dość, że skutkowało to oknami konsoli brzydszymi niż okna zwykłych aplikacji (np. w Windowsie XP ramki konsoli nie podlegały *stylowaniu* [6]), to jeszcze często prowadziło do różnorakich błędów bezpieczeństwa.

Pierwsza zmiana w powyższej architekturze nastąpiła w Windowsie 7. W tej wersji systemu obsługa konsoli została oddelegowana z CSRSS do osobnego procesu o nazwie *conhost.exe*, działającego już tylko z uprawnieniami zwykłego użytkownika. Jedynym obowiązkiem CSRSS pozostało uruchomienie programu *conhost.exe* w chwili, gdy kolejna aplikacja prosiła o nowe okno konsoli. Krok ten w znacznym stopniu poprawił bezpieczeństwo i stabilność systemu.

Kolejne zmiany nastąpiły w Windows 8, gdzie wprowadzony został sterownik trybu jądra o nazwie *condrv.sys*. Od tego czasu aż do dziś to on jest odpowiedzialny za uruchamianie *conhost.exe* dla programów z tekstowym interfejsem oraz przekazywanie między nimi informacji. Każdy program konsolowy ma kilka otwartych uchwytów do pseudo-plików obsługiwanych przez ten moduł, takich jak *\Device\ConDrv\Connect*, *\Device\ConDrv\Input*, *\Device\ConDrv\Output* itd. Za ich pośrednictwem programy tekstowe mogą czytać, pisać i zmieniać właściwości konsoli, a wysyłane przez nie zapytania są odpowiednio opakowywane i przekazywane do *conhost.exe*, który wykonuje faktyczne operacje na oknie lub buforach wejścia/wyjścia. Zarówno pomocniczy proces conhost, jak i nazwy aktywnych uchwytów można z łatwością zaobserwować przy użyciu narzędzia Process Explorer, co zostało przedstawione na Rysunku 2. Z kolei ogólny zarys architektury konsol w systemach Windows 8 i nowszych pokazano na Rysunku 3.

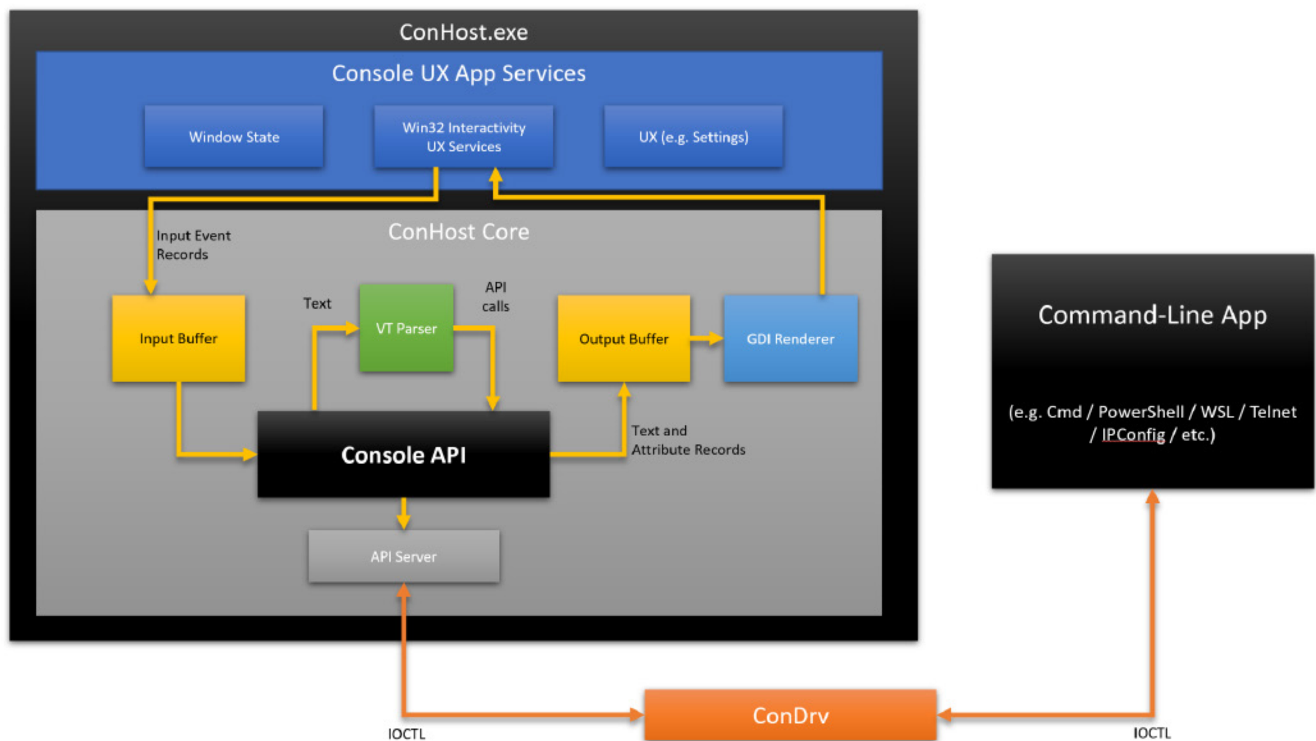


Rysunek 2. Process Explorer pokazujący proces conhost.exe i uchwyty w python.exe

Podczas rozwijania systemu Windows 10 Microsoft postawił jeszcze większy nacisk na unowocześnienie konsol, dzięki czemu możliwe stało się wspomniane wcześniej zaznaczanie tekstu liniami, wyświetlanie kolorów z 24-bitowej palety czy też pełne wsparcie dla formatowania tekstu przy użyciu tzw. ANSI Escape Codes (nazywanych przez Microsoft *Virtual Terminal Sequences*). Według oficjalnych źródeł [8] w 2014 r. Microsoft powołał specjalny zespół mający na celu poprawę jakości kodu konsoli i rozszerzenie ich możliwości przy jednoczesnym zachowaniu kompatybilności wstecznej. Jednym z korzystnych skutków ubocznych tej inicjatywy (z perspektywy badacza) było otwarcie znacznej części kodu źródłowego nowego terminala Windows Terminal, a także programu pomocniczego conhost, w serwisie GitHub [9]. Oba te komponenty zaimplementowane są w całości w C++. Dzięki temu możemy dowiedzieć się bardzo wiele o działaniu konsol u źródła, bez konieczności odwoływania się do inżynierii wstecznej.

W tym momencie warto doprecyzować jeszcze jeden szczegół techniczny. Kod odpowiadający za wyświetlanie konsoli zawiera dwa silniki renderowania: jeden oparty na tradycyjnym interfejsie graficznym GDI oraz drugi – nowszy – oparty o DirectX. Ten ostatni wspiera nowoczesne rozszerzenia czcionek, takie jak fonty zmienne (ang. *variable fonts*, czyli fonty, w których z dowolną dokładnością można kontrolować pewne parametry, jak grubość znaków) czy fonty kolorowe (umożliwiające np. wyświetlanie symboli emoji). Silnik DirectX jest używany przez wiersz poleceń Windows Terminal, natomiast zwykłe konsole (włącznie z tą używaną przez *cmd.exe*) domyślnie korzystają z interfejsu GDI, o ile nie ustawi się w rejestrze nieudokumentowanej wartości *HKCU\Console\UseDx* typu `REG_DWORD` na 1. My jednak nie będziemy tego robić i w tym artykule prześledzimy ścieżkę wykonania prowadzącą przez GDI.

Wróćmy jednak do tematu przewodniego artykułu. Funkcja `WriteConsoleW` z biblioteki *kernelbase.dll*, do której trafiliśmy



Rysunek 3. Architektura konsoli w Windowsie (źródło: [7])

w Pythonie, wysyła do pseudo-pliku *\Device\ConDrv\Output* wiadomość IOCTL o kodzie `IOCTL_CONDRV_ISSUE_USER_IO` (czyli liczbowo 0x500016). Brzmi to skomplikowanie, jednak oznacza tylko tyle, że prośba o wypisanie tekstu zostaje przekazana do wspomnianego wcześniej sterownika *condrv.sys*. Ten natomiast przekazuje tę informację dalej do procesu *conhost.exe*, który, w zależności od operacji, decyduje, co zrobić z nią dalej. Od tego momentu możemy podążać już za otwartym kodem źródłowym.

Główna funkcja odbierająca rozkazy w *conhost.exe* to `ConsoleIoThread` (zaimplementowana w *terminal/src/host/srvinit.cpp*). Po otrzymaniu nowej wiadomości wykonywanie przechodzi przez kolejne metody:

» `IoSorter::ServiceIoOperation`
» `IoDispatchers::ConsoleDispatchRequest`
» `ApiSorter::ConsoleDispatchRequest`

Następnie trafiamy do jednej z funkcji z przestrzeni nazw `ApiDispatchers`, która obsługuje dany typ operacji. Gdybyśmy chcieli na żywo prześledzić, jakie dokładnie niskopoziomowe operacje są wykonywane na danej konsoli, to wystarczy, że podepniemy się pod conhost debuggerem WinDbg, załadujemy symbole dla bibliotek systemowych z serwera Microsoftu, a następnie ustawimy breakpointy na wszystkie funkcje z przestrzeni `ApiDispatchers` przy użyciu następującej komendy:

```
bm conhost!ApiDispatchers::*
```

Na testowanym Windowsie 10 skutkuje to ustawieniem 52 punktów wstrzymania, które następnie są aktywowane w momencie jakichkolwiek zmian w oknie konsoli. W przypadku wypisywania tekstu wykonanie dociera do funkcji `ApiDispatchers::ServerWriteConsole`, czyli odpowiednika funkcji `WriteConsole` wywołanej po stronie Pythona. Dalej procesor podąża kolejnymi wywołaniami:

» `ApiRoutines::WriteConsoleWImpl`
» `WriteConsoleWImplHelper`
» `DoWriteConsole`
» `WriteChars`
» `WriteCharsLegacy`

W tej ostatniej funkcji wykonywana jest lwia część pracy zapisania tekstu do bufora wyjściowego (posiłkując się dalszymi klasami: `SCREEN_INFORMATION`, `TextBuffer` i `ROW`). Warto zaznaczyć, że na tym etapie nie dochodzi jeszcze do rysowania znaków w oknie – tym zajmuje się osobny, dedykowany wątek graficzny. I tak po zapisaniu danych do bufora wyjściowego zostają one odczytane i wyświetlone np. pod następującym stosem wywołań, podczas rysowania nowej klatki okna:

```
#0 Microsoft::Console::Render::GdiEngine::_FlushBufferLines
#1 Microsoft::Console::Render::GdiEngine::UpdateDrawingBrushes
#2 Microsoft::Console::Render::Renderer::_UpdateDrawingBrushes
#3 Microsoft::Console::Render::Renderer::_PaintBufferOutput
#4 Microsoft::Console::Render::Renderer::_PaintFrameForEngine
#5 Microsoft::Console::Render::Renderer::PaintFrame
#6 Microsoft::Console::Render::RenderThread::_ThreadProc
```

Obecnie w systemie Windows 10 konkretną funkcją GDI używaną przez conhost do wyświetlenia tekstu jest `PolyTextOutW`, choć w repozytorium Microsoftu na GitHubie możemy znaleźć informację, że wywołanie to zostało zmienione na `ExtTextOutW` w połowie 2021 r. [10]. To właśnie te funkcje odpowiadają za przekształcenie znaków zakodowanych w UTF-16 na kształty liter, które potem widzimy na ekranie.

RASTERYZACJA TEKSTU

Jak powszechnie wiadomo, fonty to pliki zawierające graficzną reprezentację znaków, które pozwalają na wyświetlanie i drukowanie tekstu o różnych krojach, rozmiarach i innych cechach. We wczesnych latach istnienia komputerów były one zapisywane w formacie bitmapowym – tzn. dla każdej pary glifu i jego wielkości przypisany był dwukolorowy obrazek reprezentujący jego wygląd. Rozwiązanie takie miało jednak wiele oczywistych wad, w związku z czym niewiele później zostały one wyparte przez fonty wektorowe, które opisują kształty przy użyciu krzywych. Pierwszym formatem tego typu były zaprojektowane przez Adobe w 1984 r. fonty Type 1 (blisko związane z językiem PostScript), a w latach 90. dołączyły do tego grona m.in. formaty TrueType (autorstwa Apple) i OpenType (opracowane przez Microsoft i Adobe). Te dwa ostatnie przyjęły się jako obowiązujące i są szeroko stosowane do dziś.

Wraz z rozwojem cyfrowej typografii i formatów wektorowych zwiększał się (i wciąż zwiększa) stopień skomplikowania fontów i co za tym idzie kodu, który je obsługuje. Z oficjalnej specyfikacji OpenType [11] dowiadujemy się, że istnieje ok. 50 rodzajów tabel SFNT (czyli mniejszych „bloczków", z których zbudowane są fonty), z czego osiem jest wymienionych jako obowiązkowe. Każda z nich reprezentuje inny rodzaj danych (zarówno binarnie, jak i koncepcyjnie), a równocześnie wiele z nich wzajemnie na siebie oddziałuje. Jeśli dodamy do tego fakt, że zarówno fonty TrueType, jak i OpenType zawierają w sobie mini-programy opisujące kształty i sposób rysowania poszczególnych znaków (z własnym stosem, operatorami arytmetycznymi, warunkowymi itd.), to rysuje nam się obraz plików, które bardzo trudno jest obsłużyć w sposób poprawny i bezpieczny. Jednocześnie większość dzisiejszych silników operujących na czcionkach ma swoje korzenie w kodzie z lat 90., a więc są napisane w C i C++ (językach znanych z notorycznych przepełnień bufora i podobnych błędów) i według standardów obowiązujących w tamtych czasach.

Pomimo potencjalnych niebezpieczeństw związanych z operowaniem na fontach Windowsy w wersji do 8.1 włącznie obsługiwały je z poziomu jądra, czyli najbardziej uprzywilejowanego trybu działania systemu operacyjnego. Konkretnie, w zależności od formatu, za daną czcionkę odpowiedzialny był jeden z dwóch sterowników systemowych:

» *win32k.sys*, czyli główny sterownik graficzny Windows, implementujący obsługę prostych fontów bitmapowych i wektorowych (rozszerzenia *.fon* i *.fnt*) oraz formatu TrueType (rozszerzenia *.ttf* i *.ttc*).

» *atmfd.sys*, którego nazwa rozwija się do *Adobe Type Manager Font Driver*, implementujący obsługę formatów Type 1 (rozszerzenia *.pfb*, *.pfm* i *.mmm*) i OpenType (rozszerzenia *.otf* i *.otc*).

Jednym z niewątpliwych plusów takiego rozwiązania była jego wydajność – podczas rasteryzacji i wyświetlania tekstu nie dochodziło do zbyt wielu przełączeń kontekstu (ang. *context switch*), dzięki czemu m.in. pamięć podręczna CPU była efektywnie wykorzystywana, a procesor płynnie przechodził pomiędzy kolejnymi etapami tłumaczenia znaków na kształty geometryczne, następnie na piksele, i wyświetlania ich na ekranie. To z kolei skutkowało szybkim i responsywnym interfejsem graficznym. Z drugiej jednak strony każdy potencjalny błąd w tym kodzie narażał użytkownika na całkowite przejęcie kontroli nad systemem, jeśli w jakiś sposób doszło do otwarcia specjalnie spreparowanej czcionki (np. osadzonej w dokumencie, stronie internetowej itp.). I faktycznie, przez lata Microsoft naprawił dziesiątki podatności znalezionych w silniku fontów przez badaczy z całego świata, a temat ten był poruszany na wielu konferencjach poświęconych bezpieczeństwu komputerowemu.

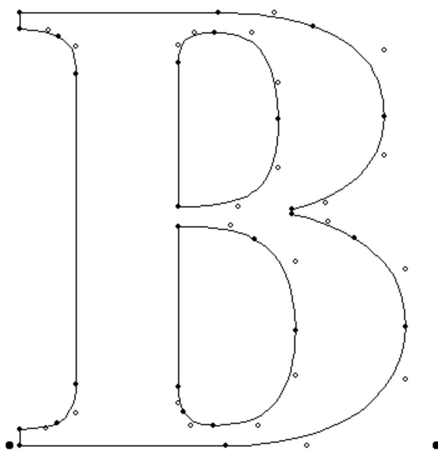
Podobnie jak w przypadku konsol, znaczna poprawa w powyższej architekturze nastąpiła wraz z nadejściem Windowsa 10. W tej wersji systemu kod odpowiedzialny za obsługę fontów został przeniesiony ze sterowników *win32k.sys* i *atmfd.dll* do pomocniczego programu trybu użytkownika o nazwie *fontdrvhost.exe*, działającego z minimalnymi uprawnieniami w tzw. piaskownicy (ang. *sandbox*). Dzięki tej zmianie, nawet jeśli w obsłudze któregoś z formatów czcionek znajdzie się kolejny błąd prowadzący do wykonania dowolnego kodu, wciąż jest długa droga do przejęcia pełnej kontroli nad komputerem użytkownika. Ponadto mniej poważne błędy skutkujące nieobsłużonym wyjątkiem nie zawieszają już całego systemu, tylko prowadzą do tymczasowego wyłączenia pomocniczego procesu, który po chwili jest restartowany. Warto przy tym nadmienić, że sterownik *win32k.sys* (który w Windows 10 został rozbity jeszcze na dwa kolejne moduły: *win32kbase.sys* i *win32kfull.sys*) wciąż odpowiada za interfejs graficzny Windowsa, a wycięty został jedynie kod ściśle związany z fontami. Z kolei sterownik *atmfd.dll* całkowicie stracił rację bytu, więc w ogóle nie znajdziemy go już na dysku w najnowszych wersjach systemu.

Jeśli chcielibyśmy lepiej zrozumieć sposób działania procesu pomocniczego *fontdrvhost.exe*, musielibyśmy przeanalizować ten właśnie plik wykonywalny. Niestety w przeciwieństwie do konsol kod źródłowy tej części systemu operacyjnego nie został publicznie udostępniony, w związku z czym pozostaje nam tylko zakasać rękawy i uruchomić disassembler. Pewnym pocieszeniem mogą być tutaj symbole debugowania (pliki *.pdb*) udostępniane przez Microsoft dla bibliotek systemowych za pośrednictwem oficjalnego serwera [12], dzięki którym możemy przynajmniej poznać nazwy poszczególnych funkcji. I tak po krótkiej analizie funkcji `main` możemy dowiedzieć się, że wątek komunikujący się z jądrem działa w funkcji o nazwie `ServerRequestLoop`, która odbiera informacje o kolejnych operacjach do wykonania i zwraca ich rezultat przez wywołanie systemowe `NtGdiExtEscape`. Następnie wywoływana jest funkcja `DispatchRequest`, która przekazuje wykonanie do kolejnych funkcji, w zależności od konkretnego żądania i formatu czcionki, którego dotyczy. Szczegółowa analiza toku wykonania *fondrvhost.exe* wykracza poza zakres tego artykułu, ale zachęcamy zainteresowanych czytelników do dalszego zagłębienia się w ten temat.

Zróbmy jednak krok w tył i przeanalizujmy proces wyświetlania napisu „Hello World" na wyższym poziomie abstrakcji. W domyślnej konfiguracji systemu Windows wiersz poleceń używa czcionki Consolas, której odpowiada plik *C:\Windows\Fonts\consola.ttf*; a więc mamy do czynienia z formatem TrueType. Pierwszą operacją,

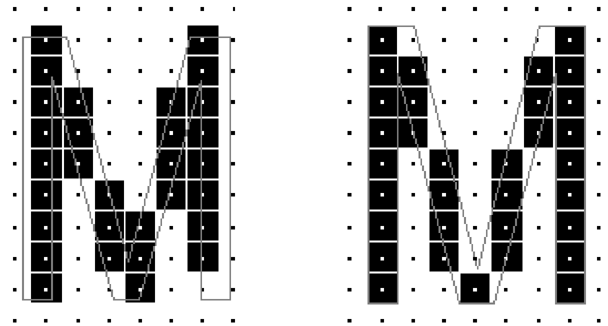
którą wykonuje interfejs GDI po wywołaniu funkcji `PolyTextOutW`, jest tzw. kształtowanie tekstu (ang. *text shaping*). Ponieważ jednak mamy tutaj do czynienia z alfabetem łacińskim oraz czcionką o stałej szerokości, to etap ten sprowadza się głównie do przetłumaczenia punktów kodowych (ang. *code points*) każdego ze znaków na odpowiadający mu identyfikator glifu (kształtu zapisanego w foncie). Z oczywistych powodów nie mają tutaj zastosowania np. kerning (regulowanie odległości między konkretnymi parami znaków), ligatury (łączenie dwóch lub więcej liter w jedną) czy inne bardziej zaawansowane transformacje.

Następnie, na podstawie tabeli `glyf`, dla każdego glifu w tekście rysowany jest jego kontur. Kształty geometryczne w TrueType są reprezentowane przez zbiór punktów na płaszczyźnie, pomiędzy którymi prowadzone są linie proste oraz krzywe Béziera drugiego stopnia. Te ostatnie są zdefiniowane przez trzy punkty: dwa znajdujące się na końcach krzywej, a także tzw. punkt kontrolny określający jej kształt. W uproszczeniu przypomina to popularną grę dla dzieci w łączenie kropek na kartce papieru w celu otrzymania określonego rysunku, z tym że komputer robi to w sposób powtarzalny i z wysoką precyzją. Na Rysunku 4 przedstawiono przykładowy kontur litery „B" opisany w formacie TrueType, gdzie wypełnione punkty to te znajdujące się na krzywej, a pomocnicze punkty kontrolne zostały oznaczone jako puste.



Rysunek 4. Kształt litery B zapisanej w formacie TrueType (źródło: [13])

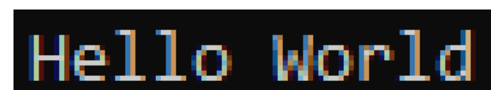
Kolejnym ważnym etapem rasteryzacji jest tzw. hinting. Jego potrzeba bierze się stąd, że odwzorowanie kształtu danej litery na ograniczonej liczbie pikseli – co ma miejsce szczególnie przy małych rozmiarach tekstu – jest trudna do osiągnięcia w sposób automatyczny. Nawet jeśli dane odwzorowanie jest wierne z matematycznego punktu widzenia, to w praktyce może być niesymetryczne, w inny sposób nieprzyjemne dla oka, lub całkowicie nieczytelne. Z tego też powodu wszystkie wiodące formaty fontów wektorowych mają wsparcie dla hintowania, czyli zaprogramowania w czcionce pewnych podpowiedzi co do tego, jak dany znak należy narysować w określonych rozmiarach. Obejmuje to informacje takie jak to, jak optymalnie dopasować kontur do siatki pikseli, albo to, które elementy litery są nieważne i mogą zostać pominięte w niskiej rozdzielczości, a które wręcz przeciwnie – muszą się znaleźć w wynikowym obrazie. Przykład tego, jak mocno hinting może wpłynąć na efekt końcowy rasteryzacji, pokazano na Rysunku 5. Jako ciekawostkę warto wspomnieć, że w formatach Type 1 i OpenType zarówno rysowanie konturów, jak i hinting odbywa się w tym samym czasie, podczas wykonywania przez interpreter silnika fontów mini-programów o nazwie „CharStrings".



Rysunek 5. Przykład litery M w postaci bitmapy bez i z włączonym hintowaniem (źródło: [14])

Co istotne, programy hintujące nie są uruchamiane w każdej klatce, w której wyświetlana jest dana litera. Bardzo ważną rolę w implementacji obsługi fontów pełni mechanizm *cachingu*, który zapisuje w pamięci raz wygenerowaną bitmapę dla znaku o danej wielkości i używa jej w przyszłości bez powtarzania wszystkich obliczeń.

Skąd jednak bierze się ta bitmapa? Tutaj dochodzimy do ostatniej części całego procesu, czyli faktycznej konwersji konturu znaku na obraz rastrowy (ang. *scan conversion*). Przy okazji zapalania kolejnych pikseli na biały i czarny (lub inny) kolor system, w zależności od ustawień, może zastosować dodatkowe kroki, aby sprawić, by tekst wyglądał lepiej na ekranie komputera. Jedną z takich metod jest wygładzanie krawędzi (ang. *antialiasing*), które może operować nie tylko na odcieniach szarości, ale wręcz na całej palecie kolorów, wykorzystując ułożenie subpikseli RGB składających się na każdy piksel na matrycy monitora (tzw. subpixel rendering). Kiedy przyjrzymy się bliżej poszczególnym pikselom w naszym napisie, prawdopodobnie okaże się, że prawie żaden z nich nie jest faktycznie szary (Rysunek 6), choć właśnie ten kolor jest rejestrowany przez ludzkie oko w oryginalnej rozdzielczości.



Rysunek 6. Wygładzanie krawędzi przez technologię Microsoft ClearType

Kiedy „drukowane" przez nasz prosty program znaki przybierają już formę nie wektorową, a bitmapową, ich dalszym losem zajmuje się układ graficzny, czyli GPU – czy to umieszczony na karcie graficznej, czy też zintegrowany z procesorem. O ile istnieje tryb tekstowy, to jednak współczesne wersje systemu Windows nie mogą w nim normalnie działać, a zawsze wyświetlają środowisko graficzne. Konsola tekstowa, w ramach której wykonuje się nasz program, ma więc postać jednego z okien wyświetlanych na wirtualnym pulpicie.

ZARZĄDZANIE OKNAMI

Zarządzaniem oknami zajmuje się część systemu o nazwie Desktop Window Manager (DWM). Aplikację *dwm.exe* możemy znaleźć na

liście uruchomionych procesów. Jej zabicie powoduje zniknięcie całego obrazu na monitorze, który jednak po chwili wraca, ponieważ system uruchamia DWM ponownie. To on wyświetla pulpit, pasek zadań i okna poszczególnych aplikacji – czy to będzie konsola tekstowa (jak w naszym przypadku), przeglądarka internetowa, odtwarzany film czy też jakaś gra. Pozwala też oknom zachodzić na siebie – tym z przodu zasłaniać te z tyłu. Proces porządkowania tego wszystkiego, aby wyświetlić finalny obraz na monitorze, nazywany jest komponowaniem (ang. *compositing*).

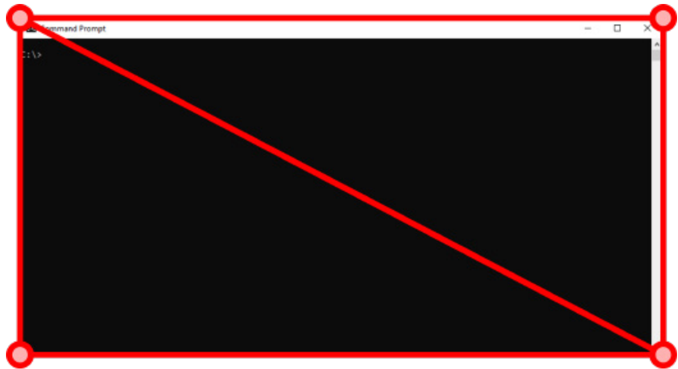
Do wydajnego zrealizowania tego zadania wykorzystywana jest akceleracja – przyspieszenie sprzętowe oferowane przez układ graficzny. Oznacza ono wykonywanie konkretnego zadania (w naszym przypadku – wyświetlania grafiki) szybciej dzięki specjalizowanym układom sprzętowym, w porównaniu z realizowaniem tej samej logiki w pełni programowo, czyli przez procesor główny. Ponieważ jednak na rynku istnieje niejeden producent układów graficznych, a każdy z nich stosuje w swoim sprzęcie inne rozwiązania, do zapewnienia kompatybilności potrzebny jest zainstalowany w systemie sterownik graficzny odpowiedni do danej karty czy układu graficznego. Bez niego system nadal potrafi wyświetlać pulpit, ale robi to z użyciem Microsoft Basic Display Adapter – uproszczonej implementacji. W niej całe renderowanie (generowanie) grafiki, łącznie ze żmudnym kopiowaniem z miejsca na miejsce bajtów reprezentujących poszczególne piksele, odbywa się po stronie CPU, a więc działa bardzo wolno.

Pewne formy sprzętowego przyspieszania wyświetlania grafiki istniały w komputerach od zamierzchłych czasów. Dawniej była to wyłącznie grafika dwuwymiarowa (2D). Wspierane operacje polegały więc głównie na kopiowaniu prostokątnych fragmentów obrazu w pamięci, co w świecie gier znane jest jako „duszki" (ang. *sprites*). W ten sposób można wydajnie renderować grafikę takich gier jak platformówki typu „Mario".

Współcześnie jednak większość gier używa grafiki trójwymiarowej (3D), a układy graficzne taką grafikę potrafią wydajnie generować. Od czasu, kiedy u szczytu popularności były pierwsze trójwymiarowe strzelanki typu „Doom" i „Quake", przyjął się schemat komponowania grafiki 3D z trójkątów. Tak opisana grafika jest wyświetlana przez obecnie pojawiające się na rynku gry, które do wydawania poleceń karcie graficznej używają jednego z tzw. API graficznych: OpenGL, Direct3D lub Vulkan.

Dlaczego w ogóle mówimy o grach, skoro naszym przedmiotem zainteresowania jest tutaj prosty program „Hello World"? Nawet jeśli duża grupa użytkowników komputera nie gra w gry, a jedynie używa swojego sprzętu do przeglądania Internetu lub zastosowań biurowych, to jednak od dawna gry wideo wyznaczają kierunek i narzucają tempo rozwojowi układów graficznych. Do wyświetlania pulpitu nie potrzeba wielu TFLOPS-ów mocy obliczeniowej ani 300 W energii, jaką pobierają najszybsze karty graficzne. Wystarczy produkt z niższej półki. Wszystkie one jednak działają na tej samej zasadzie. Różnią się jedynie ilością pamięci, liczbą rdzeni do wykonywania shaderów i innymi parametrami decydującymi o ich wydajności w grach.

Pewnym zaskoczeniem może być wiadomość, że menadżer okien w nowych wersjach Windowsa używa grafiki 3D także do rysowania pulpitu! Choć okna są ze swej natury płaskie, to jednak przy ogromnej mocy obliczeniowej, jaką oferują karty graficzne w tym konkretnym zastosowaniu, opłacalne okazało się renderowanie każdego z nich ich jako prostokąta złożonego z dwóch trójkątów, na którego powierzchnię obraz przedstawiający treść okna zostaje nałożony jako tekstura, niczym tapeta przyklejona na ścianie – Rysunek 7. W pewnym sensie więc, z punktu widzenia karty graficznej, praca z poważnymi aplikacjami biurowymi nie różni się wiele od grania w gry :).



Rysunek 7. Okno jako siatka trójkątów

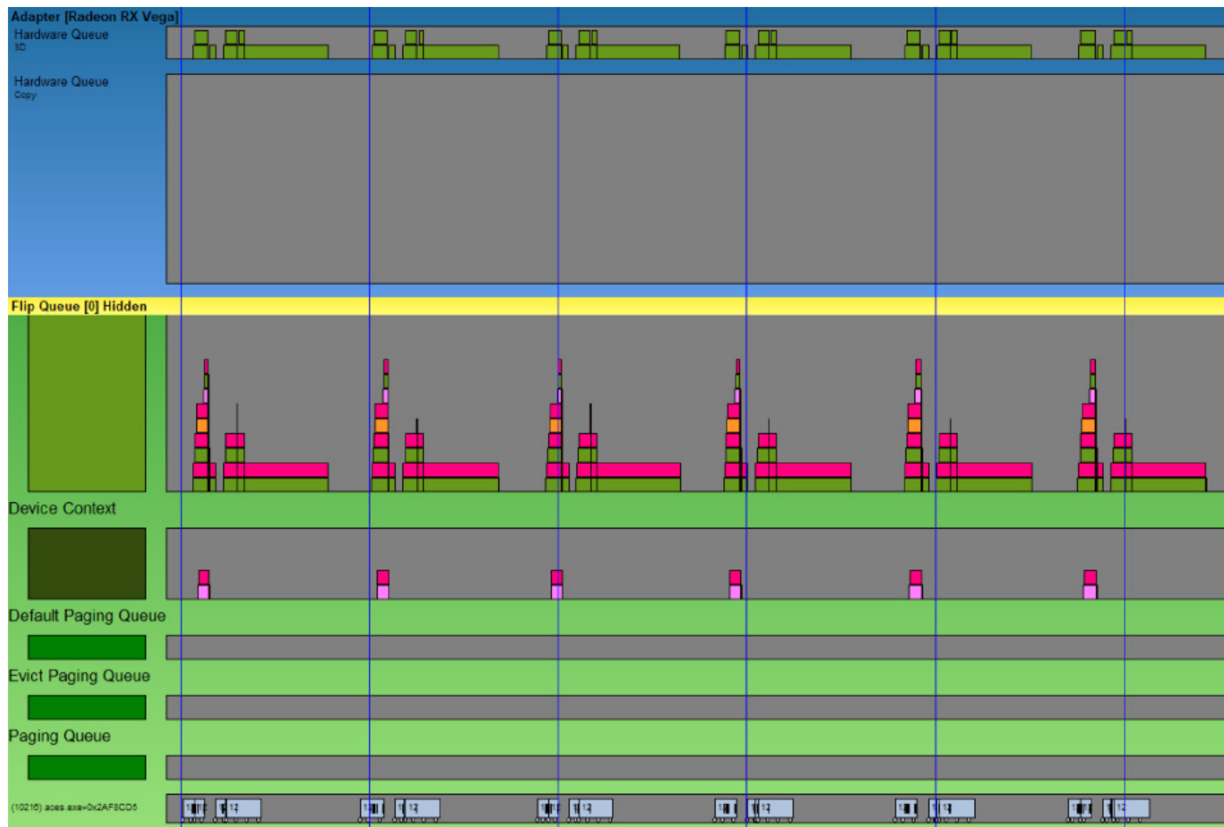
GPU

Układ graficzny, jako koprocesor, działa równolegle z procesorem głównym komputera. Programista implementujący wyświetlanie grafiki ma za zadanie wypełnić bufor komendami do wykonania przez GPU (nic dziwnego, że ten nosi nazwę… *command buffer*), a następnie podać go do wykonania. Procesor może zaraz potem zająć się innymi obliczeniami, podczas gdy układ graficzny wykonuje te komendy w swoim tempie. W ten sposób działają też gry. Podczas kiedy karta graficzna mozolnie odrysowuje wszystkie trójwymiarowe obiekty i postacie, by na końcu wyświetlić je na monitorze jako następna klatka N, CPU już wykonuje obliczenia na świecie gry (symulacja fizyki, sztuczna inteligencja przeciwników, odtwarzanie dźwięku itd.) dla klatki N+1.

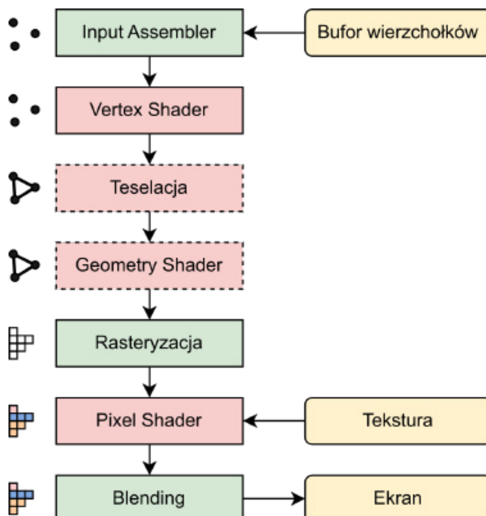
Zachowanie to można zaobserwować na przykład z użyciem darmowego narzędzia Microsoftu: GPUView. Program ten daje wgląd w bardzo niskopoziomowe działanie CPU i GPU. Z jego użyciem możemy „nagrać", co dzieje się w systemie w ciągu określonego czasu, a następnie zobaczyć aktywność GPU, jak i poszczególnych wątków CPU. Przedstawione tam informacje mogą być trudne do zinterpretowania, ale wszystkim zainteresowanym tym, co się dzieje w komputerze blisko sprzętu, polecamy nauczyć się przynajmniej podstawowego rozumienia tego typu diagramów. Podobne występują bowiem także w innych narzędziach.

Na Rysunku 8 widzimy kontekst Direct3D wykonujący bufory komend [15]. Oś pozioma to czas biegnący w prawo. W górę natomiast piętrzą się kolejne bufory zakolejkowane do wykonania. W każdym momencie „bloczek" znajdujący się na samym dole to ten, który jest aktualnie wykonywany. Wszystkie nad nim natomiast to polecenia znajdujące się dalej w kolejce, oczekujące na swoją kolej.

W samym GPU z kolei wykonywanie komend odbywa się w sposób potokowy, co oznacza, że każda komenda przechodzi przez kolejne etapy tzw. potoku graficznego zanim końcowe piksele trafią na ekran. Uproszczony schemat potoku przedstawiono na Rysunku 9.



Rysunek 8. Zrzut ekranu z programu GPUView



Rysunek 9. Schemat potoku graficznego

Etapy oznaczone na zielono są jedynie konfigurowalne, to znaczy wykonują określoną operację, której możemy co najwyżej zmieniać parametry. Te oznaczone na czerwono są natomiast programowalne, co znaczy, że sami musimy napisać program wykonywany na tym etapie. Takie programy dla układów graficznych nazywamy shaderami i piszemy w specjalnych językach – HLSL lub GLSL. Choć języki te mają składnię podobną do C czy C++, to jednak same shadery różnią się od programów wykonywanych na CPU w wielu aspektach. W Listingu 3 pokazano przykładowy pixel shader, który próbkuje (sampluje) teksturę, otrzymany kolor mnoży przez kolor wierzchołka i wynik zwraca jako kolor przeznaczony do zapisania na ekran.

Listing 3. Przykładowy, prosty pixel shader napisany w języku HLSL

```
Texture2D texture0 : register(t0);
SamplerState sampler0 : register(s0);

struct VERTEX
{
    float4 position: SV_POSITION;
    float2 textureCoordinate : TEXCOORD0;
    float4 color: COLOR0;
};

float4 pixelShaderMain(VERTEX v) : SV_TARGET
{
    return
        texture0.Sample(sampler0, v.textureCoordinate) *
        v.color;
}
```

Poniżej krótkie omówienie etapów potoku graficznego:

Input Assembler to jednostka, która pobiera pozycje i inne parametry wierzchołków, czyli punktów w przestrzeni 3D. Ich źródłem jest bufor wierzchołków (ang. *vertex buffer*) oraz opcjonalny bufor indeksów (ang. *index buffer*), który zawiera indeksy tych wierzchołków.

Vertex Shader to pierwszy z etapów programowalnych. Możemy, a nawet musimy na tym etapie przetworzyć dane każdego wierzchołka, na przykład zmieniając jego pozycję, aby znalazł się w pożądanym miejscu na ekranie. To dzięki tej operacji jeden obiekt, czy to będzie kamień albo potwór w grze, czy też okno na pulpicie, może się znaleźć na ekranie w wielu kopiach, a każda w innym miejscu i w innym rozmiarze.

Teselacja oraz Geometry Shader to etapy opcjonalne, które pozwalają na dodatkowe przetwarzanie całych trójkątów, np. zagęszczanie siatki. Możemy w nich użyć kolejnych rodzajów shaderów. Działają jednak dosyć wolno i dlatego są rzadko używane.

Rasteryzacja to operacja, w której karta graficzna wyznacza piksele pokrywające dany trójkąt na ekranie. Od tej pory nie mówimy już o wierzchołkach czy trójkątach, a o pojedynczych pikselach. Ze świata grafiki wektorowej przechodzimy więc do bitmapowej.

Zadaniem pixel shadera jest wyliczyć kolor danego piksela. Używane do tego są tzw. tekstury, czyli bitmapowe obrazy w pamięci video przedstawiające kolor danej powierzchni. W grach może to być np. zdjęcie powierzchni drewnianej czy zardzewiałego metalu. W Windowsie… obraz naszego okna lub terminala. Pixel shader próbkuje (sampluje) piksele tej tekstury w odpowiednich miejscach i zwraca ich kolor. W grach może dodatkowo wykonywać obliczenia, aby dodać efekty oświetlenia, cienia, mgły i inne potrzebne do uzyskania realistycznej grafiki.

Jeżeli zdamy sobie sprawę, że w każdej klatce i dla każdego piksela na ekranie musi się co najmniej raz wykonać cały pixel shader, aby wyliczyć kolor tego piksela, to możemy sobie wyobrazić, jak ogromna moc obliczeniowa drzemie we współczesnych kartach graficznych. Na przykład przy rozdzielczości 4K i 30 klatkach na sekundę mamy 248,832,000 pikseli renderowanych w każdej sekundzie. Co najmniej tyle razy musi się więc wykonać pixel shader – nie jego jedna instrukcja, a cały program!

Wreszcie, blending to operacja zapisania końcowego koloru piksela do pamięci video, aby potem mógł zostać wysłany do monitora. Jego kolor może zostać przy tym zmieszany z poprzednim, dzięki czemu uzyskamy efekt półprzezroczystości – będzie o tym mowa dalej.

Traktowanie okien pulpitu jako tekstur ma jeszcze jedną zaletę: system może je łatwo i szybko wyświetlać w wielu miejscach i na wiele sposobów. Pomniejszony obraz okna pokazuje się nam choćby po wciśnięciu Alt+Tab, Win+Tab bądź też po najechaniu kursorem myszy na przycisk reprezentujący uruchomiony program na pasku zadań. Ten drugi skrót klawiszowy zwykł wyświetlać listę okien zawieszonych w przestrzeni 3D, jednak w nowych wersjach Windows 10 twórcy systemu powrócili do płaskiej galerii. Wszystkie te miejsca jednak mają dostęp do obrazu okna na żywo, pokazują więc nawet odtwarzany w danej chwili film. Możliwe jest też tworzenie w prosty sposób okien półprzezroczystych lub takich o nieregularnych kształtach, choć nie spotykamy ich w zbyt wielu programach.

ALFA-BLENDING

W tym miejscu warto pewnie zrobić dygresję na temat kolorów oraz alfa-blendingu, który służy do rysowania miejsc półprzezroczystych. Jak niektórzy czytelnicy zapewne wiedzą, kolor w komputerze zazwyczaj opisuje się za pomocą trzech komponentów: RGB (od Red, Green, Blue – czerwony, zielony, niebieski). Wynika to nie tyle z natury światła, bo to stanowi pełne spektrum fal elektromagnetycznych w określonym zakresie częstotliwości, ile z budowy siatkówki ludzkiego oka, które ma trzy rodzaje receptorów reagujących na takie właśnie barwy. Gdybyśmy mieli oczy zbudowane jak owady czy ośmiornice, musielibyśmy zupełnie inaczej budować monitory, kamery, a także opisywać grafikę wewnątrz komputera. Jako ludzie natomiast tymi trzema komponentami możemy opisać dowolny widoczny dla nas kolor. Na przykład (0%, 0%, 0%) to brak jakiegokolwiek światła, czyli kolor czarny. (100%, 0%, 0%) to czysta barwa czerwona. (100%, 100%, 0%) to barwa czerwona dodana do zielonej, co daje kolor żółty. (100%, 100%, 100%) natomiast to wszystkie 3 komponenty ustawione na maksimum w równych proporcjach, co daje kolor biały.

W grafice komputerowej często do kanałów RGB dodaje się czwarty kanał oznaczony jako A (alfa), który oznacza przezroczystość. Ponieważ w informatyce lubimy potęgi dwójki, to daje okrągłe cztery komponenty na piksel. Jeżeli każdy komponent zapiszemy za pomocą jednego bajtu o wartościach 0..255, to da nam cztery bajty na piksel. Możemy więc obliczyć, ile pamięci zajmuje na przykład obraz na ekranie w rozdzielczości 4K: 3840 x 2160 x 4 bajty ≈ 31.64 MB.

Co oznacza ten czwarty kanał alfa? Pozwala on na mieszanie obrazu z tym, co zostało narysowane w danym miejscu wcześniej, a więc ma się znajdować wizualnie pod spodem. Wartość A = 0% oznacza pełną przezroczystość (ang. *transparency*), a więc nasz obiekt jest w tym miejscu niewidoczny. Kolor piksela nie ma wtedy znaczenia. A = 100% oznacza pełną nieprzezroczystość (ang. *opacity*), a więc całkowite zastąpienie starego koloru nowym. Wartości pomiędzy oznaczają półprzezroczystość, przy czym im wyższa alfa, tym większy wpływ na kolor końcowy ma kolor rysowanego aktualnie piksela. Operację tę, zwaną właśnie alfa-blendingiem, można opisać wzorem:

```
NowyKolor.rgb = NowyPiksel.rgb * NowyPiksel.a +
                StaryKolor.rgb * (100% - NowyPiksel.a)
```

Jest to tzw. operacja interpolacji liniowej (ang. *linear interpolation*, w skrócie „lerp"). Użyta tutaj notacja, z odwołaniem się do wybranych komponentów po kropce, jest charakterystyczna dla języków shaderów, jak HLSL czy GLSL. Języki te mają też wbudowaną funkcję, która realizuje tę operację (`lerp` w HLSL, `mix` w GLSL). (Osoby obeznane z tematem grafiki komputerowej mogłoby zapewne dodać, że lepsze od niej byłoby zastosowanie techniki tzw. *premultiplied alpha*, jednak to zagadnienie wykracza poza zakres tego artykułu.)

KLATKI OBRAZU

Między oknami aplikacji (w tym oknem konsoli z naszym programem) a grami występuje jedna duża różnica. Gry nieustannie odrysowują od podstaw cały swój obraz – od tła, przez postacie i przedmioty, aż po efekty specjalne. Podobnie jak podczas odtwarzania filmu, mówimy tu o „klatkach" (ang. *frame*). Aby gracz miał wrażenie płynnej animacji, gra powinna osiągać przynajmniej 30 klatek na sekundę (ang. *Frames Per Second* – FPS). Nikt nie zadaje sobie trudu, aby analizować, które miejsca na obrazie zmieniły się między klatkami. W końcu wystarczy obrócić nieznacznie pozycję kamery lub wykonać krok postacią, a na ekranie zmienia się praktycznie każdy piksel. Tymczasem aplikacje okienkowe są mniej dynamiczne i dzięki temu mogą odrysować tylko te miejsca (np. kontrolki GUI w oknie dialogowym, znaki w konsoli tekstowej) i tylko w takim momencie, w którym się zmieniły. Dzięki temu, dopóki nie uruchomimy jakiejś gry, karta graficzna jest dużo mniej obciążona pracą, co możemy poznać po cichszej pracy jej wentylatora.

Kiedy finalny obraz pulpitu ze wszystkim oknami we właściwej kolejności jest już skomponowany, pozostaje ostatni krok: przesłać go do wyświetlenia na monitorze. Monitor także pracuje, cyklicznie odświeżając całość obrazu wiele razy na sekundę. Częstotliwość odświeżania współczesnych monitorów to zazwyczaj 60 Hz, choć niektóre monitory tzw. „gamingowe" osiągają nawet ponad 200 Hz. Wysyłając kolej-

ne klatki obrazu do wyjścia DisplayPort czy HDMI, układ graficzny kopiuje te dane ze swojej pamięci operacyjnej. W przypadku układów graficznych zintegrowanych z procesorem rolę tę pełni zwykła pamięć operacyjna RAM. Dyskretne karty graficzne mają natomiast własną pamięć, nazywaną pamięcią video (VRAM). Jej rolą jest właśnie przechowywanie danych reprezentujących kolory pikseli do wyświetlenia na monitorze, jak również danych pomocniczych potrzebnych do przygotowania grafiki, np. siatki trójkątów i tekstury przedstawiające wszystkie postacie i obiekty występujące w grze bądź glify przedstawiające litery i inne znaki czcionki wyświetlane w naszej konsoli.

W ramach ciekawostki warto zdać sobie sprawę z faktu, że pamięć graficzna jest w stanie pomieścić w całości obraz wyświetlany na monitorze. Dziś, kiedy karty graficzne mają nawet 16 GB albo i więcej pamięci, trudno wyobrazić sobie, by mogło być inaczej, jednak nie zawsze tak było. W minionych dekadach istniały platformy, jak np. Atari 2600, w których pamięci nie wystarczało do tego celu. Ten konkretny komputer miał jej 128… bajtów. Mimo tego tworzone były różnorodne działające na nim gry. Procesor musiał wtedy w każdej klatce na bieżąco generować dane dla linii obrazu wyświetlanej w danej chwili. Możemy więc powiedzieć, że mimo całej tej złożoności współczesnego sprzętu i oprogramowania, w której od prostego „Hello World" do ujrzenia pikseli na ekranie wiedzie tak daleka droga, nasze życie, jako programistów, jest dziś dużo prostsze i wygodniejsze.

PODSUMOWANIE

W niniejszym artykule pokazaliśmy, że nawet uruchomienie banalnego programu „Hello World" w systemie Windows wiąże się z wykonaniem dużych pokładów znacznie bardziej skomplikowanego kodu. W naszej podróży zbadaliśmy wewnętrzne mechanizmy interpretera Pythona, by następnie prześledzić tok wykonania przechodzący przez kolejne procesy systemowe (*conhost.exe*, *fontdrvhost.exe*, *dwm.exe*) oraz sterowniki trybu jądra (*ntoskrnl.exe*, *win32k.sys*, *condrv.sys*, sterowniki karty graficznej). Można by dojść do wniosku, że rozbicie prostej operacji na tak wiele etapów jest marnotrawstwem zasobów, lecz z drugiej strony to właśnie taka architektura systemu umożliwia zastosowanie dobrych praktyk tworzenia kodu i izolację poszczególnych komponentów w celu zwiększania bezpieczeństwa. Z kolei dzięki dedykowanym sterownikom karty graficznej i dość skomplikowanemu potokowi graficznemu możemy cieszyć się przyjaznym dla oka interfejsem, który działa w każdej konfiguracji sprzętowej. Na szczęście moc obliczeniowa dzisiejszych komputerów jest na tyle duża, że pomimo wysokiego stopnia skomplikowania, cała opisana machineria i tak wykonuje się w mgnieniu oka.

Ostatnią myślą, jaką chcieliśmy przekazać, jest to, że każdą, nawet najbardziej skomplikowaną drogę można rozbić na mniejsze elementy, a następnie prześledzić, przeanalizować i ostatecznie – zrozumieć. *Stay curious!*

W sieci

[1] AST visualizer, github.com/pombredanne/python-ast-visualizer
[2] Python Bytecode Instructions, docs.python.org/3/library/dis.html?highlight=dis#python-bytecode-instructions
[3] inspect – Inspect live objects, docs.python.org/3/library/inspect.html
[4] Console Reference, docs.microsoft.com/en-us/windows/console/console-reference
[5] Introducing Windows Terminal, devblogs.microsoft.com/commandline/introducing-windows-terminal/
[6] Why aren't console windows themed on Windows XP?, devblogs.microsoft.com/oldnewthing/20071231-00/?p=23983
[7] Windows Command-Line: Inside the Windows Console, devblogs.microsoft.com/commandline/windows-command-line-inside-the-windows-console/
[8] Windows Command-Line: The Evolution of the Windows Command-Line, devblogs.microsoft.com/commandline/windows-command-line-the-evolution-of-the-windows-command-line/
[9] Windows Terminal, Console and Command-Line repo, github.com/microsoft/terminal
[10] Replace PolyTextOutW with ExtTextOutW, github.com/microsoft/terminal/commit/b7fc0f2
[11] The OpenType Font File, docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables
[12] Microsoft public symbol server, docs.microsoft.com/en-us/windows-hardware/drivers/debugger/microsoft-public-symbols
[13] Digitizing Letterform Designs, developer.apple.com/fonts/TrueType-Reference-Manual/RM01/Chap1.html#contours
[14] TrueType hinting, docs.microsoft.com/en-us/typography/truetype/hinting#what-is-hinting
[15] Understanding Graphs in Radeon GPU Profiler and GPUView, gpuopen.com/learn/understanding-graphs-in-radeon-gpu-profiler-and-gpuview/

ADAM SAWICKI

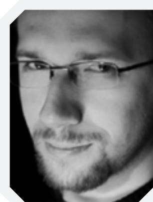
adam@asawicki.info

Zainteresowany głównie programowaniem grafiki, multimediów i gier. Aktualnie pracuje jako Developer Technology Engineer w AMD. Wcześniej m.in. w firmie Intel, Microsoft, w branży gier, w telewizji. Ma szerokie doświadczenie – od niskopoziomowego programowania sterowników graficznych i silników gier po fotografię, montaż video i VJ-ing na imprezach. Mgr inż. informatyki – absolwent Politechniki Częstochowskiej. Autor bloga asawicki.info.

MATEUSZ JURCZYK

j00ru.vx@gmail.com

Od wielu lat pasjonuje się niskopoziomowymi aspektami programowania i bezpieczeństwem komputerowym. Specjalizuje się w metodach znajdowania oraz wykorzystywania podatności w popularnych aplikacjach klienckich oraz systemach operacyjnych. Na co dzień pracuje w firmie Google w zespole Project Zero.



GYNVAEL COLDWIND

gynvael@coldwind.pl

Programista pasjonat z zamiłowaniem do bezpieczeństwa komputerowego i niskopoziomowych aspektów informatyki. Autor książki „Zrozumieć Programowanie", redaktor naczelny i twórca eksperymentalnego magazynu „Paged Out!", a także licznych artykułów, publikacji, podcastów oraz wystąpień poświęconych wspomnianym tematom. Od 2010 roku mieszka w Zurychu, gdzie pracuje dla firmy Google jako Senior Software Engineer/Information Security Engineer.