Linux

Odpowiedni kompilator to oczywiście **gcc**, oprócz tego trzeba mieć program **make**. Poszczególne dystrybucje zawierają pakiety ze skompilowanymi bibliotekami **FreeGLUT**, **GLFW** i **tiff**, które wystarczy zainstalować, można też ściągnąć pakiety źródłowe tych bibliotek i osobiście skompilować — w tym celu jest potrzebny program **cmake**.

Kompilacja aplikacji OpenGL-a opisanych w *nie takim krótkim kursie* polega na rozpakowaniu (programem `tar` albo `mc`) pliku archiwum `glsl-progII.tar.gz`, wejściu do katalogu `glsl-progII` i wydaniu polecenia `make`. Można też wejść do podkatalogu `extra` i tam wydać to polecenie. Polecenie `make clean` powoduje skasowanie plików z rozszerzeniem `.o`, a `make mrproper` czyści podłogę, tj. usuwa także programy wykonywalne.

Powyższe polecenia działają też w podkatalogach poszczególnych aplikacji, które można kompilować indywidualnie. Pliki programów wykonywalnych są zapisywane w katalogu ze źródłami, w tym samym miejscu znajdują się pliki źródłowe szaderów w GLSL-u, które muszą być obecne w katalogu, w którym jest plik wykonywalny aplikacji, jeśli ma ona działać.

Windows

W roku Pańskim 1990 standard OpenGL jeszcze nie istniał, a Windows[1] nie był systemem operacyjnym, tylko graficznym dodatkiem do systemu DOS. Po moich ówczesnych doświadczeniach od systemów operacyjnych ze słowem Windows w nazwie, jako użytkownik i tym bardziej jako programista, trzymałem się z dala aż do roku 2021, w którym przyjąłem do wiadomości powód, aby poświęcić trochę czasu (obecnemu) systemowi Windows. Tym powodem są ci studenci, którzy nie używają Linuksa lub z systemem Windows wiążą swoje plany zawodowe; im też chciałbym mieć coś do zaoferowania. Dlatego doprowadziłem do działania w systemie Windows 11 dwa z trzech programów opisanych w książce *OpenGL i GLSL* (*nie taki krótki kurs*) (zrobienie natywnej dla Windows wersji trzeciej aplikacji jest na ukończeniu) i opracowałem szkielet natywnej aplikacji tego systemu, realizującej grafikę w standardzie OpenGL. Poniżej opisuję (zanim zapomnę) sposób, w jaki zdołałem tego dokonać.

Prace w systemie Windows wykonałem przy użyciu programu **Visual Studio Community 2022**, dostępnego bezpłatnie. Instalując go (z sieci), należy zadbać o obecność (opcjonalnego) pakietu przeznaczonego do uruchamiania aplikacji graficznych systemu Windows w języku C++ (w Visual Studio nie ma osobnego kompilatora C, ale kompilator C++ radzi sobie także z kodem napisanym w C).

Windows SNAFU

- Nie ma dowiązań symbolicznych plików, więc pliki źródłowe, takie jak `utilities.c` mają kopie w katalogach ze źródłami wszystkich aplikacji. W rezultacie pliki, które w Li-

[1] po polsku „Łokna”

nuksie zajmują ok. 6 MB, a w archiwum `glsl-progII.tar.gz` połowę tego, w Windows zajmują na dysku odpowiednio więcej miejsca, a plik `glsl-progIIW.zip` ma ponad 30 MB.

To jest jeszcze nic w porównaniu z setkami megabajtów dopisywanych do katalogów przez Visual Studio — wystarczy tym programem otworzyć katalog, nawet bez zamiaru zrobienia czegokolwiek w nim (wersja skompilowana zajmuje ponad 2 GB, a po spakowaniu ok. 600 MB).

- Terminal tekstowy otwierany przez system z chwilą uruchomienia aplikacji z procedurą `main` w razie zatrzymania aplikacji natychmiast się zamyka, nie dając czasu na przeczytanie np. tekstów diagnostycznych. Podczas pracy nad aplikacją najlepiej jest wywoływać ją z linii komend w terminalu otwartym wcześniej. Przynajmniej on się nie zamknie.
- Procedury realizujące API Windows mają nazwy zaczynające się od wielkich liter — ale to wszystko, co o tych nazwach da się powiedzieć. Nie ma czegoś takiego jak w bibliotece Xlib, gdzie wszystkie procedury mają nazwy na literę `X`, albo w OpenGL-u, gdzie nazwy procedur mają prefiks `gl`. W rezultacie nie wiadomo, czy jakaś nazwa, której chciałoby się użyć dla własnego podprogramu, nie jest już zajęta. To bardzo łatwo może spowodować bałagan w kodzie.
- Kompilatory firmy Microsoft dla procesorów Intel 8088 i 8086 traktowały identyfikatory `near` i `far` jak słowa kluczowe, (służyły one do określenia, czy deklarowana zmienna wskaźnikowa ma zajmować 16, czy 32 bity). To do tej pory pokutuje, a ja użyłem tych identyfikatorów w aplikacjach. Pokuta jest zadana w ten sposób, że w pewnych plikach nagłówkowych (systemu? bibliotek? — nieważne) te identyfikatory są czynione nazwami pustych makrodefinicji, przez co są one po prostu usuwane z ciągu symboli przetwarzanego przez parser kompilatora. Aby nie trzeba było przerabiać kodów źródłowych, wystarczy usunąć te makrodefinicje, pisząc polecenia

```
#ifdef near
#undef near
#endif
#ifdef far
#undef far
#endif
```

Ja ten tekst umieściłem w plikach nagłówkowych takich jak `app1.h`, w których dotąd były tylko makrodefinicje potrzebnych aplikacji numerów specyfikacji OpenGL-a — uznałem, że w tym miejscu to najmniej zaśmieci kod.

Kompilowanie bibliotek w Visual Studio

Biblioteki **FreeGLUT**, **GLFW** i **tiff** najlepiej jest skompilować ze źródeł wziętych z sieci, dzięki czemu otrzymamy wersje dostosowane do posiadanego sprzętu. Pakiety źródłowe są zao-

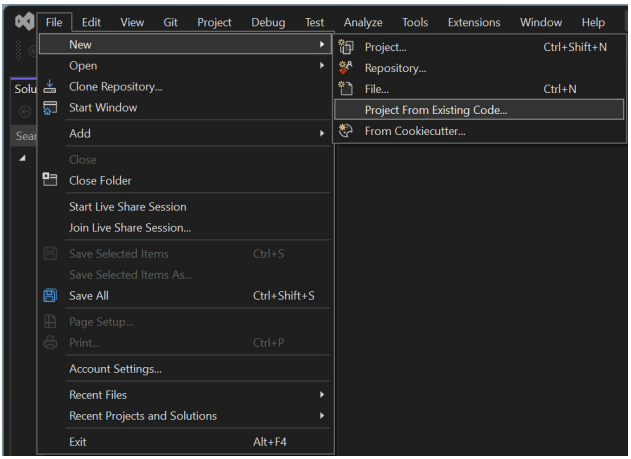
patrzone w pliki `CMakeLists.txt`, które Visual Studio, po wskazaniu w otwartym katalogu, czyta i wykonuje działania skutkujące powstaniem skompilowanych bibliotek.

Pakiety ze źródłami **bibiotek FreeGLUT, GLFW3** i **tiff** ściągnąłem odpowiednio ze stron

`https://sourceforge.net/projects/freeglut/`,
`https://sourceforge.net/projects/glfw/`,
`https://download.osgeo.org/libtiff/`.

Pliki `freeglut-3.2.1.tar.gz`, `glfw-3.3.5.tar.bz2` i `tiff-4.3.0.zip` to archiwa zapisane w różnych formatach, przy czym Windows nie wszystkie umie rozpakować bez użycia komercyjnych programów, które potrafią to zrobić. Najprościej mi było rozpakować pierwsze dwa pliki w Linuksie, ponownie spakować programem **zip** i przenieść na dysk z Windows.

Po rozpakowaniu wystarczy wejść do katalogu każdej z tych bibliotek i w menu Visual Studio najechać kursorem na plik `CMakeLists.txt`, aby pakiet został zbudowany. Domyślnie powstaje wersja `Debug`. Jeśli bibliotek chcemy tylko używać (tj. dołączać do aplikacji), to trzeba wybrać odpowiednią opcję konfiguracji cmake (np. `Release`), a potem skompilować. Biblioteki w wersji `Debug` mają do nazwy doczepioną literę `d`, np. `freeglutd.lib`. W menu VS jest też polecenie instalowania zbudowanych bibliotek. Polega ono na przepisaniu do nowego podkatalogu w projekcie. Stamtąd można skopiować biblioteki „ręcznie" do katalogów docelowych, tj. `.lib` do `??????????`, `.dll` do `??????`, a pliki nagłówkowe do `??????????`, ale do tego trzeba mieć uprawnienia administratora.

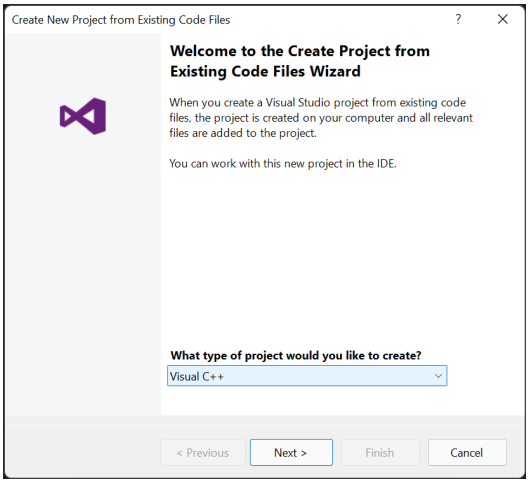


Rysunek 1.

Kompilowanie aplikacji w Visual Studio

Na przykładzie aplikacji `template-wgl`, za pomocą serii obrazków, pokazuję, co trzeba zrobić. Po wejściu do katalogu ze źródłami aplikacji, która jeszcze nie była kompilowana, trzeba utworzyć projekt. Pstrykamy `Project From Existing Code` (rys. 1).

Otwiera się dialog do określenia, jaki to ma być projekt. Wybieramy `Visual C++` (rys. 2).

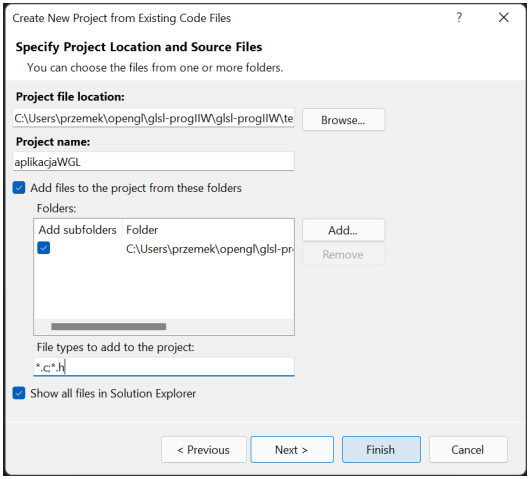


Rysunek 2.

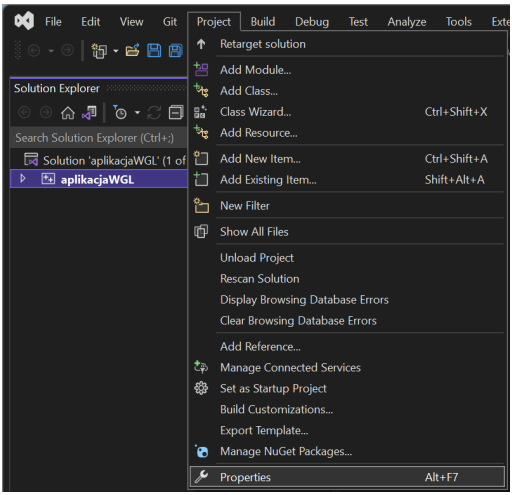
Po pstryknięciu guzika `Browse` w oknie, które się otworzy po pstryknięciu `Next`, wybieramy (bieżący) katalog. Nazwa projektu to będzie nazwa pliku wykonywalnego, który powstanie (np. `aplikacjaWGL`, z rozszerzeniem `.exe`, którego nie piszemy). Typy, tj. rozszerzenia nazw plików źródłowych podajemy `*.c` i `*.h` i potem można pstryknąć guzik `Finish`, bo domyślne odpowiedzi na dalsze pytania są wystarczające (rys. 3).

Do projektu trzeba dołączyć właściwe biblioteki. Rysunek 4 pokazuje początek drogi do odpowiedniego dialogu (trzeba pstryknąć napis `Properties`). **Uwaga:** na liście plików musi być wskazana nazwa projektu, bo inaczej nie da się tej drogi znaleźć.

W oknie dialogowym trzeba rozwinąć menu `Linker` i pstryknąć `Input`. Po pstryknięciu napisu `Additional Dependencies` na końcu linii pojawi się prostokącik, którego pstryknięcie otwiera kolejny dialog — pstrykamy napis `Edit`. Otwiera się okno, w którym trzeba wpisać nazwy bibliotek, których VS samo nie dołączy (rys. 5). Zawsze jest potrzebna biblioteka `opengl32.lib`. Dla aplikacji FreeGLUTa (wszystkie wersje pierwszej aplikacji) trzeba podać jeszcze `freeglut.lib`, a dla aplikacji GLFW (wszystkie wersje drugiej aplikacji) `glfw3.lib`. Jeśli aplikacja używa czasomierza (w jej skład wchodzi plik `timer.c`), to trzeba też podać bibliotekę `winmm.lib` (jeśli dołączenie tej biblioteki nie odnosi skutku, to trzeba w pliku `timer.c` usunąć lub wykomentować definicję symbolu `USE_HIFRCLOCK`). Aplikacje 2D–2K nakładają na obiekt tekstury przeczytane z plików w formacie TIFF, zatem



Rysunek 3.

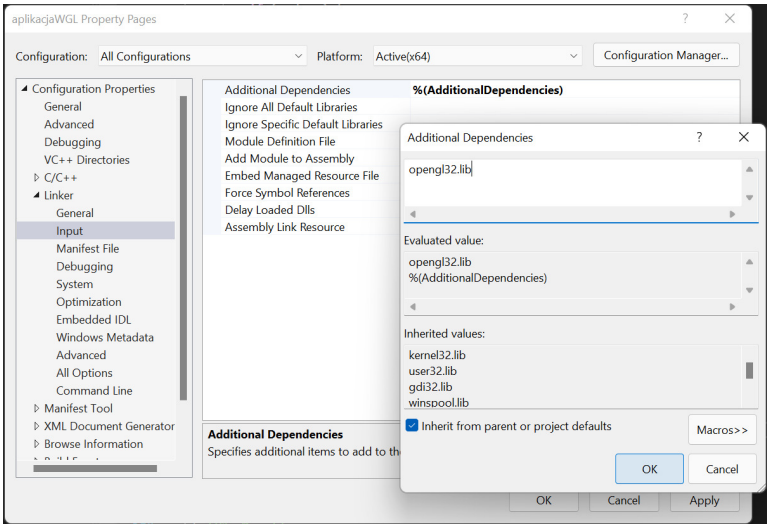


Rysunek 4.

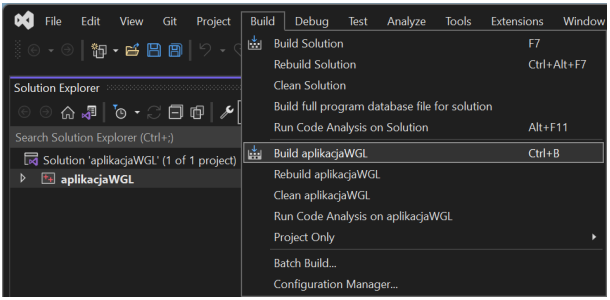
trzeba do nich dołączyć jeszcze bibliotekę `tiff.lib` (albo `tiffd.lib`). Nazwy bibliotek piszemy, rozdzielając je średnikami. Potem pstrykamy dwa guziki `OK`.

Aby skompilować projekt, pstrykamy odpowiedni napis w menu `Build` (rys. 6).

Aplikacje (z wyjątkiem szkieletów w podkatalogach `template-glut`, `template-glfw` oraz `template-wgl`) po wywołaniu czytają pliki źródłowe szaderów z bieżącego katalogu. Jeśli ich nie znajdą, to nie będą działać. Dlatego albo szadery (pliki z rozszerzeniem `.glsl`)



Rysunek 5.



Rysunek 6.

trzeba skopiować do podkatalogu `x64\Build\Debug`, albo plik wykonywalny przenieść do podkatalogu z plikami źródłowymi. Aplikacje nakładające tekstury potrzebują też obecności w katalogu, z którego są uruchamiane, plików TIFF z obrazami użytymi jako tekstury.