

Sebastian Tabaka

KURS OPENGL

Krok po kroku



Wersja z dnia: 03.04.2016

Spis treści

1.	Wstęp	3
2.	Konfiguracja systemu oraz IDE	4
2.1.	Ubuntu.....	4
2.2.	Windows.....	7
3.	Inicjalizacja pierwszego okna	11
4.	Odczyt informacji o sprzęcie	15
5.	Pierwszy trójkąt	17
6.	Wariacje na temat rysowania	22
7.	Podstawy shaderów	24
8.	Potok przetwarzania	28
9.	Obiekty VAO.....	33
10.	Antialiasing i biblioteka GLFW	36
11.	Tryb pełnoekranowy i zmiana rozmiaru okna.....	37
12.	Maszyna stanu.....	39
13.	Korzystanie z obiektów VBO	41
14.	Weryfikacja shaderów	44
15.	Zmienne uniform	47
16.	Winding order oraz face culling	50
17.	Macierze transformacji	52
18.	Transformacje.....	58
19.	Teksturowanie	62
20.	Błędy odczytu tekstur	69
21.	Licznik FPS.....	71
22.	Kamera wirtualna.....	73
23.	Klawiatura i mysz.....	80
24.	Filtrowanie tekstur	86
25.	Oświetlenie	90
26.	Wczytywanie modeli z pliku	104
27.	Mgła.....	114
28.	Mullti-teksturowanie	116
29.	Współczynnik odbicia światła na podstawie tekstur	118
30.	Odrzucanie fragmentów	122
31.	Kierunkowe źródło światła - reflektor	126

32.	Mapowanie normalnych.....	130
33.	Przechwytywanie obrazu	141
34.	Skybox.....	143
35.	Prosty generator terenu	151
36.	Korekcja gamma	163
37.	Odbicie oraz refrakcja	169
38.	Renderowanie tekstu na podstawie bitmap.....	176
39.	Renderowanie tekstu z FreeType	188
40.	Interfejsy użytkownika - GUI.....	199
40.1.	Pasek postępu	199
41.	???	209

1. Wstęp

Jakiś czas temu hobbystycznie zacząłem programować w OpenGL. Wciągnęło mnie to bardzo. Nie ma nic przyjemniejszego niż oglądanie własnoręcznie stworzonego trójwymiarowego świata, po którym możemy dowolnie się poruszać. Poza tym, programowanie grafiki trójwymiarowej jest chyba najbardziej widowiskową dziedziną programowania. No i kto nie chciał tworzyć kiedyś własnych gier komputerowych?

Lubię przekazywać wiedzę i uczyć innych. Szczególnie wtedy, gdy przekazuje innym swoją pasję. Dlatego właśnie powstała ta książka. Będę prowadził w niej kurs OpenGL od podstaw. Krok po kroku będziemy wchodzić coraz głębiej w świat trójwymiarowy poznając kolejne elementy biblioteki OpenGL.

Chciałbym zaznaczyć, że **nie jestem jakimś OpenGL guru**. Wręcz przeciwnie. Dopiero niedawno zacząłem poznawać tą bibliotekę. Na ile mi czas pozwala, staram się rozwijać i uczyć się nowych rzeczy. Jeśli znajdziesz jakiś błąd w tym co opisuję, to możesz śmiało do mnie pisać, chętnie podyskutuję. Postaram się także tematy opisywać jak najprzystępniej tak, aby każdy mógł je zrozumieć bez konieczności zaglądania do zewnętrznych źródeł.

Nie jestem jednak w stanie opisywać wszystkiego od zupełnych podstaw. Zakładam, że posiadasz już w pewnym stopniu znajomość języka C++, kompilowania, budowania projektu oraz dołączania zewnętrznych bibliotek. Podczas opisywania chcę się skupiać tylko na OpenGL, a nie na opisywaniu programowania w C++. Na pewno musisz znać podstawy C++, także mieć chociaż trochę pojęcia o programowaniu obiektowym. Nie będę stosował zaawansowanych technik programistycznych, ale dobrze jest wiedzieć, co się dzieje.

Przydałaby się także już pewna wiedza matematyczna. Przede wszystkim, czym są wektory i macierze oraz znać operacje na nich. Jeśli jednak tego nie wiesz, to spokojnie. Postaram się to w miarę możliwości wytłumaczyć, kiedy już będzie to potrzebne.

Ważnym elementem (jak nie najważniejszym) jest posiadana karta graficzna. **Musi ona wspierać OpenGL co najmniej w wersji 3.0.** Najprawdopodobniej będziesz mógł to sprawdzić na stronie producenta w dokumentacji technicznej. Nie będę opisywał starego standardu OpenGL opartego na gotowych funkcjach. Skupimy się na programowalnym potoku, czyli na shaderach.

To tyle tytułem wstępu. Startujemy!

2. Konfiguracja systemu oraz IDE

Zaczynamy od rzeczy najbardziej oczywistej, czyli od przygotowania. Nie będziemy mogli pracować przedtem nie instalując odpowiednich bibliotek oraz nie konfigurując środowiska programistycznego.

Niestety, tutaj pojawia się pierwszy problem, a zarazem ich mnóstwo. Napisanie uniwersalnego poradnika konfiguracji nie jest możliwe. Każdy użytkownik ma swoje preferencje co do wyboru systemu operacyjnego, środowiska programistycznego oraz wielu innych czynników. Ponadto, każdy komputer jest indywidualnym przypadkiem. Na niektórych komputerach mogą pojawić się błędy czy problemy, które nie występują na innych komputerach. Mogę tylko mniej więcej nakreślić poprawny kierunek działania. W celu rozwiązania tych indywidualnych problemów odsyłam do internetu. Wystarczy trochę poszukać, a na pewno coś ciekawego się znajdzie na interesujący nas temat.

Z racji, że najpopularniejszymi systemami operacyjnymi są **Windows** oraz **Ubuntu**, to dla nich przedstawię konfigurację. Co do środowiska programistycznego, będzie to **Qt Creator** dla obydwu tych systemów.

2.1. Ubuntu

Zakładam, że masz już poprawnie zainstalowane sterowniki karty graficznej. Instrukcja ta jest dla **Ubuntu 14.04**, ale powinna także być prawidłowa dla innych wersji tego systemu.

1. Uruchamiamy terminal i instalujemy niezbędne pakiety:

```
>> sudo apt-get install mesa-common-dev  
>> sudo apt-get install mesa-utils  
>> sudo apt-get install freeglut3-dev
```

2. Wpisujemy w terminalu następujące polecenia w celu sprawdzenia, czy wszystko działa prawidłowo:

```
>> glxinfo | grep "OpenGL version"
```

Powinniśmy otrzymać informację zwrotną o wersji OpenGL:

```
OpenGL version string: 3.3.0 NVIDIA 331.113
```

Sprawdzamy jeszcze, czy karta graficzna została wykryta poprawnie:

```
>> glxinfo | grep render  
direct rendering: Yes  
OpenGL renderer string: GeForce GTX 285/PCIe/SSE2
```

Jeśli karta graficzna nie wspiera OpenGL w wersji co najmniej **3.0**, to niestety czas najwyższy na zmianę sprzętu.

3. Zakładamy sobie gdzieś na komputerze katalog o nazwie, np. **opengl_libs**, w którym będziemy umieszczać niezbędne biblioteki (**każda w osobnym katalogu**).

4. Wchodzimy na stronę <http://glew.sourceforge.net/>, ściągamy najnowszy kod źródłowy biblioteki **GLEW**, rozpakowujemy go (do osobnego katalogu) i umieszczamy w katalogu z punktu 3, budujemy oraz instalujemy:

```
>> make  
>> sudo make install
```

5. Pobieramy źródła biblioteki **GLFW** (<http://www.glfw.org/download.html>), rozpakowujemy do katalogu utworzonego w punkcie 3 oraz budujemy ją według instrukcji zawartej na stronie: <http://www.glfw.org/docs/latest/compile.html>. W większości przypadków będzie to po prostu wydanie następujących poleceń w katalogu tej biblioteki:

```
>> mkdir build  
>> cmake ..  
>> make  
>> sudo make install
```

6. Instalujemy biblioteki **boost**:

```
>> sudo apt-get install libboost-all-dev
```

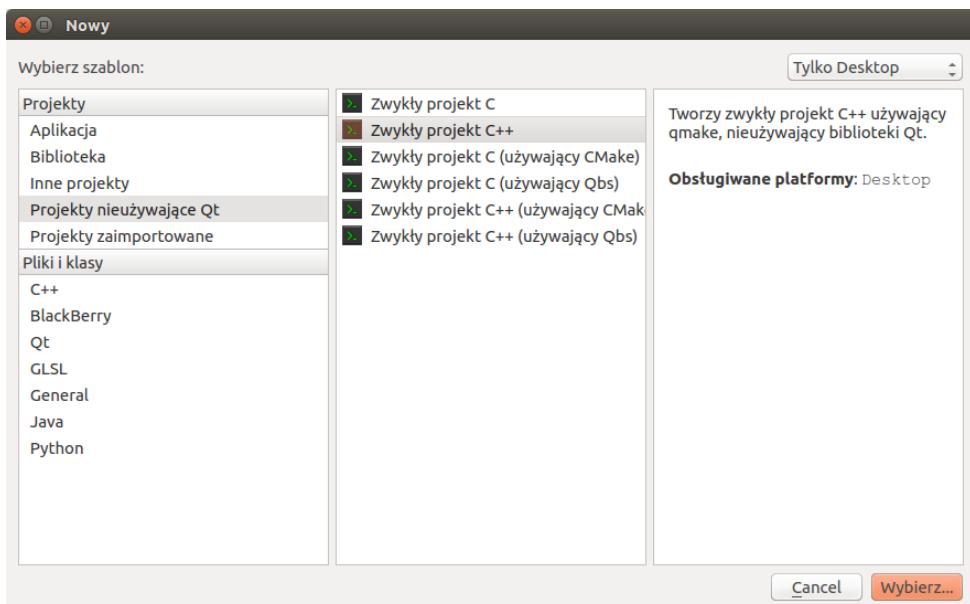
7. Pobieramy kod źródłowy biblioteki **assimp** ze strony http://assimp.sourceforge.net/main_downloads.html, rozpakowujemy do katalogu z punktu 3, budujemy oraz instalujemy (polecenia wydajemy oczywiście jak wyżej w katalogu biblioteki):

```
>> mkdir build  
>> cmake ..  
>> make  
>> sudo make install
```

8. Pobieramy bibliotekę matematyczną **GLM** ze strony <http://sourceforge.net/projects/ogl-math/> oraz rozpakowujemy ją do naszego katalogu. Nie wymaga ona budowania oraz instalacji. Oparta jest tylko na plikach nagłówkowych.
9. Pobieramy źródła biblioteki **FreeImage** ze strony <http://freeimage.sourceforge.net/download.html>. Tak samo jak we wszystkich przypadkach wyżej rozpakowujemy ją do osobnego folderu i wrzucamy do katalogu z punktu 3. Otwieramy w terminalu katalog biblioteki i budujemy ją za pomocą następujących poleceń:

```
>> make  
>> sudo make install
```

10. Instalujemy darmową wersję **Qt Creator** ze strony <http://www.qt.io/download-open-source/>.
11. Po zainstalowaniu już wszystkiego czas na stworzenie nowego projektu. W tym celu uruchamiamy Qt Creator, wchodzimy w menu **Plik**, a następnie **Nowy plik lub projekt**. Z listy **Projekty** wybieramy **Projekty nieużywające Qt**, a potem wybieramy **Zwykły projekt C++** (rys. 2.1). Klikamy przycisk **Wybierz**.



Rys. 2.1. Tworzenie nowego projektu w Qt Creator.

12. W kolejnym oknie wpisujemy nazwę projektu oraz jego docelową lokalizację. Naciskamy **Dalej**. Zestawu narzędzi z reguły nie musimy edytować, a więc naciskamy **Dalej**. Na końcu naciskamy przycisk **Zakończ**. Nasz projekt jest już gotowy.
13. Z listy plików projektu znajdującej się z lewej strony otwieramy plik z rozszerzeniem ***.pro**. Jest to plik konfiguracyjny projektu. W nim musimy dodać wymagane biblioteki oraz ustawić ścieżki dostępu do plików nagłówkowych. Dodajemy zatem następujący fragment:

```
INCLUDEPATH += $$/home/user_name/opengl_libs/glm/glm
INCLUDEPATH += $$/usr/lib64

DEPENDPATH += $$/usr/lib64

LIBS += -L$$/usr/lib64/ -lGlew
LIBS += -L$$/usr/lib64/ -lglfw3
LIBS += -L$$/usr/lib64/ -lfreeimage
LIBS += -L$$/usr/lib64/ -lassimp

LIBS += -L$$/usr/lib64/ -lGLU
LIBS += -L$$/usr/lib64/ -lGL
LIBS += -L$$/usr/lib64/ -lX11
LIBS += -L$$/usr/lib64/ -lXxf86vm
LIBS += -L$$/usr/lib64/ -lXrandr
LIBS += -L$$/usr/lib64/ -lpthread
LIBS += -L$$/usr/lib64/ -lXi
```

Ostatecznie plik **.pro** będzie wyglądał następująco:

```
TEMPLATE = app

CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

INCLUDEPATH += $$/home/user_name/Libraries/glm/glm
INCLUDEPATH += $$/usr/lib64

DEPENDPATH += $$/usr/lib64

LIBS += -L$$/usr/lib64/ -lGlew
```

```
LIBS += -L$$/usr/lib64/ -lglfw3
LIBS += -L$$/usr/lib64/ -lfreetype
LIBS += -L$$/usr/lib64/ -lassimp

LIBS += -L$$/usr/lib64/ -lGLU
LIBS += -L$$/usr/lib64/ -lGL
LIBS += -L$$/usr/lib64/ -lX11
LIBS += -L$$/usr/lib64/ -lXxf86vm
LIBS += -L$$/usr/lib64/ -lXrandr
LIBS += -L$$/usr/lib64/ -lpthread
LIBS += -L$$/usr/lib64/ -lXi

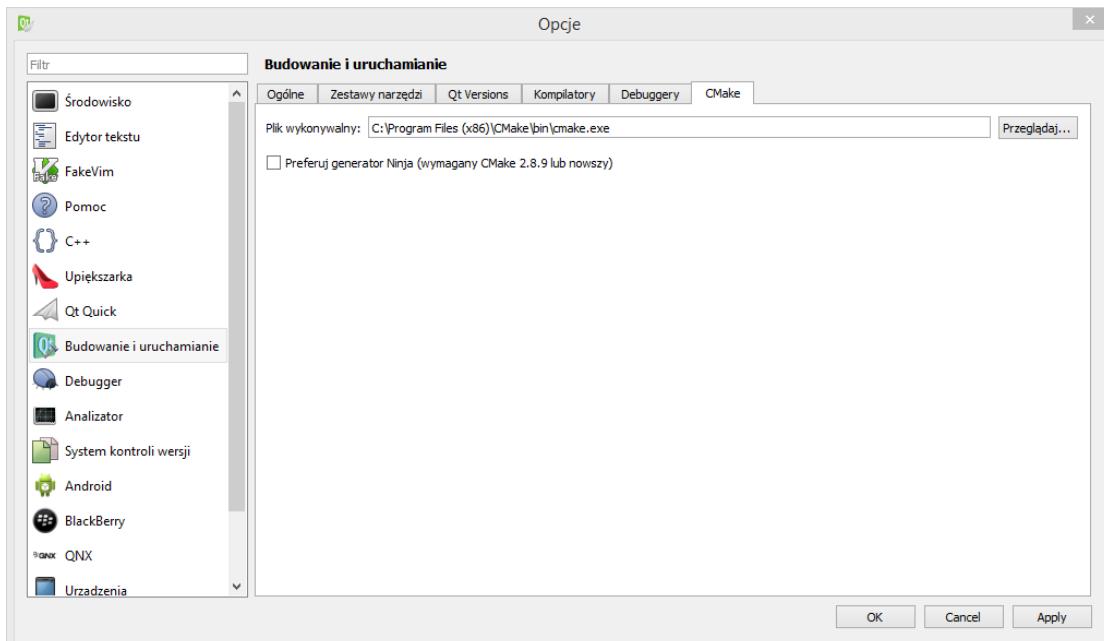
SOURCES += main.cpp
```

14. Otwieramy menu **Budowanie** i wybieramy opcję **Uruchom qmake**.

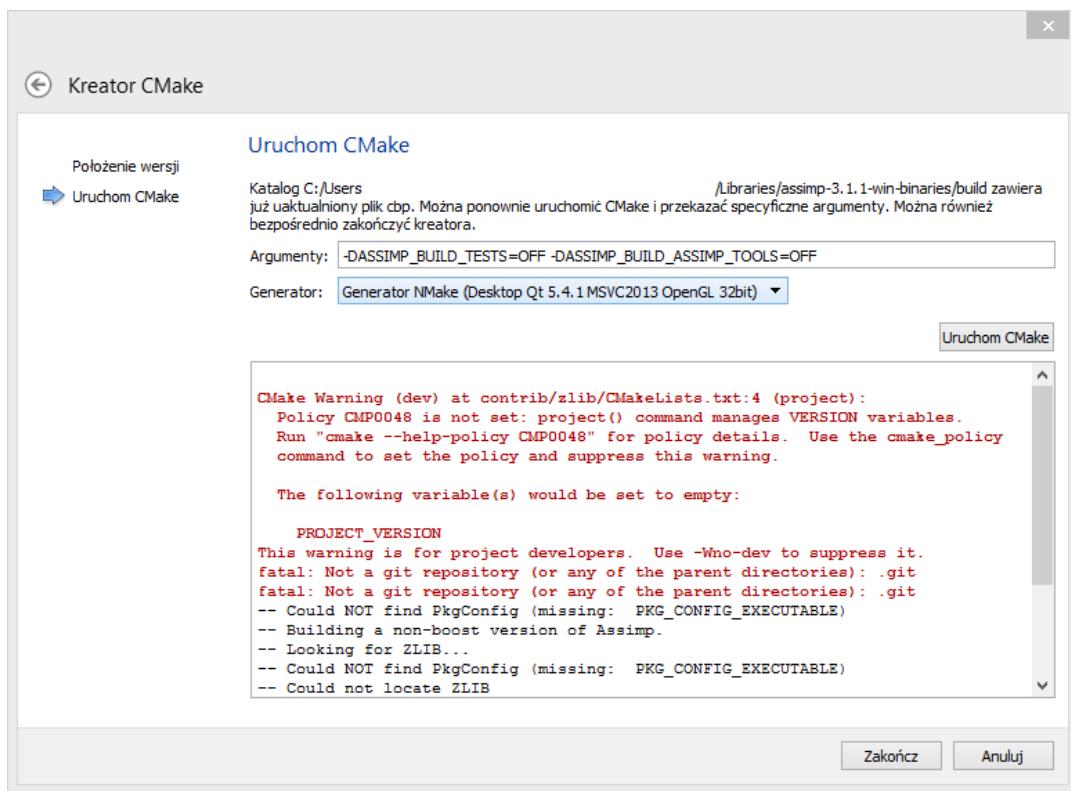
15. Nasz projekt jest już przygotowany.

2.2. Windows

1. Zakładamy gdzieś na komputerze katalog o nazwie, np. `opengl_libs`, w którym będziemy umieszczać niezbędne biblioteki (**każda w osobnym katalogu**).
2. Wchodzimy na stronę <http://glew.sourceforge.net/>, pobieramy najnowszą zbudowaną (**Windows binaries**) wersję biblioteki i rozpakowujemy ją.
3. Pobieramy **32-bitową** wersję biblioteki **GLFW** ze strony <http://www.glfw.org/download.html> i rozpakowujemy ją.
4. Pobieramy bibliotekę matematyczną **GLM** ze strony <http://sourceforge.net/projects/ogl-math/> oraz rozpakowujemy ją do naszego katalogu. Nie wymaga ona budowania oraz instalacji. Oparta jest tylko na plikach nagłówkowych.
5. Wchodzimy na stronę <http://freeimage.sourceforge.net/download.html> i pobieramy wersję biblioteki **FreeImage** dla Windows (Win32/Win64). Rozpakowujemy ją do naszego katalogu.
6. Pobieramy kod źródłowy biblioteki **assimp** ze strony http://assimp.sourceforge.net/main_downloads.html i rozpakowujemy ją do katalogu.
7. Instalujemy darmową wersję **Qt Creator** ze strony <http://www.qt.io/download-open-source/>. Proponuję wybrać opcję **Qt Offline Installers**, a następnie wersję **32-bitową VS2013**.
8. Pobieramy i instalujemy **CMake** ze strony <http://www.cmake.org/>.
9. Teraz należy przystąpić do zbudowania biblioteki **assimp**. W tym celu uruchamiamy **Qt Creator**, wchodzimy w menu **Narzędzia > Opcje > Budowanie** oraz **uruchamianie > CMake**. Klikamy **Przeglądaj** i wybieramy lokalizację pliku **cmake.exe** z katalogu instalacji **CMake** (rys. 2.2).
10. Wchodzimy w **Plik > Otwórz plik lub projekt** i wybieramy plik **CMakeLists.txt** z katalogu głównego biblioteki **assimp**.
11. W okienku **Położenie wersji** wybieramy katalog, gdzie ma zostać zbudowana biblioteka. Proponuję stworzyć katalog o nazwie **build** wewnątrz katalogu biblioteki i tutaj go ustawić. Naciskamy **Dalej**.
12. W nowym oknie w pole **Argumenty** wpisujemy: **-DASSIMP_BUILD_TESTS=OFF -DASSIMP_BUILD_ASSIMP_TOOLS=OFF**, a potem naciskamy **Uruchom CMake** (rys. 2.3). Czekamy za zakończenie procesu i naciskamy **Zakończ**.



Rys. 2.2. Konfiguracja CMake w Qt Creator.



Rys. 2.3. Uruchomienie CMake.

13. Wchodzimy w menu **Budowanie**, a następnie wybieramy **Zbuduj wszystko**.
14. Czekamy na zakończenie budowania. Po skończonym budowaniu wchodzimy w menu **Plik > Zamknij wszystkie projekty i edytory**.
15. Wchodzimy w menu **Plik**, a następnie **Nowy plik lub projekt**. Z listy **Projekty** wybieramy **Projekty nieużywające Qt**, a potem wybieramy **Zwykły projekt C++** (rys. 2.1). Klikamy przycisk **Wybierz**.

16. W kolejnym oknie wpisujemy nazwę projektu oraz jego docelową lokalizację. Naciskamy **Dalej**. Zestawu narzędzi z reguły nie musimy edytować, a więc naciskamy **Dalej**. Na końcu naciskamy przycisk **Zakończ**. Nasz projekt jest już gotowy.
17. Z listy plików projektu znajdującej się z lewej strony otwieramy plik z rozszerzeniem ***.pro**. Jest to plik konfiguracyjny projektu. W nim musimy dodać wymagane biblioteki oraz ustawić ścieżki dostępu do plików nagłówkowych. Dodajemy zatem następujący fragment:

```
LIBS_PATH = C:/Users/user/Documents/OpenGL/Libraries

LIBS += -lopengl32
LIBS += -luser32
LIBS += -lgdi32
LIBS += -lcomdlg32
LIBS += -lshell32

INCLUDEPATH += $$${LIBS_PATH}/glm/glm
INCLUDEPATH += $$${LIBS_PATH}/glew-1.12.0/include
INCLUDEPATH += $$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/include
INCLUDEPATH += $$${LIBS_PATH}/FreeImage/Dist/x32
INCLUDEPATH += $$${LIBS_PATH}/assimp-3.1.1-win-binaries/include

DEPENDPATH += $$${LIBS_PATH}/glew-1.12.0/include
LIBS += -L$$${LIBS_PATH}/glew-1.12.0/lib/Release/Win32 -lglew32

DEPENDPATH += $$${LIBS_PATH}/FreeImage/Dist/x32
LIBS += -L$$${LIBS_PATH}/FreeImage/Dist/x32 -lFreeImage

DEPENDPATH += $$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/include
LIBS += -L$$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/lib-vc2013 -lglfw3

DEPENDPATH += $$${LIBS_PATH}/assimp-3.1.1-win-binaries/include
LIBS += -L$$${LIBS_PATH}/assimp-3.1.1-win-binaries/build/code -lassimpd
```

W zmiennej **LIBS_PATH** ustawiamy ścieżkę dostępu do katalogu z bibliotekami. Ostatecznie plik ***.pro** będzie wyglądał następująco:

```
TEMPLATE = app

CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt
CONFIG += c++11

LIBS_PATH = C:/Users/user/Documents/OpenGL/Libraries

LIBS += -lopengl32
LIBS += -luser32
LIBS += -lgdi32
LIBS += -lcomdlg32
LIBS += -lshell32

INCLUDEPATH += $$${LIBS_PATH}/glm/glm
INCLUDEPATH += $$${LIBS_PATH}/glew-1.12.0/include
INCLUDEPATH += $$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/include
INCLUDEPATH += $$${LIBS_PATH}/FreeImage/Dist/x32
INCLUDEPATH += $$${LIBS_PATH}/assimp-3.1.1-win-binaries/include

DEPENDPATH += $$${LIBS_PATH}/glew-1.12.0/include
LIBS += -L$$${LIBS_PATH}/glew-1.12.0/lib/Release/Win32 -lglew32

DEPENDPATH += $$${LIBS_PATH}/FreeImage/Dist/x32
LIBS += -L$$${LIBS_PATH}/FreeImage/Dist/x32 -lFreeImage

DEPENDPATH += $$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/include
LIBS += -L$$${LIBS_PATH}/glfw-3.1.1.bin.WIN32/lib-vc2013 -lglfw3

DEPENDPATH += $$${LIBS_PATH}/assimp-3.1.1-win-binaries/include
```

```
LIBS += -L${LIBS_PATH}/assimp-3.1.1-win-binaries/build/code -lassimpd  
SOURCES += \  
main.cpp
```

18. Otwieramy menu **Budowanie** i wybieramy opcję **Uruchom qmake**.
19. Nasz projekt jest już przygotowany.
20. Pamiętajmy jeszcze o dodaniu kilku DLL do naszego programu: assimpd.dll, FreeImage.dll, glew32.dll.

3. Inicjalizacja pierwszego okna

Pisząc standardową aplikację z reguły mamy do wyboru kilka możliwości podczas tworzenia okna tej aplikacji. Najczęściej jednak korzystamy z zewnętrznych bibliotek. Do stworzenia okna wykorzystującego OpenGL posłuży nam biblioteka **GLFW**. Jest to multi-platformowa darmowa biblioteka o otwartych źródłach. Pozwala w łatwy sposób stworzyć okno obsługujące OpenGL. Umożliwia także łatwe zarządzanie urządzeniami wejściowymi takimi jak klawiatura, mysz i joystick.

Z tej racji, że GLFW jest multi-platformowe, nie jest konieczne dodawanie do naszego programu dodatkowych plików nagłówkowych specyficznych dla danego systemu operacyjnego, np. `Windows.h` dla Windows. Biblioteka ta zawiera w sobie już niezbędne elementy do utworzenia okna i jego obsługi na danym systemie operacyjnym. Jeśli natomiast chcemy skorzystać z specyficznych elementów jakiegoś API, np. funkcji `MessageBox()`, musimy dodać plik nagłówkowy `Windows.h` do naszego kodu źródłowego. **Ale uwaga, należy dodać go przed dołączeniem pliku nagłówkowego biblioteki GLFW.** Czyli w taki sposób:

```
#include <Windows.h>
#include <GLFW/glfw3.h>
```

Biblioteka GLFW dołącza także odpowiednie pliki nagłówkowe OpenGL, a zatem o to także nie musimy się już martwić. Wystarczy jedna dyrektywa `include` i wszystko załatwione.

W celu bardziej obrazowego dalszego wytłumaczenia na początku przedstawię bardzo prosty szablon, który można zastosować do późniejszych programów.

```
#include <GLFW/glfw3.h>
#include <iostream>

static void error_callback(int error, const char* description)
{
    std::cerr << "Error: " << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}

int main(void)
{
    glfwSetErrorCallback(error_callback);

    if(!glfwInit())
        return 1;

    GLFWwindow* window = glfwCreateWindow(640, 480, "My GLFW Window", NULL, NULL);
    if(!window)
    {
        glfwTerminate();
        return 1;
    }

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);

    while(!glfwWindowShouldClose(window))
    {
```

```
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}
```

Pierwszą czynnością, którą należy wykonać jest przeprowadzenie inicjalizacji biblioteki GLFW. Dokonuje się tego za pomocą funkcji `glfwInit()`. Jeśli inicjalizacja przebiegnie z sukcesem, funkcja zwraca `GL_TRUE`. W przeciwnym wypadku zwraca `GL_FALSE` oraz wywołuje automatycznie funkcję `glfwTerminate()` zwalniającą zajęte zasoby.

Przed inicjalizacją umieszczone jest jeszcze wywołanie funkcji `glfwSetErrorCallback()`. Jest to jedna z kilku funkcji GLFW, które mogą zostać wywołane jeszcze przed inicjalizacją. Ustawienie funkcji błędu umożliwia wyświetlenie na ekranie ewentualnych komunikatów błędów, które pojawią się począwszy od inicjalizacji, aż do późniejszych fragmentów programu. Przyjmuje ona wskaźnik do funkcji zwracającej typ `void` oraz przyjmującej dwa parametry: pierwszy typu `int`, a drugi typu `const char*`. Jeśli zachcemy "wyciszyć" funkcję błędu wystarczy wywołać funkcję `glfwSetErrorCallback()` z parametrem `NULL`.

Gdy już biblioteka GLFW zostanie poprawnie zainicjalizowana można uruchomić funkcję tworzącą okno. Służy do tego funkcja `glfwCreateWindow()`, która zwraca uchwyt do utworzonego okna. Parametrami tej funkcji są kolejno:

- szerokość okna,
- wysokość okna,
- tytuł,
- monitor, na którym ma być uruchomiony tryb pełnoekranowy; jeśli parametr ten ustawiony jest na `NULL` uruchomiony zostanie tryb okienkowy,
- okno, z którym mają być dzielone zasoby; na razie nie jest to wykorzystywane, można ustawić po prostu na `NULL`.

W przypadku, gdy nie uda się stworzyć okna funkcja zwróci wartość `NULL`.

Jeśli chcielibyśmy uruchomić program na pełnym ekranie należały skorzystać z funkcji `glfwGetPrimaryMonitor()`, która zwraca wskaźnik do monitora głównego. A zatem instrukcja tworząca okno wyglądałaby wtedy tak:

```
GLFWwindow* window = glfwCreateWindow(640, 480, "My GLFW Window", glfwGetPrimaryMonitor(),
NULL);
```

Jeśli z różnych powodów nie uda się utworzyć okna, wywoływana jest funkcja `glfwTerminate()`. Niszczy ona wszystkie utworzone okna, zwalnia używane zasoby oraz ustawia bibliotekę GLFW w stan niezainicjalizowany.

Gdy już mamy gotowe okno należy teraz wskazać OpenGL, gdzie ma rysować. Służy do tego funkcja `glfwMakeContextCurrent()`. Podawany jest do niej uchwyt do naszego okna.

Ustawiany jest wtedy kontekst aktualnego okna jako aktywny. Mówiąc w uproszczeniu, kontekstem jest pewna struktura określająca jak i gdzie należy rysować.

Działanie całego programu odbywa się w pętli. To w tej pętli będzie zachodziło renderowanie naszej grafiki oraz obsługa całej logiki programu. Z każdym krokiem pętli sprawdzany jest warunek, czy okno powinno zostać zamknięte. Służy do tego funkcja `glfwWindowShouldClose()`. Sprawdza ona, jaki jest stan flagi danego okna informujący o konieczności zamknięcia okna. Flaga taka jest na przykład ustawiana wtedy, gdy naciśnięty zostanie krzyżyk na górnej belce okna albo w przypadku naciśnięcia kombinacji klawiszy ALT+F4. Zmianę stanu flagi zamknięcia można także wywołać sztucznie. Służy do tego funkcja `glfwSetWindowShouldClose()`, która przyjmuje wskaźnik do danego okna oraz nowy stan flagi zamknięcia. Funkcję tą wykorzystujemy w funkcji obsługi klawiszy, która została zdefiniowana jeszcze przed funkcją `main()`. Aby ustawić własną funkcję obsługi klawiszy należy przesłać jej wskaźnik do funkcji `glfwSetKeyCallback()`. Jej wywołanie widzimy przed pętlą `while`. Oczekuje ona na dwa parametry: wskaźnik na dane okno oraz wskaźnik na funkcję obsługi klawiszy, która będzie związana z tym oknem. W naszej funkcji obsługi klawiszy umożliwiliśmy zamknięcie okna po naciśnięciu klawisza ESC.

W naszym przypadku ciało pętli `while` jest na razie ubogie. Znajdują się w nim tylko dwie funkcje. Pierwsza z nich, tzn. `glfwSwapBuffers()` służy do zamiany buforów renderowania. GLFW obsługuje dwa bufory. Jeśli jeden z nich jest wyświetlany aktualnie na ekranie, to na drugim odbywa się renderowanie. Po skończeniu renderowania ten drugi bufor jest wyświetlany na ekranie, a renderowanie odbywa się na pierwszym buforze. Można powiedzieć, że te bufory to takie jakby dwie klatki filmu. Gdy jedna z nich jest wyświetlana, druga jest przygotowywana do wyświetlenia.

Jak wiemy współczesne systemy operacyjne oparte są na zdarzeniach. Naciśnięcie przycisku, przesunięcie, zminimalizowanie okna, itp. to zdarzenia. Aby takie zdarzenia obsłużyć potrzebujemy funkcji, która w każdym obiegu pętli sprawdzi, czy są jakieś zdarzenia w kolejce do obsłużenia. Służy do tego druga użyta przez nas funkcja `glfwPollEvents()`. Funkcja ta obsługuje tylko te zdarzenia, które zostały już umieszczone w kolejce i nie czeka na nowe. Zwraca natychmiast po ich obsłużeniu. Jeśli nie umieścilibyśmy tej funkcji w pętli nie można byłoby między innymi zamknąć okna programu. Trzeba byłoby wtedy zabić proces poprzez menadżer zadań bądź wiersz poleceń. **Uwaga:** funkcję tą należy umieszczać zawsze po, a nie przed funkcją `glfwSwapBuffers()`.

Po opuszczeniu pętli `while`, np. poprzez naciśnięcie przycisku ESC, wykonywane są dwie funkcje czyszczące. Funkcja `glfwDestroyWindow()` niszczy dane okno oraz jego kontekst. Żadne nowe przychodzące zdarzenia nie będą już obsługiwane. Funkcję `glfwTerminate()` omówiłem już nieco wyżej. Powinniśmy ją zawsze umieszczać na końcu programu, aby użyte zasoby zostały poprawnie zwolnione.

Jeśli wszystko poszło dobrze, to na ekranie ukaże się nam czarne okienko.

Tips & Tricks:

- Na początkach nauki OpenGL nie stosuj żadnych zaawansowanych technik programowania, pisz wszystko w jednym pliku. Zamiast skupić się na tym co jest

naprawdę ważne, będziesz myślał jak zorganizować kod albo jak go zoptymalizować.

- Nauka OpenGL nie jest łatwa, nie poddawaj się!

Kod źródłowy z tego rozdziału jest w katalogu: 1_Iinicjalizacja_okna

4. Odczyt informacji o sprzęcie

Gdy już wiemy jak prawidłowo tworzyć okno możemy teraz przystąpić do dalszego rozwoju naszego programu. Na początku dodajmy jeszcze jedną bibliotekę - **GLEW**. Tak samo jak **GLFW**, jest to biblioteka multi-platformowa. Dostarcza ona przede wszystkim mechanizm pozwalający stwierdzić, które elementy (rozszerzenia) OpenGL są dostępne na aktualnej platformie sprzętowej. Ponadto umożliwia używanie najnowszej wersji OpenGL.

Inicjalizację GLEW należy przeprowadzić **dopiero po ustawieniu aktywnego kontekstu**. Pamiętamy, że służyła do tego funkcja `glfwMakeContextCurrent()`. Do zainicjalizowania biblioteki GLEW służy funkcja `glewInit()`. W przypadku powodzenia zwraca ona wartość `GLEW_OK`. Należy jeszcze zwrócić uwagę na to, aby dyrektywa `include` biblioteki GLEW była przed dyrektywą `include` biblioteki GLFW. **Jest to bardzo ważne**.

Mogemy wykorzystać bibliotekę OpenGL do odczytania kilku interesujących nas parametrów, np. karty graficznej użytej do renderowania. Do tego celu posłużymy się funkcją `glGetString()`. Jako parametr może ona przyjąć następujące wartości:

- `GL_VENDOR` - nazwa producenta odpowiedzialnego za używaną implementację OpenGL,
- `GL_RENDERER` - nazwa używanej karty graficznej do renderowania,
- `GL_VERSION` - numer wersji używanego OpenGL,
- `GL_SHADING_LANGUAGE_VERSION` - numer wersji używanego GLSL (OpenGL Shading Language).

Użytkownicy technologii, np. Nvidia Optimus, mogą wybrać sobie podczas uruchamiania programu, z którą kartą graficzną go uruchomić. Wyświetlone informacje będą się wtedy różnić pomiędzy dwoma kartami graficznymi.

A na zakończenie tego krótkiego rozdziału podaję jeszcze kod źródłowy.

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

static void error_callback(int error, const char* description)
{
    std::cerr << "Error: " << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if(key == GLFW KEY_ESCAPE && action == GLFW PRESS)
        glfwSetWindowShouldClose(window, GL TRUE);
}

int main(void)
{
    glfwSetErrorCallback(error callback);

    if(!glfwInit())
        return 1;

    GLFWwindow* window = glfwCreateWindow(640, 480, "My GLFW Window", NULL, NULL);
    if(!window)
```

```
{  
    glfwTerminate();  
    return 1;  
}  
  
glfwMakeContextCurrent(window);  
  
GLenum error_code = glewInit();  
if(error_code != GLEW_OK)  
{  
    std::cerr << "GLEW init error: " << glewGetErrorString(error_code);  
}  
  
glfwSetKeyCallback(window, key_callback);  
  
const GLubyte* vendor = glGetString(GL_VENDOR);  
const GLubyte* renderer = glGetString(GL_RENDERER);  
const GLubyte* version = glGetString(GL_VERSION);  
const GLubyte* shading = glGetString(GL_SHADING_LANGUAGE_VERSION);  
  
std::cout << vendor << std::endl << renderer << std::endl << version << std::endl <<  
shading << std::endl;  
  
while(!glfwWindowShouldClose(window))  
{  
    glfwSwapBuffers(window);  
    glfwPollEvents();  
}  
  
glfwDestroyWindow(window);  
glfwTerminate();  
  
return 0;  
}
```

Tips & Tricks:

- Aby włączyć obsługę nowszych wersji OpenGL należy ustawić zmienną `glewExperimental` na wartość `GL_TRUE`. Należy to zrobić jeszcze przed inicjalizacją biblioteki GLEW.

```
glewExperimental = GL_TRUE;  
glewInit();
```

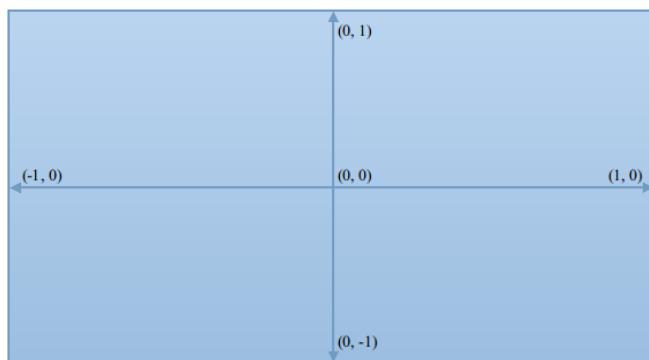
Kod źródłowy z tego rozdziału jest w katalogu: [2_GLEW_Odczyt_informacji](#)

5. Pierwszy trójkąt

Nadszedł czas na coś ciekawszego, narysujemy trójkąt na ekranie. Jak to mówią: nie od razu Kraków zbudowano, trzeba od czegoś zacząć.

Trójkąt jak wiemy posiada trzy wierzchołki. Każdy z wierzchołków możemy opisać za pomocą trzech współrzędnych X, Y oraz Z. Potrzebujemy też miejsca, gdzie możemy zapisać te wierzchołki, aby potem z nich skorzystać. Stworzymy obiekt **VBO (Vertex Buffer Object)**. Pozwala on przechowywać dane różnego rodzaju w szybkiej pamięci karty graficznej. Umieszczenie danych w pamięci karty graficznej znacznie zwiększa prędkość dostępu do tych danych, co dalej związane jest ze wzrostem wydajności samego renderowania grafiki.

Na początku musimy jednak zdefiniować tablicę, w której opiszemy położenie wierzchołków na ekranie. Poniżej przedstawiony rysunek (rys. 5.1) pokazuje ułożenie układu współrzędnych na ekranie. Jak widać punkt (0, 0) znajduje się w punkcie centralnym ekranu. Maksymalną wartością każdej osi jest 1, a minimalną wartością jest -1. I tak na przykład, lewy górny róg ekranu ma współrzędne (-1, 1), a prawy górny ma współrzędne (1, 1).



Rys. 5.1. Układ współrzędnych ekranu.

Nasz trójkąt możemy zdefiniować na przykład następująco:

```
GLfloat triangle points[] = {
    -0.8f, -0.8f, 0.0f,
    0.0f, 0.8f, 0.0f,
    0.8f, -0.8f, 0.0f
};
```

Wierzchołki należy podawać zawsze zgodnie z ruchem wskazówek zegara bądź przeciwnie. Albo tak albo tak. Należy sobie wybrać i przyjąć jakąś konwencję. Umożliwi to później stwierdzić, która strona figury jest wyświetiana: przednia czy tylna. Oczywiście współrzędne podajemy w kolejności: X, Y, Z, X, Y, Z, itd.

Często w kodzie będą pojawiały się zmienne typu: `GLfloat`, `GLuint`, itp. Są to po prostu typy stworzone poprzez `typedef` na potrzeby OpenGL. Chyba wszystkie obecnie dostępne IDE pozwalają "podejrzeć", co się kryje za daną definicją typu stworzonego za pomocą `typedef`. Ogólnie koncepcja jest taka: zmiennych tych używamy tylko w odniesieniu do

funkcji oraz elementów OpenGL. Ma to jakby oddzielić zwykłe zmienne C++ od tych definiowanych przez OpenGL oraz zwiększyć czytelność kodu. Od razu wtedy widać, która zmienna jest zmienną używaną w OpenGL, a która jest zwykłą zmienną używaną do innych celów.

Przejdźmy dalej. Stworzymy teraz wcześniej wspomniany już obiekt VBO. Na początku pokażę kod, a potem wytłumaczę, o co w tym wszystkim chodzi.

```
GLuint vbo;  
glGenBuffers(1, &vbo);  
 glBindBuffer(GL_ARRAY_BUFFER, vbo);  
 glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_points), triangle_points, GL_STATIC_DRAW);
```

Funkcja `glGenBuffers()` zapisuje do zmiennej `vbo` listę liczb całkowitych zbudowaną z indeksów wolnych (nieużywanych) buforów. W tym przypadku jako pierwszy parametr tej funkcji podaliśmy wartość 1, a zatem życzymy sobie, aby otrzymać tylko jedną wartość. Tą wartością jest właśnie nazwa (indeks) nieużywanego bufora. Jest to wartość całkowita dodatnia. Z tego względu, że chcemy tylko otrzymać indeks jednego bufora to zdefiniowaliśmy obiekt `vbo` jako po prostu `GLuint`. Jeśli natomiast chcielibyśmy otrzymać więcej wartości (np. dwa wolne bufory), wtedy obiekt `vbo` musiałby być tablicą dwu-elementową. Nie mamy dodatkowo gwarancji, że otrzymane indeksy będą kolejnymi wartościami, np. 4 i 5. Gdy już mamy zapisany numer (albo numery) wolnego bufora możemy go z czymś powiązać. Służy do tego funkcja `glBindBuffer()`. Przyjmuje ona dwa parametry. Pierwszym jest cel, czyli, z czym należy ten bufor powiązać. W naszym przypadku jest to `GL_ARRAY_BUFFER`. Oznacza to, że dany bufor będzie tablicą danych. Drugim parametrem, jak łatwo się domyślić, jest indeks danego bufora, który otrzymaliśmy w poprzednim kroku.

Ostatnim krokiem jest wpisanie danych do przygotowanego wcześniej bufora. W tym celu wykorzystuje się funkcję `glBufferData()`. Jako jej parametry podajemy kolejno cel (taki sam jak wyżej), rozmiar danych, dane oraz przewidywane zastosowanie tych danych. W naszym przypadku jest to `GL_STATIC_DRAW`. Oznacza to, że dane nie będą modyfikowane oraz będą używane często do rysowania.

Stworzyliśmy już bufor do przechowywania położenia wierzchołków figury (modelu). Każdy model może posiadać także inne cechy, np. współrzędne tekstury. W celu opisania tych innych cech należy także stworzyć kolejny bufor, w którym będą zapisane na przykład współrzędne tekstury. Żeby potem się w tym wszystkim nie pogubić, np. który bufor współrzędnych tekstury powinien być powiązany z danym buforem położenia wierzchołków, możemy stworzyć wspólny obiekt wiążący kilka buforów w jeden. Daje nam to duży komfort, gdyż potem nie ma już konieczności zastanawiać się, które bufory należy powiązać ze sobą. W jednym obiekcie mamy zdefiniowany wtedy cały model. W tym celu definiuje się obiekt **VAO (Vertex Attribute Object)**. Tak samo jak powyżej, najpierw pokażę kod, a potem go omówię.

```
GLuint vao;  
 glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);  
 glBindBuffer(GL_ARRAY_BUFFER, vbo);  
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);  
 glEnableVertexAttribArray(0);
```

Funkcja `glGenVertexArrays()` działa tak samo jak przedstawiona wyżej funkcja `glGenBuffers()`. Zwraca indeks (nazwę) wygenerowanej tablicy danych opisujących wierzchołek. Indeks ten jest także typu `unsigned int`. Tablica taka, jak już wspomniałem wyżej, może składać się z kilku buforów, np. bufor wierzchołków, wektorów normalnych, współrzędnych tekstury, ale my na razie wrzucimy tylko współrzędne wierzchołków. Następnie wywoływana jest funkcja `glBindVertexArray()`, która uaktywnia wygenerowaną przed chwilą tablicę. Od tej pory wszystkie operacje będą odbywały się właśnie na tej tablicy. Funkcję `glBindBuffer()` także już znamy. Nie napisałem tylko o niej jednego. Powoduje ona także, że przekazany do niej obiekt (w tym przypadku `vbo`) staje się aktywny. Jeśli mielibyśmy kilka buforów VBO, np. jeden z wierzchołkami, drugi z wektorami normalnymi, to wywołanie tej funkcji powoduje, że aktywujemy tylko jeden z nich, jakby wyciągamy go na wierzch. To wszystko stanie się bardziej jasne, gdy w następnym rozdziale pokażę jak wyświetlić na ekranie dwa trójkąty. Funkcja `glVertexAttribPointer()` definiuje wskaźnik pokazujący na pewne cechy figury (modelu), np. współrzędne wierzchołków, wektory normalne, itp. Z racji, że wyżej mieliśmy wywołanie funkcji `glBindBuffer()` dla obiektu `vbo`, w którym zapisane są współrzędne wierzchołków, to stworzymy wskaźnik pokazujący na współrzędne wierzchołków. Parametrami tej funkcji są po kolei: indeks, pod którym będziemy się odwoływać, liczba elementów przypadająca na cechę (u nas są to 3, gdyż mamy trzy współrzędne: X, Y i Z), typ danych, normalizacja. Dwa ostatnie parametry są na razie nie istotne i je zerujemy. Na zakończenie odblokowujemy możliwość korzystania z tego indeksu wskazywanego przez wskaźnik. Dokonuje się tego za pomocą funkcji `glEnableVertexAttribArray()`, a jako jej parametr podaje się numer indeksu użyty w funkcji powyżej. Schematycznie cały ten proces można przedstawić tak jak na rys. 5.2 (w celu lepszego zobrazowania wprowadziłem dwa bufory VBO).

Najtrudniejszy element jest za nami. Przystąpmy teraz do narysowania naszego trójkąta na ekranie. Rysowanie odbywa się w pętli. Każda iteracja pętli rysuje jedną klatkę obrazu. Pętla wykonuje się do momentu, gdy zażądamy zamknięcia okna, np. poprzez naciśnięcie krzyżyka na górnej belce okna lub naciśnięcie klawisza `Escape`. Sprawdzanie warunku zamknięcia okna odbywa się za pomocą funkcji `glfwWindowShouldClose()`.

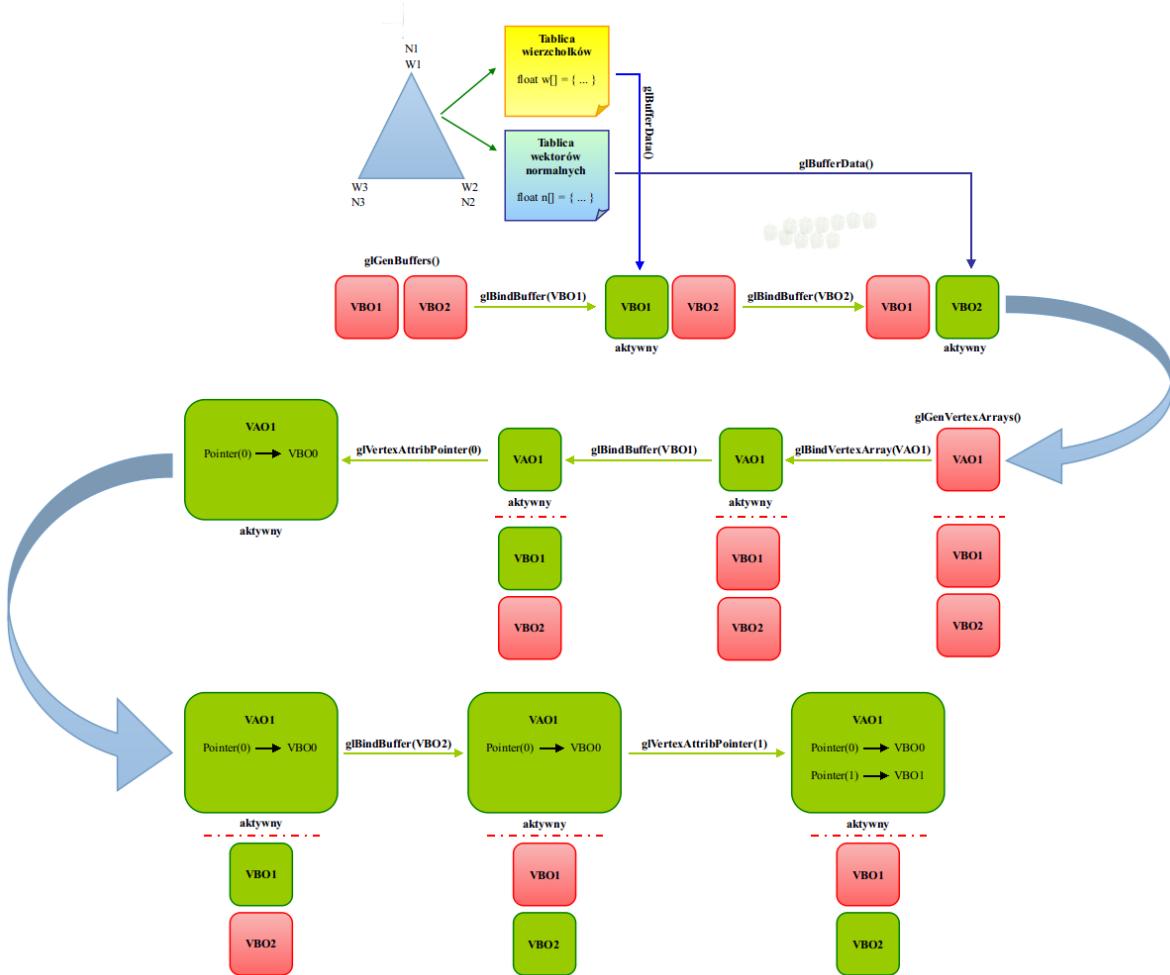
W pętli na początku wywołujemy funkcję `glClear()`, która czyści podany bufor. W naszym przypadku żądamy wyczyszczenia dwóch buforów: bufora głębokości oraz bufora do rysowania kolorowego. Funkcja `glClear()` czyści bufor, bądź raczej wypełnia go wartością ustaloną przez inne funkcje, np. funkcję `glClearColor()`, która ustawia kolor RGBA bufora do rysowania kolorowego.

Dalej musimy wybrać, który obiekt VAO chcemy narysować. Używamy do tego znanej już nam funkcji `glBindVertexArray()`, która uaktywnia dany obiekt VAO. Ostatnią czynnością jest wywołanie funkcji rysującej `glDrawArrays()`. Jako jej pierwszy parametr podajemy jakiego typu prymitywów chcemy używać do renderowania. Prymitywem jest podstawa figura geometryczna, z których buduje się inne bardziej skomplikowane. W tym przypadku używamy trójkątów, ale możemy też porobić eksperymenty z innymi prymitywami, np. `GL_POINTS` lub `GL_LINES`. Drugim parametrem funkcji `glDrawArrays()` jest indeks początkowy w tablicy wierzchołków (od którego

wierzchołka należy rysować), natomiast w trzecim parametrze podajemy ile wierzchołków chcemy wyrenderować. Całość prezentuje się następująco:

```
while(!glfwWindowShouldClose(window))
{
    glClearColor(0.5f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

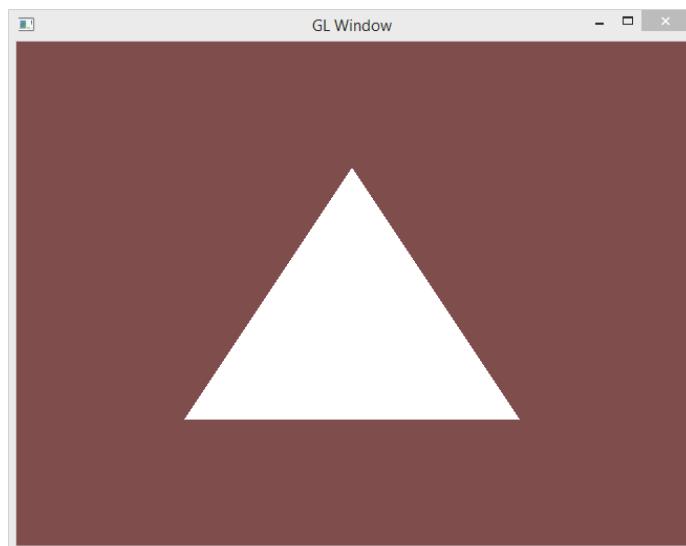


Rys. 5.2. Schematyczny proces zarządzania obiektami VBO i VAO.

W wyniku otrzymamy okno przedstawione na rys. 5.3.

Tips & Tricks:

- Na początku nie powinno ustawiać się tła okna na kolor czarny. Najlepiej jest ustawić jakiś jasny kolor. Może nasz program działa dobrze, ale nic nie widzimy na ekranie z powodu czarnego tła.
- Jeśli masz problemy z określeniem współrzędnych wierzchołków, to najpierw narysuj sobie wszystko na kartce.



Rys. 5.3. Wynik działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: **3_Pierwszy_trojkat**

6. Wariacje na temat rysowania

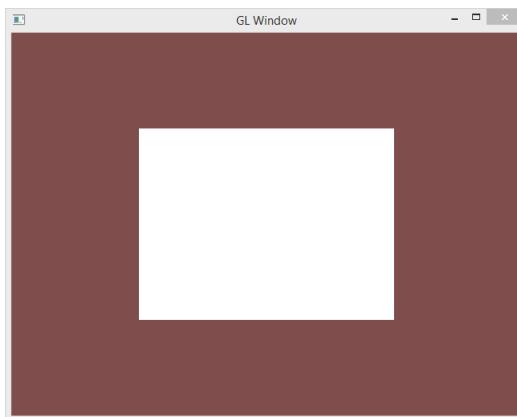
Już wiemy jak narysować prosty trójkąt. Teraz pobawimy się trochę rysowaniem. Na początku narysujemy kwadrat (prostokąt). Będzie składał się on z dwóch trójkątów o jednej wspólnej krawędzi. Zdefiniujmy więc wierzchołki.

```
GLfloat points[] = {  
    -0.5f, -0.5f, 0.0f,  
    -0.5f, 0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    -0.5f, 0.5f, 0.0f,  
    0.5f, 0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f  
};
```

Widzimy, że zdefiniowałem tutaj 6 wierzchołków (po 3 na każdy trójkąt). Pierwsza trójka zbioru współrzędnych należy do trójkąta pierwszego, a druga do drugiego trójkąta. Kolejną zmianą, którą należy wprowadzić jest zmiana liczby rysowanych wierzchołków na 6. Dokonujemy tego w tej linii:

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

W wyniku zobaczymy na ekranie okno takie jak na rys. 6.1.



Rys. 6.1. Wynik działania programu.

W poprzednim rozdziale wspomniałem o prymitywach, których używamy do rysowania. Teraz trochę rozszerzę ten temat.

Tablica, w której zapisujemy wierzchołki jest jakby strumieniem. Musimy zatem określić jak OpenGL ma interpretować ten strumień. OpenGL może interpretować strumień wierzchołków jako punkty, linie lub trójkąty.

1. Punkty

Każdy indywidualny wierzchołek interpretowany jest jako punkt. Możemy ustawić rozmiar punktu za pomocą funkcji `glPointSize()`, której podajemy żądany rozmiar punktu.

```
glPointSize(10.0);  
// ...
```

```
glDrawArrays(GL_POINTS, 0, 6);
```

2. Linie

W przypadku linii istnieje większa liczba możliwości, gdyż są trzy rodzaje prymitywów złożonych z linii. Są to:

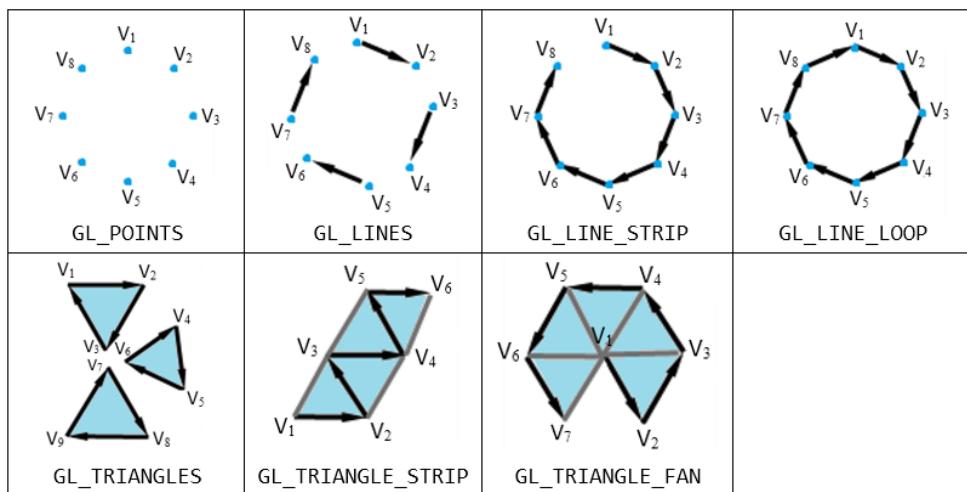
- **GL_LINES** - następujące po sobie pary wierzchołków interpretowane są jako punkt początkowy oraz końcowy danej linii. Czyli linia tworzona jest przez wierzchołki (1, 2), (3, 4), itd. Jeśli podamy nieparzystą liczbę wierzchołków, to ten ostatni dodatkowy wierzchołek zostanie zignorowany.
- **GL_LINE_STRIP** - sąsiednie kolejne wierzchołki tworzą kolejny punkt linii łamanej. Jeśli podamy n wierzchołków, otrzymamy $n - 1$ linii.
- **GL_LINE_LOOP** - podobnie jak wyżej, z tą różnicą, że pierwszy i ostatni wierzchołek również utworzą linię. Stąd wynika nazwa tego prymitywu - pętla. A więc z n wierzchołków otrzymamy n linii.

3. Trójkąty

W tym przypadku także istnieją trzy rodzaje prymitywów.

- **GL_TRIANGLES** - każda rozdzielna trójka wierzchołków tworzy trójkąt. A zatem trójkąt tworzą wierzchołki (1, 2, 3) i (4, 5, 6), itd.
- **GL_TRIANGLE_STRIP** - każda sąsiednia grupa trzech wierzchołków tworzy trójkąt. Czyli z n wierzchołków otrzymamy $n - 2$ trójkątów.
- **GL_TRIANGLE_FAN** - pierwszy podany wierzchołek jest wierzchołkiem wspólnym dla wszystkich kolejnych trójkątów. A zatem w wyniku otrzymujemy trójkąty o wierzchołkach: (1, 2, 3), (1, 3, 4), (1, 4, 5), itd. Z n wierzchołków otrzymamy $n - 2$ trójkątów.

Bardziej obrazowo prezentuje to poniższy obrazek (rys. 6.2).



Rys. 6.2. Rodzaje prymitywów.

7. Podstawy shaderów

W celu osiągnięcia bardziej widowiskowych efektów rysowania stosuje się shadery. Są to takie programy, których zadaniem jest wykonanie zaawansowanych obliczeń właściwości wierzchołków oraz pikseli. Dzięki nim możliwe jest zatem bardziej skomplikowane obliczenie oświetlenia oraz materiału na danym obiekcie.

W najprostszym przypadku będziemy potrzebowali dwóch shaderów:

- **vertex shader** - uruchamiany jest dla każdego zdefiniowanego wierzchołka. Zajmuje się on w uproszczeniu transformacją położenia wierzchołka na ekranie.
- **fragment shader** - jego zadaniem jest obliczenie koloru pikseli, czyli między innymi obliczeniem oświetlenia sceny.

W OpenGL shadery programowane są w języku **GLSL** (OpenGL Shading Language). Jest to język przypominający swoją składnią język C. Shadery kompilowane są dopiero wtedy, gdy chcemy ich użyć. Aby napisać swój własny shader mamy dwie możliwości. Napisać go jako łańcuch znaków, np. w obiekcie `std::string`. Te rozwiązanie jest oczywiście niewygodne oraz niezalecane, no ale jest taka możliwość. Zalecanym rozwiązaniem jest po prostu wczytywanie shadera z pliku. Shadery piszemy w plikach tekstowych głównie (ale niekoniecznie, aby był to jakiś format tekstowy) z rozszerzeniem `.glsl`. Napiszmy więc na początku funkcję, która wczyta nam shadery, abyśmy potem mogli je już wykorzystać.

```
GLint loadShaders(std::string vertex_shader, std::string fragment_shader)
{
    GLuint vertex_shader_id = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragment_shader_id = glCreateShader(GL_FRAGMENT_SHADER);

    std::string vertex_shader_data;
    std::ifstream vertex_shader_file(vertex_shader.c_str(), std::ios::in);
    if(vertex_shader_file.is_open())
    {
        std::string line;
        while(std::getline(vertex_shader_file, line))
            vertex_shader_data += "\n" + line;

        vertex_shader_file.close();
    }

    std::string fragment_shader_data;
    std::ifstream fragment_shader_file(fragment_shader.c_str(), std::ios::in);
    if(fragment_shader_file.is_open())
    {
        std::string line;
        while(std::getline(fragment_shader_file, line))
            fragment_shader_data += "\n" + line;

        fragment_shader_file.close();
    }

    const char * vertex_ptr = vertex_shader_data.c_str();
    const char * fragment_ptr = fragment_shader_data.c_str();
    glShaderSource(vertex_shader_id, 1, &vertex_ptr, NULL);
    glShaderSource(fragment_shader_id, 1, &fragment_ptr, NULL);

    glCompileShader(vertex_shader_id);
    glCompileShader(fragment_shader_id);

    GLuint shader_programme = glCreateProgram();
    glAttachShader(shader_programme, vertex_shader_id);
    glAttachShader(shader_programme, fragment_shader_id);
    glLinkProgram(shader_programme);
```

```
glDeleteShader(vertex_shader_id);
glDeleteShader(fragment_shader_id);

return shader programme;
}
```

Napisaliśmy funkcję, która przyjmuje dwa argumenty. Pierwszym argumentem jest ścieżka do pliku z vertex shader, a drugim jest ścieżka do pliku z fragment shader. W ciele tej funkcji na początku korzystamy z funkcji `glCreateShader()`, której podajemy jakiego rodzaju shader chcemy utworzyć. W tym przypadku chcemy utworzyć vertex shader oraz fragment shader, a więc z takimi parametrami wywołujemy tą funkcję. W wyniku otrzymujemy unikalny identyfikator różny od zera, który potem umożliwi nam odniesienie się do danego shadera.

Po wczytaniu naszych shaderów z plików tekstowych musimy je załadować do stworzonych wcześniej obiektów. Służy do tego funkcja `glShaderSource()`, która wstawia kod źródłowy shadera do obiektu shadera. Pierwszym parametrem jest numer identyfikatora shadera. W drugim parametrze podajemy ile łańcuchów znaków przekazujemy. U nas jest to jeden długi łańcuch znaków. Następnie podajemy tablicę wskaźników pokazujących na łańcuchy znaków zawierające kod źródłowy shadera. Ostatnim parametrem jest tablica zawierająca długości poszczególnych łańcuchów znaków. Tego parametru nie wykorzystujemy.

W kolejnym kroku po załadowaniu kodu źródłowego shaderów do obiektów musimy je skompilować. Wykonywane jest to za pomocą funkcji `glCompileShader()`, której podajemy identyfikator danego shadera.

Po skompilowaniu shaderów musimy połączyć je w jeden obiekt, który jest jakby programem wykonywanym przez GPU. Pierw zatem za pomocą funkcji `glCreateProgram()` tworzymy nowy obiekt i w wyniku otrzymujemy identyfikator nowo utworzonego obiektu. Następnie dodajemy skompilowane shadery do naszego programu. W tym celu używamy funkcji `glAttachShader()`, która przyjmuje identyfikator programu oraz identyfikator danego shadera. Jako, że mamy dwa shadery to musimy tą funkcję wywołać dla każdego shadera z osobna, czyli dwukrotnie. Ostatnim krokiem jest wywołanie funkcji `glLinkProgram()`. Linkuje ona dodane shadery do programu w jeden pojedynczy program.

Pozostało nam jeszcze posprzątać po sobie, czyli zwolnić pamięć zajętą przez pojedyncze shadery oraz zwrócić z funkcji identyfikator obiektu programu, który stworzyliśmy.

Otwórzmy teraz jakiś prosty edytor tekstu, np. Notatnik i stwórzmy dwa pliki (nazwa pliku oczywiście dowolna, rozszerzenie też).

Plik `vertex_shader.glsl`:

```
#version 330
layout(location = 0) in vec3 vp;
void main()
{
    gl_Position = vec4(vp, 1.0);
}
```

Plik fragment_shader.glsł:

```
#version 330
out vec4 frag colour;
void main()
{
    frag_colour = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Działanie tych shaderów wyjaśnię szczegółowo w następnym rozdziale. Wtedy bardziej między innymi wnikniemy w cały potok (pipeline) przetwarzania.

Teraz wystarczy, że w kodzie programu wywołamy naszą funkcję wczytującą shadery.

```
GLuint shaders = loadShaders("vertex_shader.glsł", "fragment_shader.glsł");
```

W wyniku otrzymujemy identyfikator programu shadera. Aby teraz wykorzystać program shadera wystarczy, że w pętli rysowania wywołamy funkcję `glUseProgram()`. Jako jej parametr podajemy identyfikator programu shadera, którego chcemy użyć.

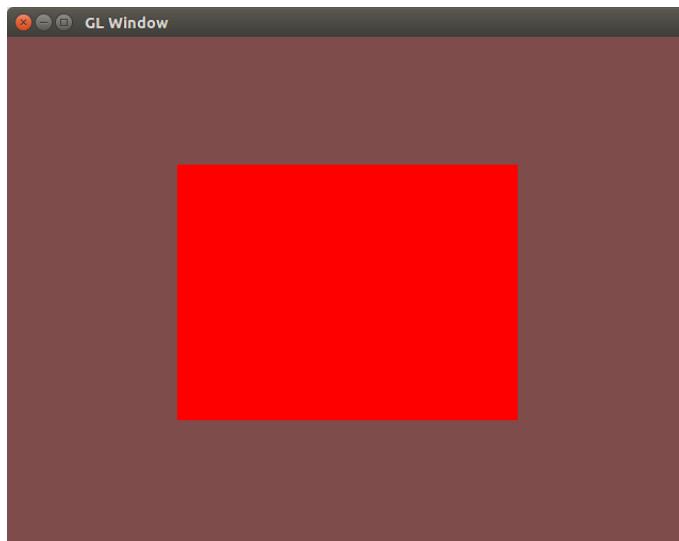
```
while(!glfwWindowShouldClose(window))
{
    glClearColor(0.5f, 0.3f, 0.3f, 1.0f);
    glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);

    glUseProgram(shaders);

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Wynikiem działania tego programu będzie okno jak na rys. 7.1.



Rys. 7.1. Wynik działania programu.

Tips & Tricks:

- Nie należy wykonywać skomplikowanych obliczeń w kodzie programu. Pracę tą należy zostawić programom shader.

- Do pisania programów shader można wykorzystać program Shader Designer (<https://www.opengl.org/sdk/tools/ShaderDesigner/>). Jest to wygodne narzędzie, w którym możemy na bieżąco oglądać efekty naszej pracy.
- Pisząc shadery łatwo jest popełnić błąd. Jeśli coś nie działa jak powinno, należy zweryfikować poprawność shaderów.

Kod źródłowy z tego rozdziału jest w katalogu: **4_Podstawy_shaderow**

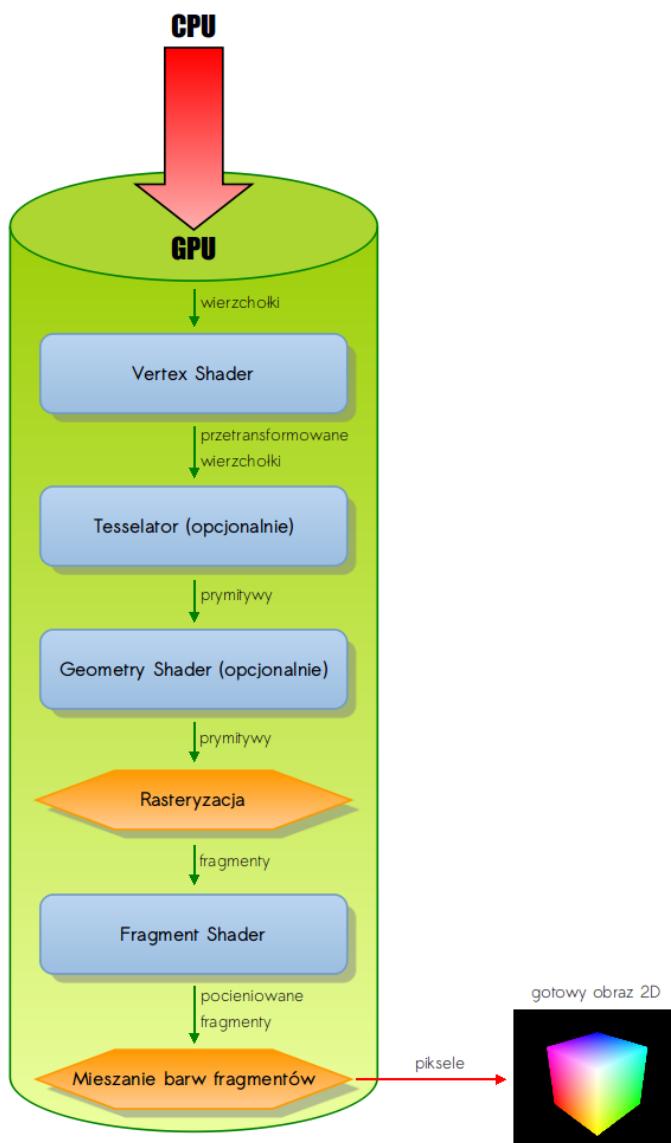
8. Potok przetwarzania

Jak już to wspomniałem w poprzednim rozdziale, shadery są to takie programy, które są uruchamiane na GPU (czyli na karcie graficznej). Definiują one styl renderowania. Określają jak ma dana scena wyglądać ostatecznie. A zatem poprzez shadery możemy zdefiniować swój własny potok (pipeline) graficzny. Potokiem graficznym nazywamy drogę przepływu pomiędzy kartą graficzną a buforem ramki zawierającym gotową klatkę obrazu. Możemy więc zdefiniować swoje własne obliczenia oświetlenia, kolorowania czy animacji.

Minimalnie będziemy zawsze potrzebować dwóch shaderów:

- Vertex Shader
- Fragment Shader

Oczywiście możemy jeszcze zdefiniować kilka dodatkowych shaderów. Ogólnie w dużym uproszczeniu cały potok graficzny można przedstawić tak, jak na rys. 8.1.



Rys. 8.1. Potok graficzny OpenGL.

Vertex Shader jest pierwszym etapem przetwarzania. W danej chwili czasu operuje on tylko na jednym pojedynczym wierzchołku. Z tego właśnie względu, że działa on na pojedynczym wierzchołku nie ma informacji na temat, do jakiego prymitywu należy ten wierzchołek. Czyli tak podsumowując, vertex shader przyjmuje na wejściu jeden wierzchołek i na wyjście także przekazuje wierzchołek (już przetworzony).

Karta graficzna posiada wiele jednostek cieniujących. Ich ilość zależy od zaawansowania karty graficznej. Im karta lepsza, tym więcej jednostek cieniujących posiada. Istnienie dużej ilości tych jednostek zwiększa równoległość obliczeń, gdyż w danej chwili czasu może być obliczane kilka (-dziesiąt, -set, -tysięcy) wierzchołków. Widzimy zatem tutaj dużą przewagę renderowania za pomocą karty graficznej nad renderowaniem programowym za pomocą CPU. Z reguły procesor ma tylko od 1 do 8 rdzeni. Karta graficzna ma ich dużo więcej.

Przypomnę jeszcze, że wierzchołek to nie tylko pozycja w przestrzeni 3D. Wierzchołek bowiem może posiadać kilka atrybutów, takich jak wektor normalny i współrzędne tekstury.

Wróćmy do naszego vertex shadera. Jego kod jest następujący:

```
#version 330
layout(location = 0) in vec3 vp;
void main()
{
    gl_Position = vec4(vp, 1.0);
}
```

Zacznijmy od tego, że każdy shader musi mieć funkcję `main()`. Jej działanie jest takie samo jak w innych językach programowania (np. C++), czyli wewnętrz jej ciała wykonywany jest kod shadera.

Zanim jednak napiszemy funkcję `main()` shadera musimy określić wersję języka GLSL jaką wykorzystujemy do pisania shadera. Robi się to poprzez deklarację `#version`, za którą stawia się numer informujący o wersji języka. Możliwości przedstawione są na rys. 8.2.

Wersja OpenGL	Wersja GLSL	<code>#version <ver></code>
1.2	brak	brak
2.0	1.10.59	<code>#version 110</code>
2.1	1.20.8	<code>#version 120</code>
3.0	1.30.10	<code>#version 130</code>
3.1	1.40.08	<code>#version 140</code>
3.2	1.50.11	<code>#version 150</code>
3.3	3.30.6	<code>#version 330</code>
4.0	4.00.9	<code>#version 400</code>
4.1	4.10.6	<code>#version 410</code>
4.2	4.20.11	<code>#version 420</code>
4.3	4.30.8	<code>#version 430</code>
4.4	4.40	<code>#version 440</code>
4.5	4.50	<code>#version 450</code>

Rys. 8.2. Wersje GLSL.

Wspomniałem wyżej o potoku graficznym. Jest to tak jakby manufaktura produkująca obraz. Różne etapy produkcji następują po sobie przekazując sobie dane. To tak jak na przykład w fabryce telewizorów. Skompletowane elementy elektroniczne przekazywane są do lutowania. Gdy proces lutowania się zakończy gotowa płytka drukowana przekazywana jest dalej do działu montażu obudowy. Po zamontowaniu obudowy gotowy telewizor przekazywany jest do działu pakowania do pudełka, a potem dalej do transportu. Aby właśnie taki potok działał prawidłowo shadery muszą sobie przekazywać informacje. Atrybuty wejściowe shadera definiuje się za pomocą słowa kluczowego `in`. W naszym shaderze mamy tylko jedną zmienną wejściową typu `vec3` o nazwie `vp`. Pamiętacie o tym jak za pomocą funkcji `glVertexAttribPointer()` tworzyliśmy wskaźnik na atrybut wierzchołka? Wtedy tym atrybutem było położenie (współrzędne) wierzchołków i przypisaliśmy im indeks 0 (pierwszy argument funkcji `glVertexAttribPointer()`). Tutaj za pomocą słowa kluczowego `layout` określamy indeks, z jakiego ma być odczytywana ta zmienna wejściowa.

Przejdzmy dalej. Jak widzimy ciało funkcji `main()` jest ubogie, jest tu tylko jedna instrukcja i tajemnicza nigdzie nie zdefiniowana zmienna `gl_Position`. Otóż vertex shader posiada wewnętrznie zdefiniowaną taką strukturę:

```
out gl_PerVertex {  
    vec4 gl_Position;  
    float gl_PointSize;  
    float gl_ClipDistance[];  
};
```

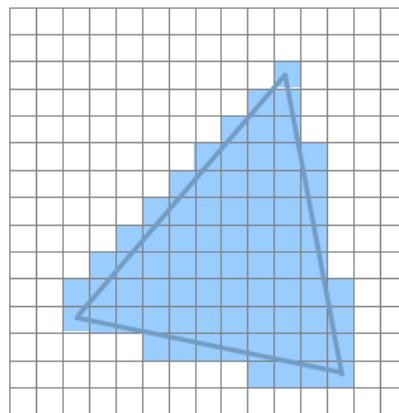
Widzimy, że znajduje się w niej zmienna `gl_Position` i jest typu `vec4`, czyli jest to wektor cztero-elementowy. Widzimy także, że zmienna ta zdefiniowana jest jako zmienna wyjściowa `out`, a zatem jej wartość będzie przekazana dalej. Wpisujemy do niej żądane współrzędne wierzchołka na ekranie. Z tego względzu, że jest to wektor cztero-elementowy to uzupełniamy go trój-elementowym wektorem wejściowym `vp` oraz czwartym składnikiem równym 1. Ten czwarty składnik wynika z współrzędnych jednorodnych. Oczywiście moglibyśmy w jakiś sposób zmodyfikować wektor wejściowy `vp`, ale my po prostu przepisujemy dane wejściowe na wyjście. A zatem w wyniku otrzymamy takie samo położenie wierzchołków na ekranie jak zdefiniowaliśmy w tablicy w kodzie naszego programu.

Dalszymi krokami potoku graficznego są teselacja oraz geometry shader. Są to kroki opcjonalne i nimi na razie nie będziemy się zajmować. Powiem tylko, że teselacja polega na generowaniu nowych prymitywów w celu poprawy jakości wyświetlanego obrazu. Geometry shader natomiast jest jakby uzupełnieniem vertex shader, gdyż posiada on pełną wiedzę na temat prymitywu, na którym aktualnie operuje.

Rasteryzacja (rys. 8.3) jest procesem polegającym na wyznaczeniu pikseli należących do danego prymitywu. Wróćmy do naszego trójkąta, którego rysowaliśmy w poprzednim kursie. Zdefiniowaliśmy wtedy trzy wierzchołki, które zostały połączone liniami. Trzeba zatem określić, które piksele wchodzą w skład tego trójkąta. Mając tą wiedzę możemy pokolorować dany trójkąt na żądaną kolor.

Każdy taki pojedynczy mały obszar, który trzeba zamalować nazywany jest **fragmentem**. Każdy fragment posiada swoje współrzędne XY na ekranie oraz głębokość Z. Zadaniem fragment shader jest zatem koloryzacja tych wszystkich wyznaczonych podczas rasteryzacji fragmentów. Tak samo jak w przypadku vertex shader, jeden fragment shader zajmuje się koloryzacją tylko jednego fragmentu.

Nie należy mylić fragmentu obrazu z pikselem. Jest to trochę co innego. Piksel jest elementem, który tworzy wynikowy obraz 2D na ekranie monitora. Fragmentem jest obszar o wielkości jednego piksela. Czasami może być tak, że kilka fragmentów nakłada się w pewnym miejscu na siebie. W takiej sytuacji jeden piksel tworzony jest przez kilka fragmentów. Na przykład mamy scenę, która posiada pewną głębię. W oddali rysujemy jakiś obszar na czerwono, natomiast później rysujemy jakiś obiekt bliżej nas na zielono. W wyniku tego czerwone fragmenty w oddali zostają przystoione przez zielone. Ostatecznie więc otrzymujemy zielone piksele na ekranie.



Rys. 8.3. Uproszczone działanie rasteryzacji.

Nasz fragment shader wygląda następująco:

```
#version 330
out vec4 frag_colour;
void main()
{
    frag colour = vec4(1.0, 0.0, 0.0, 1.0);
```

Tutaj także widzimy deklarację `#version` oraz funkcję `main()`, tak samo jak w przypadku vertex shader. Zadaniem fragment shader jest koloryzacja fragmentów, a zatem definiujemy zmienną wyjściową `out`, która będzie kolorem danego fragmentu. Zmienna `frag_colour` jest typu `vec4`, gdyż musimy podać kolor w postaci Red-Green-Blue-Alpha. My ustawiamy kolor wszystkich fragmentów na czerwony.

Jeszcze takie małe wyjaśnienie na koniec. Zmienna `frag_colour` może mieć dowolną nazwę. A więc, możemy nazwać ją po prostu `kolor_wyjsciowy`. To pewnie zastanawiasz się skąd potem będzie nasz program wiedział, że w tej zmiennej jest zapisany żądany kolor fragmentu. A stąd, że jest to pierwsza definiowana zmienna wyjściowa i ma przypisany domyślnie indeks 0 (`layout`, o którym już wyżej wspomniałem). Potem w programie zostanie właśnie kolor odczytany z zmiennej o indeksie 0. Cała magia.

Temat shaderów jest tematem bardzo obszernym i zawiłym. Wiem, że zostaje jeszcze dużo niejasności po tym rozdziale, no ale temat ten będzie stale rozszerzany i wiedza także będzie stopniowo uzupełniana.

9. Obiekty VAO

W tym rozdziale pokażę jak wykorzystać kilka obiektów VAO, czyli narysujemy odrębne modele. Pobawimy się też trochę shaderami.

Pamiętamy, że obiekt VAO jest obiektem, który łączy w sobie kilka atrybutów takich jak **położenie wierzchołków na ekranie, współrzędne tekstury oraz wektory normalne**. Aby posiadać kilka odrębnych modeli musimy również posiadać kilka oddzielnych obiektów VAO. Zdefiniujmy je więc sobie:

```
GLfloat points[] = {
    -0.5f, -0.5f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
};

GLuint vbo = 0;
 glGenBuffers(1, &vbo);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);

GLuint vao1 = 0;
 glGenVertexArrays(1, &vao1);
 glBindVertexArray(vao1);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);

GLuint vao2 = 0;
 glGenVertexArrays(1, &vao2);
 glBindVertexArray(vao2);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);

GLuint vao3 = 0;
 glGenVertexArrays(1, &vao3);
 glBindVertexArray(vao3);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);
```

Zdefiniowaliśmy na początku współrzędne wierzchołków trójkąta. Z tego względu, że mam zamiar Wam pokazać modyfikację położenia wierzchołków przez vertex shader to wystarczy nam jedna wspólna tablica położenia wierzchołków dla trzech modeli.

Dalej tworzymy obiekt VBO i wpisujemy do niego współrzędne wierzchołków. Następnie definiujemy trzy oddzielne obiekty VAO i każdemu z nich przypisujemy ten sam obiekt VBO, czyli tą samą tablicę współrzędnych. Nie było tu nic skomplikowanego.

Chcę, aby każdy z obiektów miał swoje shadery, a więc wczytuję je:

```
GLuint shaders1 = loadShaders("vertex_shader1.glsl", "fragment_shader1.glsl");
GLuint shaders2 = loadShaders("vertex_shader2.glsl", "fragment_shader2.glsl");
GLuint shaders3 = loadShaders("vertex_shader3.glsl", "fragment_shader3.glsl");
```

Spójrzmy teraz na kod tych shaderów.

vertex_shader1.glsl:

```
#version 330
```

```
layout(location = 0) in vec3 vp;
void main()
{
    gl_Position = vec4(vp.x - 0.25, vp.y - 0.25, vp.z, 1.0);
```

vertex_shader2.glsl:

```
#version 330
layout(location = 0) in vec3 vp;
void main()
{
    gl_Position = vec4(vp.x + 0.25, vp.y - 0.25, vp.z, 1.0);
```

vertex_shader3.glsl:

```
#version 330
layout(location = 0) in vec3 vp;
void main()
{
    gl_Position = vec4(vp.x, vp.y + 0.25, vp.z, 1.0);
```

fragment_shader1.glsl:

```
#version 330
out vec4 frag_colour;
void main()
{
    frag_colour = vec4(1.0, 0.0, 0.0, 1.0);
```

fragment_shader2.glsl:

```
#version 330
out vec4 frag_colour;
void main()
{
    frag_colour = vec4(0.0, 1.0, 0.0, 1.0);
```

fragment_shader3.glsl:

```
#version 330
out vec4 frag_colour;
void main()
{
    frag_colour = vec4(0.0, 0.0, 1.0, 1.0);
```

Co tu jest ciekawego? Widzimy jak można modyfikować położenie wierzchołków na ekranie za pomocą vertex shader. Zmienna wejściowa `vp` posiada swoje składowe określające poszczególne współrzędne. Wystarczy, że dodamy do nich jakąś wartość, a wierzchołek na ekranie zostanie przesunięty.

Co do kolorowania to widzimy, że w każdym fragment shader wpisujemy inny kolor wyjściowy.

Napiszmy teraz główną pętlę rysującą:

```
while(!glfwWindowShouldClose(window))
{
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(shaders1);
    glBindVertexArray(vaol);
    glDrawArrays(GL_TRIANGLES, 0, 3);
```

```
glUseProgram(shaders2);
glBindVertexArray(vao2);
glDrawArrays(GL_TRIANGLES, 0, 3);

glUseProgram(shaders3);
glBindVertexArray(vao3);
glDrawArrays(GL_TRIANGLES, 0, 3);

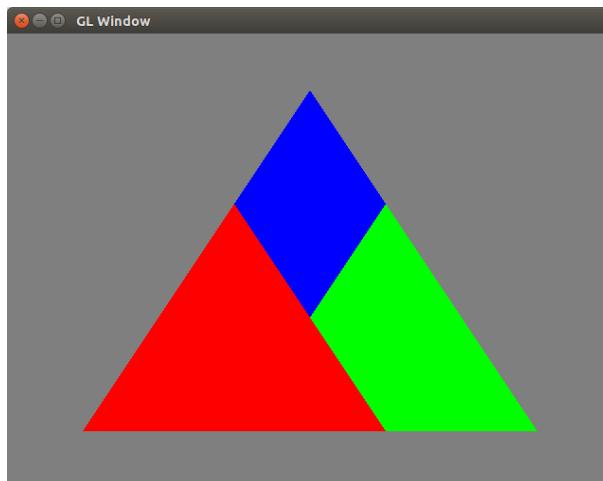
glfwSwapBuffers(window);
glfwPollEvents();
}
```

Dla każdego rysowanego obiektu pierw wczytujemy jego shadery, ustawiamy dany obiekt VAO jako aktywny i go rysujemy.

Żeby wszystko ładnie wyglądało musimy jeszcze powyżej pętli rysowania włączyć **testowanie głębi**. Na razie nie będę tego dokładnie wyjaśniał. Polega to na pewnej inteligencji GPU. Wykrywa on po prostu, które obiekty znajdują się dalej na scenie i zaczyna rysowanie od nich.

```
 glEnable(GL_DEPTH_TEST);
 glDepthFunc(GL_LESS);
```

W wyniku otrzymamy efekt taki jak na rys. 9.1.



Rys. 9.1. Końcowy efekt działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: 5_Obiekty_VAO

10. Antialiasing i biblioteka GLFW

To będzie krótki rozdział. Jedynie małe uzupełnienie na temat pewnych przydatnych funkcji biblioteki GLFW.

Jeśli uruchamialiście poprzednie programy to może zauważycie, że skośne krawędzie są nieco poszarpane. Wynika to z rasteryzacji. Biblioteka GLFW pozwala nam włączyć w OpenGL **anti-aliasing** w celu wygładzenia krawędzi. Wystarczy do tego jedna instrukcja.

```
glfwWindowHint(GLFW_SAMPLES, 4);
```

Drugim parametrem tej funkcji jest ilość próbek. Ogólnie im więcej próbek tym krawędzie będą gładzsze, ale też zwiększy się ilość obliczeń powodujących spadek wydajności. **Uwaga:** funkcję tą musimy wywołać jeszcze przed funkcją `glfwCreateWindow()`.

Mozemy także wewnątrz naszego programu określić jakiej co najmniej wersji OpenGL on wymaga. Jeśli używamy teselacji, która jest dostępna dopiero w OpenGL od wersji 4.0, to wtedy nasz program nie powinien uruchomić się na komputerach nie posiadających OpenGL w wersji 4.0 lub nowszych. Robimy to w sposób następujący:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Mozemy oczywiście wpisać inną wymaganą wersję. Wystarczy, że zmienimy wartość `MAJOR` na np. 3, a wartość `MINOR` na np. 2.

11. Tryb pełnoekranowy i zmiana rozmiaru okna

Większość gier komputerowych uruchamianych jest na pełnym ekranie. Pokażę w tym rozdziale jak tego dokonać.

W jednym z pierwszych rozdziałów pokazałem już jak możemy wyświetlić nasz program w formie pełno-ekranowej. Ale niestety musielibyśmy wtedy podać ręcznie rozdzielczość okna. Nie jest to wygodne, gdyż nasz program nie jest wtedy przystosowany do pracy na innych rozdzielczościach. Ręczna modyfikacja nie zawsze jest wygodna. Teraz pokażę jak wykryć używaną rozdzielczość i ustawić ją automatycznie.

```
const GLFWvidmode* video_mode = glfwGetVideoMode(glfwGetPrimaryMonitor());  
GLFWwindow* window = glfwCreateWindow(video mode->width, video mode->height, "GL Window",  
glfwGetPrimaryMonitor(), NULL);
```

Za pomocą funkcji `glfwGetVideoMode()` odczytujemy aktualną rozdzielczość głównego monitora, a później w funkcji tworzącej nowe okno ustawiamy szerokość oraz wysokość, czyli żądaną rozdzielczość okna. To tyle. Czyli jak widzimy wystarczyła jedna instrukcja więcej.

Teraz zrobimy coś innego. Czasami jednak wygodniej jest pracować nie na pełnym ekranie, a w okienku. Niestety, jeśli użytkownik zmieni rozmiar okna, jego zawartość z tym, co narysowaliśmy nie przeskakuje się automatycznie.

Na początku zdefiniujmy sobie dwie zmienne globalne.

```
int window_height = 600;  
int window_width = 800;
```

Jak wiemy (i już mówiłem o tym) okna w systemie operacyjnym działają na zdarzeniach. Zmiana rozmiaru okna jest także zdarzeniem, które trzeba obsłużyć. Zdefiniujmy więc funkcję, która będzie obsługiwała zdarzenie zmiany rozmiaru okna.

```
void window_size_callback(GLFWwindow* window, int width, int height)  
{  
    window_width = width;  
    window_height = height;  
}
```

Co tu robimy, to jedynie aktualizujemy nasze globalne zmienne przechowujące rozmiar okna aktualnym rozmiarem okna. Parametry `width` i `height` funkcji `window_size_callback()` zawierają bowiem aktualny rozmiar okna.

Teraz musimy naszemu oknu przypisać tą funkcję obsługi zdarzenia. Służy do tego funkcja `glfwSetWindowSizeCallback()`.

```
glfwSetWindowSizeCallback(window, window_size_callback);
```

Ostatnią czynnością jest aktualizacja rozmiaru naszej rzutni (viewport). Rzutnia jest to ten obszar wewnętrz okna, na którym rysujemy. Nazywamy go rzutnią, ponieważ obraz 3D jest spłaszczany (rzutowany) na obraz (ekran) 2D. Żeby zmiany rozmiaru okna były od razu

uwzględnianie musimy oczywiście aktualizację rozmiaru rzutni wrzucić do pętli rysującej. Do aktualizacji rozmiaru rzutni służy funkcja `glViewport()`.

```
glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glViewport(0, 0, window_width, window_height);
```

Funkcja `glViewport()` przyjmuje cztery parametry. Pierwsze dwa to współrzędne XY położenia rzutni na ekranie. Natomiast drugie dwa to szerokość oraz wysokość rzutni.

Mozemy teraz uruchomić nasz program i zobaczymy, że zmieniając rozmiar okna jego zawartość będzie skalowana.

Kod źródłowy z tego rozdziału jest w katalogu: 6_Pelny_ekran

12. Maszyna stanu

Zaczniemy od wyjaśnienia, czym jest **maszyna stanu** w OpenGL. Przechowuje ona otóż pewne stany, które dostępne są globalnie. Globalnie, to znaczy, że mamy do nich dostęp z każdego miejsca naszego programu.

Maszyna stanu przechowuje informacje na temat:

- aktywnych buforów, np. VBO,
- aktywnych trybów renderowania, np. wykorzystanie funkcji głębokości,
- aktywnych shaderów.

Maszynę stanu wykorzystywaliśmy już wiele razy. A kiedy? Wtedy, gdy wykorzystywaliśmy **funkcje zmieniające stan maszyny stanu**. Tymi funkcjami są:

- funkcje aktywujące dany bufor - `glBindBuffer()` i `glBindVertexArray()`,
- funkcja włączająca dany tryb renderowania - `glEnable()`,
- funkcja aktywująca dany shader - `glUseShader()`.

Jeśli zatem użyjemy którejś z tych funkcji, zmieniamy zapisany stan w maszynie stanu. Od tej pory ten nowy stan jest aktywny globalnie. Na przykład, gdy aktywujemy shader za pomocą funkcji `glUseShader()` to od tej pory on staje się globalnie aktywnym shaderem rysowania.

Maszyna stanu może być także powodem problemów. Ustawimy jakiś stan i zapomnimy o tym, np. ustawimy nieprawidłowy aktywny bufor VBO. W wyniku otrzymujemy nie takie efekty, jakich żądamy. Należy mieć to na uwadze, żeby potem się w tym wszystkim nie pogubić.

Mogliśmy sprawdzić aktywny stan maszyny stanu za pomocą rodziny funkcji `glGet()`, np. `glGetBooleanv()`, `glGetIntegerv()`, `glGetDoublev()`. Na przykład, żeby odczytać aktualnie ustawiony bufor VBO możemy wywołać funkcję `glGetIntegerv()` z parametrem `GL_ARRAY_BUFFER_BINDING`.

```
int index_vbo = 0;
glGetIntegerv(GL_ARRAY_BUFFER_BINDING, &index_vbo);
std::cout << "Active VBO index: " << index_vbo;
```

Drugim parametrem funkcji `glGetIntegerv()` jest adres celu, do którego należy zapisać pobraną wartość. Oczywiście w tym przypadku użyliśmy funkcji `glGetIntegerv()`, gdyż pobieramy liczbę całkowitą. Do innych typów zmiennych są inne funkcje. Te podstawiłem na początku tego akapitu.

Lista stanów, które można pobrać za pomocą `glGet()` jest bardzo długa. W najbliższej przyszłości zagłębimy się w to bardziej. Na razie tyle informacji na temat maszyny stanu wystarczy.

Tips & Tricks:

- Należy pilnować co mamy aktualnie ustawione w maszynie stanu. Na przykład funkcja `glDrawArrays()` korzysta z aktualnie aktywnego bufora VAO i aktualnie aktywnego programu shadera. Jeśli coś rysuje się nie tak jak powinno, szukanie błędu należy zacząć od maszyny stanu.

13. Korzystanie z obiektów VBO

Wróćmy znowu do tematu obiektów VBO. Chciałbym między innymi zaznaczyć, że tak na prawdę nie jest to taki obiekt rozumiany w stylu zorientowanym obiektywem. Nie jest to żadna klasa. **Jest to po prostu tablica danych**, zazwyczaj typu `float`. Przyjęto się natomiast mówić obiekt VBO, a nie tablica VBO, więc zostaniemy przy tej konwencji.

Już mówiłem o tym kilka razy, że jakiś nasz model trójwymiarowy może składać się z kilku buforów VBO odpowiedzialnych za położenie wierzchołków, tekstury i wektorów normalnych. Ogólnie rzecz biorąc, w buforach VBO możemy trzymać dowolne dane. Ważne, abyśmy my wiedzieli, do czego one są.

Mówiłem też, że obiekt VBO jest jakby strumieniem danych ułożonych w szereg. To od nas także zależy jak będziemy je interpretować. W funkcji `glVertexAttribPointer()` w drugim parametrze podajemy ile liczb z bufora ma być pobranych na każdy wierzchołek. Wpisywaliśmy tam wartość równą 3, gdyż do obiektu VBO wpisywaliśmy liczby w postaci X, Y, Z, X, Y, Z, X, Y, Z, itd.

Możemy teraz zdefiniować sobie kolejny obiekt VBO, ale nie będzie on przechowywał położenia wierzchołków na ekranie. Będzie to bufor przechowujący kolory poszczególnych wierzchołków. Wartości podajemy w kolejności Red, Green, Blue, Red, Green, Blue, itd.

```
GLfloat colours[] = {
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f,
};
```

Jako, że rysujemy dalej trójkąt na ekranie to podaliśmy trzy kolory (na każdy kolor przypadają trzy wartości - składowe R, G i B). Każdy wierzchołek trójkąta będzie miał zatem inny kolor. Stwórzmy teraz nowy obiekt VBO.

```
GLuint position_vbo = 0;
 glGenBuffers(1, &position_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);

GLuint colour_vbo = 0;
 glGenBuffers(1, &colour_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, colour_vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(colours), colours, GL_STATIC_DRAW);
```

Teraz nadeszła pora na zdefiniowanie obiektu VAO. Będzie on składał się z dwóch obiektów VBO. Spójrzmy pierw na kod.

```
GLuint vao1 = 0;
 glGenVertexArrays(1, &vao1);
 glBindVertexArray(vao1);
 glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindBuffer(GL_ARRAY_BUFFER, colour_vbo);
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);
 glEnableVertexAttribArray(1);
```

Przypomnę, że danej chwili w maszynie stanu może być tylko jeden aktywny bufor VBO. A zatem na początku ustawiamy bufor VBO przechowujący położenie wierzchołków jako aktywny za pomocą funkcji `glBindBuffer()`. Następnie tworzymy wskaźnik na ten bufor wewnątrz obiektu VAO i przypisujemy mu indeks równy 0 (pierwszy parametr funkcji `glVertexAttribPointer()`). Dalej ustawiamy bufor VBO przechowujący kolory wierzchołków za aktywny i tworzymy wskaźnik na ten bufor wewnątrz obiektu VAO z indeksem równym 1. Koniecznie musimy pamiętać o uaktywnieniu tych wskaźników za pomocą funkcji `glEnableVertexAttribArray()`. Zwróćcie jeszcze uwagę na to, że przy tworzeniu wskaźnika za pomocą funkcji `glVertexAttribPointer()` na obiekt VBO zawierający kolory podałem także wartość równą 3 przy liczbie danych przypadających na jeden wierzchołek.

Teraz musimy zmodyfikować nasze shadery. Zaczniemy od vertex shader. Nowy kod wygląda następująco:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 colour;

out vec3 vertex_colour;

void main()
{
    vertex colour = colour;
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
```

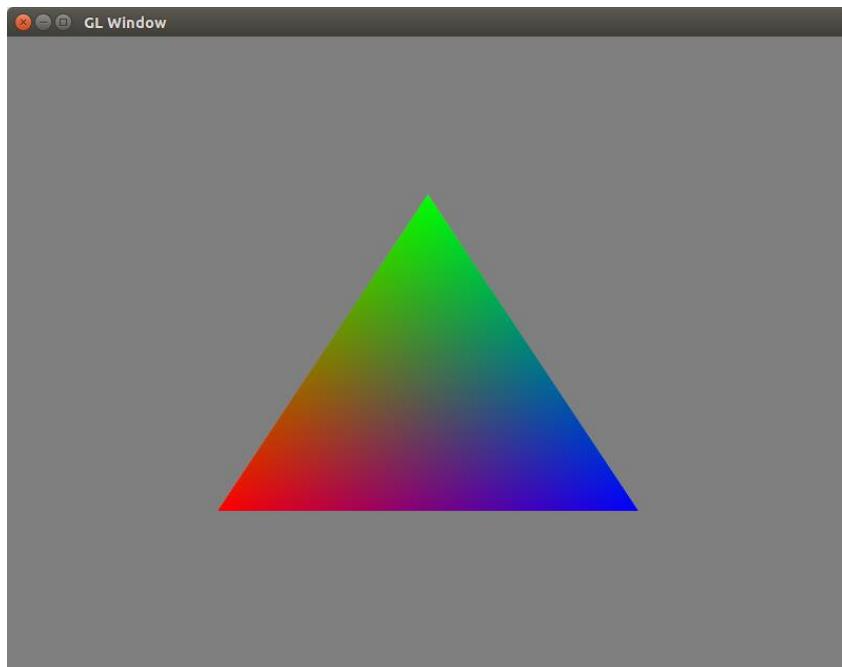
Wykorzystujemy tutaj dwie zmienne wejściowe do vertex shadera. Pierwsza zmienna wejściowa pochodzi z warstwy o indeksie 0, a druga z warstwy o indeksie 1. Wyżej zdefiniowaliśmy dwa wskaźniki. Pierwszy o indeksie 0 pokazywał na obiekt VBO zawierający położenie wierzchołków, a drugi o indeksie 1 pokazywał na obiekt VBO zawierający kolory wierzchołków. Odnosimy się tutaj zatem do tych dwóch wskaźników. Definiujemy także zmienną wyjściową `vertex_colour`. W funkcji `main()` przypisujemy jej wartość zmiennej wejściowej zawierającą kolor aktualnie przetwarzanego wierzchołka. Z tej racji, że w vertex shader nie możemy kolorować wierzchołków, musimy kolor wierzchołka przekazać do kolejnego etapu potoku przetwarzania. Jest nim oczywiście fragment shader. Przyjrzyjmy się.

```
#version 400
in vec3 vertex_colour;
out vec4 frag_colour;

void main()
{
    frag colour = vec4(vertex colour, 1.0);
}
```

Fragment shader przyjmuje zmienną wejściową o takiej samej nazwie jaką jest na wyjściu z vertex shader, czyli `vertex_colour`. Jest to kolor danego wierzchołka. Wykorzystujemy go podczas ustawiania wartości `frag_colour`, która jest kolorem przetwarzanego fragmentu. Zmienna wejściowa jest typu `vec3`, a zmienna wyjściowa `frag_colour` jest typu `vec4`. Ten czwarty ostatni składnik będący składową alpha koloru musimy podać ręcznie.

Po uruchomieniu programu zobaczymy na ekranie takie okienko jak na rys. 13.1.



Rys. 13.1. Efekt działania programu.

Widzimy nasz trójkąt pokolorowany na kolory tęczy. Ciekawostką są tutaj właśnie te kolory. Zdefiniowaliśmy przecież, że każdy wierzchołek ma mieć inny kolor. Tak też się stało. Ale też trzeba jakoś wypełnić tą przestrzeń pomiędzy wierzchołkami. Zastosowana tutaj została interpolacja, czyli wprowadzone zostały automatycznie kolory pośrednie.

Kod źródłowy z tego rozdziału jest w katalogu: 7_VBO

14. Weryfikacja shaderów

Debugowanie shaderów **nie jest łatwe**. Nie możemy dodać wewnątrz shadera linii kodu odpowiedzialnych za wyświetlenie na ekranie, np. wartości jakiejś zmiennej. Czasami mały błąd w kodzie shadera powoduje, że nic na ekranie się nam nie wyświetla. Takie sytuacje mogą powodować wyrywanie włosów z głowy. W tym rozdziale pokażę kilka łatwych sposobów na kontrolę kodu shadera.

Pierwszą metodą jest sprawdzenie stanu komplikacji shadera. Do tego celu można użyć funkcji `glGetShaderiv()`. Ogólnym jej przeznaczeniem jest pobieranie różnych parametrów z obiektów shadera. Przyjmuje ona trzy parametry: pierwszym jest identyfikator danego shadera, następnie nazwa parametru oraz wskaźnik na `GLint`, gdzie zapisana zostanie zwrócona wartość. My skorzystamy z parametru `GL_COMPILE_STATUS`, aby uzyskać status komplikacji. W wyniku otrzymamy `GL_TRUE`, gdy komplikacja przebiegła prawidłowo. W przeciwnym razie otrzymamy wartość `GL_FALSE`.

Zmodyfikujmy naszą funkcję wczytującą shadery:

```
int vertex_shader_status = -1;
glGetShaderiv(vertex_shader_id, GL_COMPILE_STATUS, &vertex_shader_status);
if(vertex_shader_status != GL_TRUE)
{
    std::cout << "Vertex shader \\" << vertex_shader << "\"" compile error!" << std::endl;
    return -1;
}
else
    std::cout << "Vertex shader \\" << vertex_shader << "\"" compile success!" << std::endl;

int fragment_shader_status = -1;
glGetShaderiv(fragment_shader_id, GL_COMPILE_STATUS, &fragment_shader_status);
if(fragment_shader_status != GL_TRUE)
{
    std::cout << "Fragment shader \\" << fragment_shader << "\"" compile error!" << std::endl;
    return -1;
}
else
    std::cout << "Fragment shader \\" << fragment_shader << "\"" success!" << std::endl;
```

W przypadku, gdy wykryty zostanie błąd komplikacji shadera wyjdziemy z funkcji wczytującej shadery z wartością równą -1. Potem wystarczy, że w funkcji `main()` przechwycimy ten stan i poczynimy odpowiednie działanie.

Podobnie jak wyżej możemy sprawdzić stan linkowania programu shadera. Tym razem użyjemy funkcji `glGetProgramiv()`. Tak samo jak funkcja podana powyżej przyjmuje ona trzy parametry: identyfikator programu shadera, żądany parametr oraz wskaźnik na zmienną `GLint`. Chcemy sprawdzić stan linkowania, a zatem funkcję tą musimy wywołać z parametrem `GL_LINK_STATUS`. Jeśli program został poprawnie zlinkowany to otrzymamy wartość `GL_TRUE`. W przypadku braku powodzenia otrzymamy wartość `GL_FALSE`.

```
glLinkProgram(shader_programme);
int link_status = -1;
glGetProgramiv(shader_programme, GL_LINK_STATUS, &link_status);
if(link_status != GL_TRUE)
{
    std::cout << "Shader programme link error!" << std::endl;
    return -1;
}
```

```
}

else
    std::cout << "Shader programme link success!" << std::endl;
```

Istnieje także możliwość wyświetlenia bardziej szczegółowych logów na temat stanu danego shadera bądź programu shadera. Do tego celu można wykorzystać funkcje `glGetShaderInfoLog()` oraz `glGetProgramInfoLog()`. Przyjmuję one podobne parametry:

- identyfikator danego shadera (dla `glGetShaderInfoLog()`) bądź programu shadera (dla `glGetProgramInfoLog()`),
- długość bufora, do którego log zostanie zapisany,
- zwrocona długość danych,
- wskaźnik na bufor.

Używając funkcji `glGetShaderInfoLog()` możemy się między innymi dowiedzieć, w której linii kodu shadera wystąpił błąd.

```
const int max_length = 2048;
int length = 0;
char log_text[max_length];
glGetShaderInfoLog(vertex_shader_id, max_length, &length, log_text);
std::cout << log_text;
```

W wyniku otrzymamy, np. komunikat:

```
0(3) : warning C7568: #version 400 not fully supported on current GPU target profile
0(11) : error C1008: undefined variable "gl_Poition"
```

Widzimy tutaj, że przede wszystkim zrobiliśmy literówkę w nazwie zmiennej `gl_Position` oraz podana wersja shadera 400 nie jest w pełni obsługiwana przez GPU.

W skład programu shadera jak wiemy wchodzi kilka shaderów. Aby zatem otrzymać informację zbiorczą na temat każdego z shaderów nie musimy dla każdego z nich wywoływać funkcji `glGetShaderInfoLog()`. Wystarczy, że użyjemy funkcji `glGetProgramInfoLog()`. Pobierze ona nam log programu shadera w wygodniej do czytania formie, tzn. z podziałem na poszczególne shadery.

```
int length = 0;
char log_text[max_length];
glGetProgramInfoLog(shader_programme, max_length, &length, log_text);
std::cout << log_text;
```

Po uruchomieniu otrzymamy na przykład taki komunikat:

```
Vertex info
-----
0(3) : warning C7568: #version 400 not fully supported on current GPU target profile
0(11) : error C1008: undefined variable "gl Poition"
Fragment info
-----
0(3) : warning C7568: #version 400 not fully supported on current GPU target profile
0(8) : error C1031: swizzle mask element not present in operand "w"
0(8) : warning C1068: array index out of bounds
```

Ciekawym narzędziem do przeprowadzania walidacji shaderów jest **Glslang**. Jest to kompilator języka shaderów, czyli GLSL. Istnieje zarówno wersja dla systemu Windows jak

i Linux. Używanie jego jest bardzo proste. Musimy tylko zadbać o odpowiednie rozszerzenia plików naszych shaderów. Wymagania są następujące:

- **.vert** - dla vertex shader,
- **.frag** - dla fragment shader.

Używamy go poprzez wywołanie z linii poleceń, np. tak:

```
glslangValidator.exe fragment_shader1.vert
```

W konsoli zostaną nam wtedy wyświetcone ewentualne komunikaty błędów.

15. Zmienne uniform

W tym rozdziale temat związany z shaderami. Poznaliśmy już wcześniej zmienne wejściowe oraz wyjściowe shadera. Teraz zajmiemy się zmiennymi typu `uniform`.

Używając zmiennych `uniform` mamy możliwość przesyłania wartości z CPU do programu shadera. Czyli pozwalają nam na komunikację pomiędzy naszą aplikacją, a programem shadera pracującym na GPU. Brzmi ciekawie. Ale zanim przejdziemy dalej, powiem jeszcze, jakie są podstawowe typy zmiennych, które możemy użyć z kwalifikatorem `uniform`. Otóż są to następujące typy:

- `bool` - wartość logiczna,
- `int` - liczba całkowita ze znakiem,
- `float` - liczba zmiennoprzecinkowa,
- `vec3` - wektor trójwymiarowy zmiennoprzecinkowych,
- `vec4` - wektor czterowymiarowy zmiennoprzecinkowych,
- `mat3` - macierz 3x3 liczb zmiennoprzecinkowych,
- `mat4` - macierz 4x4 liczb zmiennoprzecinkowych.

Zaczniemy od przykładu, żeby łatwiej było tłumaczyć. Otwórzmy kod vertex shader i dodajmy do niego zmienną `uniform`.

```
#version 330
layout(location = 0) in vec3 position;

uniform float trans;

void main()
{
    gl_Position = vec4(position.x + trans, position.y + trans, position.z, 1.0);
}
```

Widzimy już jak definiuje się zmienne `uniform`. Na początku stawiamy kwalifikator `uniform`, a później typ zmiennej oraz jej nazwa. Widzimy także, że wartość tej zmiennej będziemy wykorzystywać do przesuwania współrzędnych, gdyż do współrzędnej `x` oraz `y` dodajemy wartość `trans`.

Przejdźmy teraz do kodu naszej aplikacji. Zdefiniowaliśmy sobie zmienną `uniform` wewnątrz naszego programu shadera, teraz musimy jakoś się do niej dostać, aby można było modyfikować jej wartość. Służy do tego funkcja `glGetUniformLocation()`, która zwraca położenie danej zmiennej w programie shadera. Podajemy jej dwa parametry:

- identyfikator programu shadera,
- string zawierający nazwę tej zmiennej.

W praktyce wygląda to następująco:

```
GLuint shaders = LoadShaders("vertex shader.gls1", "fragment shader.gls1");
GLint trans_uniform = glGetUniformLocation(shaders, "trans");
```

Nasza zmienna w programie shadera nazywała się `trans`, a zatem też taką samą nazwę przekazujemy w formacie łańcucha znaków w drugim parametrze. W wyniku otrzymujemy liczbę całkowitą określającą lokalizację danej zmiennej `uniform` w programie shadera. Nazwa podana w łańcuchu znaków nie może zawierać żadnych białych znaków. Ale to chyba jasne, bo nazwa zmiennej nie może mieć w sobie białych znaków.

Aby teraz wpisać nową wartość do tej zmiennej `uniform` należy skorzystać z funkcji `glUniform1f()`. Nazwa tej funkcji nawiązuje do typu zmiennej `uniform`, którą zdefiniowaliśmy. Umożliwia modyfikację jednej (stąd ta jedynka w nazwie funkcji) zmiennej `uniform` typu `float` (literka '`f`' na końcu nazwy).

```
glUniform1f(trans_uniform, 0.5);
```

Pierwszym parametrem tej funkcji jest uzyskana wcześniej lokalizacja zmiennej w programie shadera, a drugim parametrem jest oczywiście wartość, jaką ma ta zmienna przyjąć.

Teraz wykorzystamy zmienną `vec4` do ustawiania koloru wewnątrz fragment shader. Zmodyfikujmy więc kod fragment shader:

```
#version 330
out vec4 frag_colour;
uniform vec4 colour;
void main()
{
    frag_colour = colour;
}
```

Jak widzimy, zdefiniowaliśmy zmienną `uniform` typu `vec4`. Teraz wpiszemy do niej jakąś wartość z wnętrza aplikacji.

```
GLuint shaders = LoadShaders("vertex_shader1.gls1", "fragment_shader1.gls1");
GLuint colour_uniform = glGetUniformLocation(shaders, "colour");
// ...
glUniform4f(colour_uniform, 0.5, 0.3, 0.1, 1.0);
```

W tym przypadku użyliśmy funkcji `glUniform4f()`, gdyż nasza zmienna `uniform` jest typu `vec4`, czyli wektor czterowymiarowy. Kolejnymi parametrami tej funkcji są wartości składowe wektora czterowymiarowego.

Nie napisałem jeszcze w sumie ważnej rzeczy. Funkcje `glUniform*` musimy używać już po wywołaniu funkcji `glUseProgram()` uaktywniającej dany program shadera. Wywołanie ich przed tą funkcją nic by nie spowodowało. Nie mamy przecież jeszcze żadnego aktywnego programu shadera.

W wyniku otrzymamy na ekranie trójkąt, którego wszystkie wierzchołki będą pokolorowane na ten sam kolor.

Jeszcze kilka uwag na koniec. Wywołanie funkcji `glGetUniformLocation()` jest dość kosztowne. Nie należy jej zatem wywoływać w pętli rysującej. Najlepiej na początku przeprowadzić sobie inicjalizację oraz pobrać i zapisać lokalizację zmiennych `uniform` w

programie shadera. Potem wystarczy, że będziemy już tylko zmieniać wartości tych zmiennych za pomocą rodziny funkcji `glUniform*`(). Ponadto zawsze warto sprawdzać wartość zwracaną przez funkcję `glGetUniformLocation()`. W przypadku, gdy podana w drugim parametrze zmienna nie zostanie odnaleziona, funkcja zwróci wartość -1, np. w przypadku literówki. Możemy to więc łatwo wykryć.

Kosztowne jest także wywołanie jakieś funkcji z rodziny `glUniform*`(). Najlepszym rozwiązaniem jest po prostu używanie jej tylko w momentach, gdy chcemy wpisać inną wartość do zmiennej `uniform`. Wywoływanie jej w kółko pętli z tą samą wartością jest krótko mówiąc bez sensu.

Ostatnią kwestią, którą poruszę jest wartość początkowa zmiennych `uniform`. Wszystkie zmienne `uniform` są początkowo inicjalizowane wartością równą 0. Nie musimy zatem ich już zerować.

Tips & Tricks:

- Funkcja `glGetUniformLocation()` nie odnajdzie w programie shadera podanej zmiennej `uniform`, która jest tylko zdefiniowana, ale nie jest nigdzie używana.
- Użycie funkcji `glGetUniformLocation()` w czasie działania programu jest kosztowne. Należy użyć jej jeszcze przed pętlą renderowania i zapisać w zmiennych lokalizacje żądanych zmiennych `uniform` w programie shadera.

Kod źródłowy z tego rozdziału jest w katalogu: 8_Uniform

16. Winding order oraz face culling

Przy okazji lekcji o pierwszym trójkącie mieliśmy już do czynienia z tym tajemniczym jak na razie zagadnieniem. Pamiętacie na pewno jak wspomniałem o podawaniu wierzchołków w kierunku zgodnym bądź przeciwnym do ruchu wskazówek zegara. Jest to właśnie tak zwany **winding order**. W tym rozdziale zajmiemy się bardziej tą kwestią, gdyż jest również bardzo ważna.

Trójkąty są prymitywami. To z nich powstają później skomplikowane modele graficzne. Trójkąt jest figurą dwuwymiarową. Nie ma on żadnej głębi. Ma tylko dwie strony. Możemy go narysować na kartce papieru używając tylko trzech linii, a potem sobie go wyciąć.

Załóżmy teraz, że narysowaliśmy mnóstwo takich trójkątów pod różnymi kątami do siebie. Powstała nam w wyniku tej czynności jakaś bryła trójwymiarowa, np. model kubka z uchem. Otrzymaliśmy też ogromną ilość wierzchołków, bo "mnóstwo trójkątów" razy 3 = "ogromna ilość wierzchołków". Może trochę przesadziłem, ale przyjmijmy taką wizję do rozważań. Teraz wszystkie te wierzchołki musimy wyrenderować, tzn. dla każdego z nich obliczyć położenie, kolor, oświetlenie, itd. Ustawiliśmy nasz kubek blisko nas (kamery) tak, że widzimy tylko jego ograniczony kawałek. Po co więc tracić moc obliczeniową i rysować wierzchołki, które i tak nie są widoczne? Bez sensu.

W tym celu wprowadzono właśnie technikę optymalizacyjną nazwaną **face culling**. Polega ona na tym, że OpenGL może odrzucić te niewidoczne oraz skierowane do nas "plecami" trójkąty tworzące kompletny model trójwymiarowy. Zredukowana zostaje w związku z tym także liczba użytych vertex shader oraz fragment shader obciążających GPU.

Co do tych "pleców" trójkąta. Jeśli chcemy narysować na ekranie trójkąt, który ma się obracać to wtedy oczywiście nie chcemy, aby OpenGL nie rysował nam tylnej strony trójkąta. Chcemy wtedy rysować dwie strony, bo tak to przez 180 stopni obrotu nie mielibyśmy nic na ekranie. Jeśli natomiast z trójkątów złożymy sobie sześcian i będzie się on obracał, to wtedy nie chcemy rysować trójkątów zwróconych do nas plecami i tracić mocy obliczeniowej.

Jeśli jeszcze tego nie rozumiecie, to poniższy rysunek (rys. 16.1) powinien rozwiać wszelkie wątpliwości. Prezentuje on rzut z góry na scenę. Kolorem zielonym oznaczony jest przednia strona ścian (trójkątów), a kolorem czerwonym tylna.

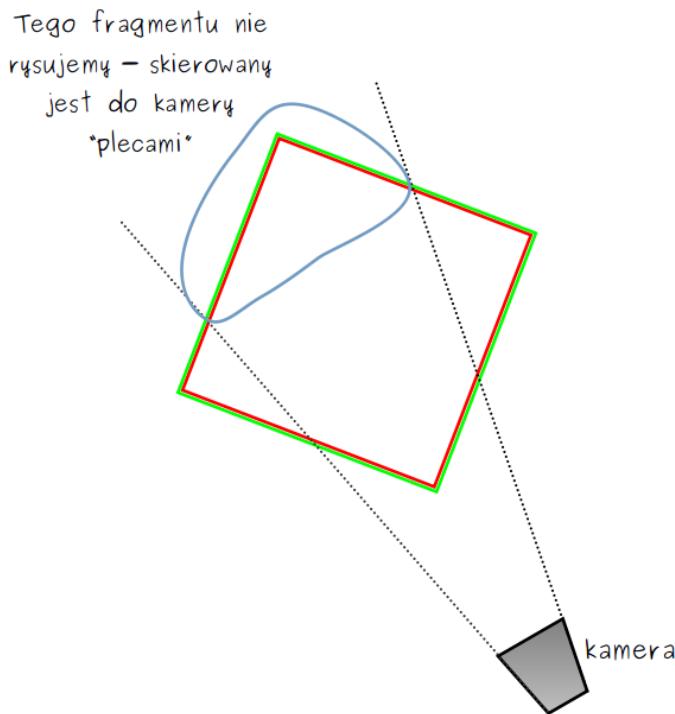
Aby w OpenGL ustawić, która strona trójkąta będzie przednia należy użyć funkcji `glFrontFace()`. Jako parametr przyjmuje ona jedną z dwóch wartości: `GL_CCW` albo `GL_CW`. `GL_CCW` oznacza, że stroną przednią będzie ta strona trójkąta, której wierzchołki podano w kierunku przeciwnym do ruchu wskazówek zegara. `GL_CW` oznacza natomiast kierunek zgodny z ruchem wskazówek zegara.

```
glFrontFace(GL_CCW);
```

Włączenie optymalizacji face culling sprowadza się do użycia dwóch funkcji:

```
glEnable(GL_CULL_FACE);  
glCullFace(GL_BACK);
```

Pierwszą funkcję już znamy. Włącza ona w maszynie stan pewien podany stan. Tutaj żądamy włączenia face culling. Drugą funkcją natomiast określamy, którą stronę chcemy pomijać przy rysowaniu. My życzymy sobie pomijania strony tylnej, dlatego jej parametrem wywołania jest `GL_BACK`. Można także podać `GL_FRONT` dla pomijania ściany przedniej.



Rys. 16.1. Prezentacja procesu face culling.

17. Macierze transformacji

Ten czas musiał kiedyś niestety nadjeść. To jest raczej nudny, ale taki "must-know" rozdział. Zajmiemy się trochę matematyką. Ale spokojnie. Postaram się wyjaśnić wszystko najprościej jak się da i w takim stopniu nieprzekraczającym naszych potrzeb. Niby wektory i macierze powinny być poruszone już w liceum, ale to chyba zależy od profilu danej klasy. Gębiej w ten temat wchodzi się dopiero na studiach. Przyznam, że miałem dylemat, czy po cichu założyć, że każdy już ma pewne pojęcie na ten temat i nie pisać wszystkiego, czy pisać praktyczne od zera. Jednak z tego, co widzę w internecie, coraz młodsze osoby pochłania pasja programowania, a więc stwierdziłem, że napiszę prawie od zera.

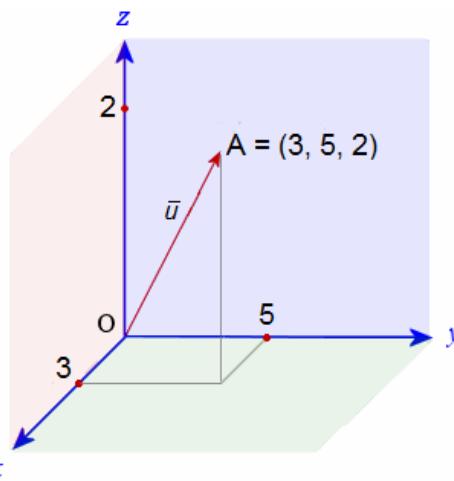
Wektory

Jak wiemy, położenie punktu w przestrzeni dwuwymiarowej bądź trójwymiarowej możemy opisać za pomocą współrzędnych. W przypadku przestrzeni dwuwymiarowej są to dwie współrzędne X i Y, natomiast w przypadku przestrzeni trójwymiarowej mamy trzy współrzędne X, Y oraz Z.

Załóżmy, że mamy punkt A o współrzędnych (3, 5, 2). Założymy też, że stoiemy w początku układu współrzędnych, czyli w punkcie (0, 0, 0). Możemy powiedzieć sobie, że musimy przejść o 3 jednostki wzdłuż osi X, o 5 jednostek wzdłuż osi Y oraz o 2 jednostki wzdłuż osi Z, aby dojść do tego punktu. To jest właśnie wektor. To przesunięcie z punktu zerowego do punktu A możemy zapisać właśnie za pomocą wektora:

$$\vec{u} = [3, 5, 2]$$

Wektory oznaczamy rysując małą strzałkę nad nazwą wektora i zapisując składowe wektora w nawiasach kwadratowych. Rysunek poniżej prezentuje to jaśniejs (rys. 17.1).



Rys. 17.1. Określenie położenia punktu w przestrzeni za pomocą wektora.

Widzimy, że graficznie wektor przedstawiamy jako strzałkę. W tym przypadku nasz wektor zaczepiony jest w punkcie (0, 0, 0), czyli przesuwamy się z tego właśnie punktu o podane wartości. Ale możemy przyjąć sobie dowolny inny punkt zaczepienia. Jeśli zaczepimy się w punkcie (4, 1, 2) to wtedy nasz wektor będzie miał następujące składowe: [-1, 4, 0].

Jeśli mamy dwa punkty, np. $A = (4, 2, 6)$ i $B = (1, 4, 9)$ oraz chcielibyśmy zdefiniować wektor w punkcie początkowym A i punkcie końcowym B, to postępujemy według wzoru:

$$A = (x_A, y_A, z_A)$$

$$B = (x_B, y_B, z_B)$$

$$\overrightarrow{AB} = [x_B - x_A, y_B - y_A, z_B - z_A]$$

W wyniku otrzymujemy wektor o składowych $[-3, 2, 3]$.

Wektor charakteryzowany jest także poprzez jego długość.

$$A = (x_A, y_A, z_A)$$

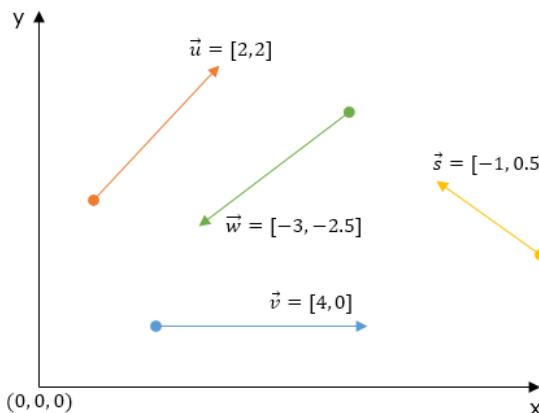
$$B = (x_B, y_B, z_B)$$

$$\overrightarrow{AB} = [x_B - x_A, y_B - y_A, z_B - z_A]$$

$$|\overrightarrow{AB}| = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

Nazwa wektora ujęta w dwie pionowe kreski oznacza długość wektora.

Wektory służą nie tylko do określania przesunięcia. Wykorzystywane są także do określania pewnego kierunku. Kierunek wektora wyznaczany jest przez prostą, na której on leży. Grot strzałki określa natomiast zwrot wektora (rys. 17.2).



Rys. 17.2. Wektory wskazujące różne kierunki i zwroty.

Widzimy, że te wektory zaczepione są w różnych punktach, a strzałka wektora wskazuje pewien zwrot wektora zależny od składowych tego wektora.

Macierze transformacji

Macierzą nazywamy pełną uporządkowaną tablicę danych. Dane umieszczone są w niej w poziomych wierszach oraz pionowych kolumnach. Przykładem macierzy jest na przykład następująca macierz:

$$\begin{bmatrix} 3 & 6 & 1 \\ 1 & 2 & 3 \\ 8 & 9 & 4 \end{bmatrix}$$

Jest to macierz o wymiarach 3x3. W pierwszej kolejności podaje się liczbę wierszy, a następnie liczbę kolumn.

Macierze będziemy wykorzystywać do przeprowadzania operacji transformacji na obiektach. Do tych operacji możemy zaliczyć translację (przesunięcie), skalowanie oraz rotację. Dla każdej z tych operacji definiujemy inną macierz. Będzie oddzielna macierz odpowiedzialna za translację, oddzielna odpowiedzialna za skalowanie oraz oddzielna odpowiedzialna za rotację. Wygodnie byłoby, jeśli mielibyśmy tylko jedną macierz, która by zawierała w sobie już wszystkie transformacje, a nie trzy oddzielne macierze.

Okazało się niestety, że nie możliwe jest opisanie wszystkich przekształceń w przestrzeni trójwymiarowej za pomocą jednej macierzy 3x3. Należy przyjąć przestrzeń o 1 większą, czyli należy stworzyć macierz o wymiarach 4x4. W tym celu wprowadzono właśnie **współrzędne jednorodne (homogeniczne)**. Wektory trójwymiarowe, czyli posiadające współrzędne XYZ, będąmy opisywać za pomocą czterech współrzędnych XYZW. Tyle abstrakcji nam wystarczy, dalej nie będziemy wnikać.

A o co chodzi z tą czwartą współrzędną W? Stanowi ona jakby dzielnik współrzędnych X, Y oraz Z. Jeśli współrzędna W jest różna od zera, czyli, np. ma wartość równą 1, to dzielimy przez nią pozostałe współrzędne. W wyniku otrzymujemy zmodyfikowane współrzędne punktu, ale nadal jest to punkt. Natomiast, jeśli współrzędna W ma wartość równą 0, to dzielić przez zero nie możemy. Możemy w uproszczeniu powiedzieć, że otrzymujemy wtedy prostą wyznaczającą pewien kierunek.

Jak wspomniałem, macierze transformacji będąmy wykorzystywać do różnych operacji przekształcających. Przekształcać będziemy wektor, np. wektor opisujący położenie jakiegoś wierzchołka. Przekształcenia dokonuje się poprzez pomnożenie macierzy transformacji i tego wektora.

UWAGA: kolejność w przypadku tego mnożenia ma znaczenie. Kolejność jest następująca: macierz transformacji x wektor wierzchołka = przetransformowany wierzchołek

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

Ogólnie zasada mnożenia jest prosta. Mnożymy wszystkie poszczególne wiersze macierzy przez składniki tego wektora. Tak naprawdę ten wektor tutaj jest traktowany jako macierz o wymiarach 4x1, gdyż inaczej nie można mnożyć. Ale to tylko taka dygresja.

Jeśli mamy macierz o wymiarach 4x4 to nasz wektor także musi mieć cztery składniki. Mnożenie macierzy 4x4 przez wektor trójwymiarowy nie jest możliwe.

Do pełni szczęścia potrzebne jest jeszcze jedno pojęcie. Jest nim macierz jednostkowa. **Macierzą jednostkową** nazywamy taką macierz, która na przekątnej ma same jedynki.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Już wiemy wszystko, co nam potrzeba odnośnie macierzy, możemy zająć się teraz operacjami transformacji.

1. Translacja

Jest to po prostu przesunięcie wzdłuż osi X, Y oraz Z. Macierz translacji wygląda następująco:

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

W miejscu wartości X, Y oraz Z podajemy żądane wartości przesunięcia wzdłuż poszczególnych osi.

Przykład

Mamy punkt o współrzędnych (6, 9, 3) i chcemy go przesunąć o wektor [2, 4, 0].

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 6 \\ 9 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 6 + 0 \cdot 9 + 0 \cdot 3 + 2 \cdot 1 \\ 0 \cdot 6 + 1 \cdot 9 + 0 \cdot 3 + 4 \cdot 1 \\ 0 \cdot 6 + 0 \cdot 9 + 1 \cdot 3 + 0 \cdot 1 \\ 0 \cdot 6 + 0 \cdot 9 + 0 \cdot 3 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \\ 3 \\ 1 \end{bmatrix}$$

W wyniku otrzymaliśmy punkt o współrzędnych (8, 13, 3), czyli wszystko się zgadza.

2. Skalowanie

Niezależne skalowanie względem każdej osi. Ogólny wzór macierzy skalowania:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Na przekątnej macierzy podajemy współczynniki skalowania dla poszczególnych osi.

Przykład

Skalujemy punkt (6, 9, 3) względem osi X oraz względem osi Z:

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 6 \\ 9 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 6 + 0 \cdot 9 + 0 \cdot 3 + 0 \cdot 1 \\ 0 \cdot 6 + 1 \cdot 9 + 0 \cdot 3 + 0 \cdot 1 \\ 0 \cdot 6 + 0 \cdot 9 + 2 \cdot 3 + 0 \cdot 1 \\ 0 \cdot 6 + 0 \cdot 9 + 0 \cdot 3 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 18 \\ 9 \\ 6 \\ 1 \end{bmatrix}$$

Widzimy, że otrzymaliśmy punkt, którego poszczególne współrzędne są przemnożone przez podane współczynniki.

3. Rotacja

Obroty względem poszczególnych osi, tzn. X, Y oraz Z. Ogólnie sama rotacja polega na tym samym co dwie operacje przedstawione już wyżej. Bardziej skomplikowane są tylko macierze transformacyjne. Spójrzmy na nie.

$$R_X(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Z(\gamma) = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 & 0 \\ \sin \gamma & \cos \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pierwsza macierz reprezentuje obrót wokół osi X o kąt alfa, druga obrót wokół osi Y o kąt beta, a trzecia obrót wokół osi Z o kąt gamma.

Przykładu w tym przypadku nie prezentuje, gdyż nie będzie według mnie tak obrazowy jak w przypadkach poprzednich. Poza tym, mnożyć macierz przez wektor już potrafimy.

Transformacje składane

Mogemy kilka transformacji połączyć w jedną. W wyniku otrzymamy złożoną macierz transformacyjną.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Punkt o współrzędnych (x, y, z) zostanie w pierwszej kolejności poddany operacji rotacji, a następnie translacji. Kolejność zapisu macierzy może być myląca, gdyż kolejność od lewej do prawej sugerowałaby, że w pierwszej kolejności zostanie wykonana operacja translacji, a dopiero potem rotacji. Należy patrzeć zawsze od wektora wejściowego, który jest poddawany przekształceniom i należy pisać operacje przekształcenia od prawej do lewej.

Kolejność mnożenia ma znaczenie. Wszystkie operacje transformacji odbywają się według punktu zaczepienia danego obiektu. A więc, transformacja złożona w pierwszej kolejności z rotacji, a później translacji jest inną transformacją złożoną w pierwszej kolejności z translacji, a później rotacji. W pierwszym przypadku obracamy się na początku w miejscu, a dopiero potem przesuwamy. Natomiast w drugim przypadku obracamy się wokół punktu zaczepienia po okręgu o promieniu wartości tego przesunięcia.

Ogólnie do naszych zastosowań najczęściej będziemy potrzebowali takiej kolejności:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = [T_M] \cdot [R_M] \cdot [S_M] \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Operacja skalowania jest pierwsza, później następuje rotacja, a dopiero później translacja. Czyli pierw ustalamy rozmiar obiektu, obracamy go pod żądanym kierunkiem, a potem przesuwamy na nowe miejsce.

18. Transformacje

Teorię odnośnie transformacji mamy już za sobą. Teraz się trochę pobawimy. Skorzystamy z biblioteki matematycznej **GLM**.

Zacznijmy od translacji. Na początku zdefiniujmy sobie macierz translacji korzystając z biblioteki GLM.

```
glm::mat4 trans(glm::translate(glm::mat4(1.0), glm::vec3(0.5, 0.3, 0.0)));
```

Zdefiniowaliśmy tutaj macierz transformacji, a dokładniej translacji, o nazwie `trans`. Macierz tą inicjalizujemy macierzą translacji stworzoną przez funkcję `translate()`. Przyjmuje ona dwa parametry. Pierwszym parametrem jest wejściowa macierz transformacji, do której zostanie dodana operacja translacji. My nie mamy jeszcze żadnej macierzy transformacji, a zatem parametr ten inicjalizujemy macierzą jednostkową. Drugim parametrem jest wektor translacji. Wektorem translacji jest wektor [0.5, 0.3, 0.0], czyli przesuniemy wierzchołki o 0.5 jednostki wzdłuż osi X oraz o 0.3 jednostki wzdłuż osi Y. Macierz transformacji jest oczywiście macierzą 4x4, a więc definiujemy ją jako `mat4`. Obiekt `mat4` reprezentuje bowiem macierz 4x4.

Teraz zmodyfikujmy vertex shader. Będziemy chcieli przesyłać do niego naszą macierz transformacji, a zatem musimy zdefiniować w nim zmienną typu `uniform`. Będzie to zmienna typu `mat4`, gdyż chcemy oczywiście przesyłać macierz o wymiarach 4x4.

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 vertex_colour;

uniform mat4 trans_matrix;

out vec3 colour;

void main()
{
    colour = vertex_colour;
    gl_Position = trans_matrix * vec4(position, 1.0);
}
```

Zwracam uwagę na kolejność mnożenia. Już o tym wspominałem w rozdziale poprzednim.

Skoro mamy już zdefiniowaną zmienną typu `uniform`, to teraz w kodzie głównym aplikacji musimy uzyskać jej lokalizację w programie shadera. Służy do tego funkcja `glGetUniformLocation()`. Robiliśmy już to parę rozdziałów wstecz.

```
GLuint shaders = LoadShaders("vertex shader.gls", "fragment shader.gls");
GLint trans_uniform = glGetUniformLocation(shaders, "trans_matrix");
if(trans_uniform == -1)
    std::cout << "Variable 'trans_matrix' not found." << std::endl;
```

Nie zostało nam już nic innego jak teraz przesłać zdefiniowaną wcześniej macierz transformacji do programu shadera. Tym razem użyjemy funkcji

`glUniformMatrix4fv()`, która umożliwia wpisanie do zmiennej `uniform` macierzy o wymiarach 4×4 .

```
glUniformMatrix4fv(trans_uniform, 1, GL_FALSE, glm::value_ptr(trans));
```

Funkcja ta przyjmuje cztery parametry. Pierwszym jest uzyskana lokalizacja zmiennej w programie shadera. W drugim parametrze podajemy liczbę macierzy, do których będziemy zapisywać. W naszym przypadku nie mamy żadnej tablicy macierzy w programie shadera, tylko jedną pojedynczą macierz, a zatem podajemy tutaj wartość równą 1. Trzeci parametr określa, czy wiersze macierzy mają zostać zamienione z kolumnami (transpozycja macierzy). Nie jest nam tutaj to potrzebne, dlatego ustawiamy na `GL_FALSE`. W czwartym parametrze podajemy dane, które chcemy wysłać, czyli naszą macierz transformacji. Funkcja `value_ptr()` przyjmuje obiekty typu `vec3`, `mat4`, itp. i zwraca wskaźnik do pamięci, gdzie są one zapisane. W taki sposób należy przekazywać obiekty biblioteki GLM do zmiennych typu `uniform`.

Jeśli teraz uruchomimy program, to nasz trójkąt powinien być przesunięty o podany wektor.

Zamiana operacji translacji na skalowanie bądź rotację już teraz jest dzieciinnie prosta. Wystarczy, że zmodyfikujemy macierz transformacji. I tak, dla skalowania mamy:

```
glm::mat4 scale_matrix(glm::scale(glm::mat4(1.0), glm::vec3(0.5, 0.5, 0.0)));
```

Jedyną różnicą w porównaniu z macierzą translacji jest tutaj użycie funkcji `scale()`, która utworzy macierz skalowania. Jako wektor skalowania podaliśmy wektor $[0.5, 0.5, 0.0]$, a zatem skalujemy względem osi X oraz Y. W wyniku otrzymamy trójkąt o połowie swojej wysokości i szerokości początkowej.

Z rotacją postępujemy identycznie. Macierz rotacji tworzona jest za pomocą funkcji `rotate()`. Przyjmuje ona trzy parametry. W pierwszym parametrze tak samo jak w funkcjach poprzednich podajemy wejściową macierz transformacji. Korzystamy z pustej macierzy transformacji, dlatego podajemy w tym parametrze macierz jednostkową. W drugim parametrze podajemy kąt obrotu, a w trzecim oś obrotu.

```
glm::mat4 rot(glm::rotate(glm::mat4(1.0), 45.f, glm::vec3(0, 1, 0)));
```

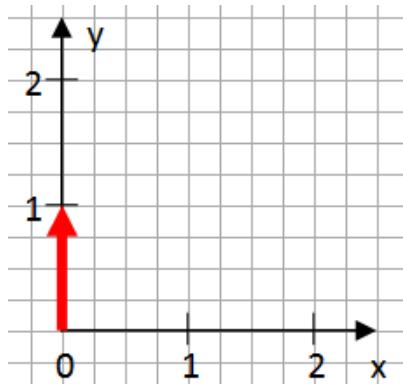
Kąt obrotu domyślnie podawany jest w stopniach. A więc powyższa macierz rotacji, obróci nam wierzchołki o kąt 45 stopni. Jeśli chcielibyśmy wymusić używanie radianów zamiast stopni, wystarczy, że przed dyrektywą `include` włączającą odpowiednie pliki nagłówkowe biblioteki GLM napiszemy taką definicję:

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
```

Wektorem rotacji jest tutaj wektor $[0, 1, 0]$. Jak wspomniałem w poprzednim rozdziale, wektory wskazują pewien kierunek. Jest to ta prosta, na której leży wektor (rys. 18.1).

Sytuacja jest taka sama jak na rysunku 18.1. Mamy wektor, który zaczepiony jest w punkcie $(0, 0, 0)$, a jego punktem końcowym jest punkt $(0, 1, 0)$. Wektor ten leży na osi Y,

a zatem możemy powiedzieć, że kierunkiem tego wektora jest os Y. Wracając do naszego kodu, zdefiniowana macierz rotacji obróci wierzchołki o kąt 45 stopni wokół osi Y. Jeśli zmienilibyśmy wektor na [1, 0, 0], to obrót nastąpiłby wokół osi X, a wektor [0, 0, 1] spowodowałby obrót wokół osi Z.



Rys. 18.1. Wektor wyznaczający kierunek osi Y.

Teraz zaszalejmy. Stworzymy pierwszą animację. Nasz kolorowy trójkąt będzie poruszał się po okręgu, a na dodatek będzie się obracał dookoła osi Y.

Na początku tworzymy sobie dwie macierze: macierz translacji oraz macierz rotacji. Musimy także zdefiniować kilka dodatkowych zmiennych.

```
glm::mat4 translate_matrix;
glm::mat4 rotate_matrix;

const float radius = 0.5;
float x_trans = 0.0;
float y_trans = 0.0;
float angle = 0;
```

Trójkąt będzie poruszał się po okręgu o promieniu 0.5, a zatem wartość tą definiujemy jako stałą. Wektor translacji będzie obliczany na bieżąco, a składowe tego wektora będą przechowywane w zmiennych `x_trans` oraz `y_trans`. Zmienna `angle` będzie przechowywała aktualny kąt obrotu trójkąta. Poniższy kod umieszczony jest już w głównej pętli rysującej.

```
static double prev_time = glfwGetTime();
double actual_time = glfwGetTime();
double elapsed_time = actual_time - prev_time;

if(elapsed_time > 0.01)
{
    prev_time = actual_time;
    angle += 1;

    x_trans = radius * cos(angle * M_PI / 180.0);
    y_trans = radius * sin(angle * M_PI / 180.0);
}
```

Animacja będzie sterowana czasem. Czas pobieramy z funkcji `glfwGetTime()`, która zwraca liczbę sekund, jaka upłynęła od chwili zainicjalizowania biblioteki GLFW. Pobieramy zatem aktualny czas i porównujemy go z czasem poprzednio pobrańym. Jeśli czas, który upłynął jest większy od jednej setnej sekundy, to wtedy aktualizujemy czas

poprzedni czasem aktualnym. Dodatkowo zwiększamy kąt obrotu oraz obliczamy nowe składowe wektora transformacji.

Teraz musimy obliczyć macierze transformacji i wysłać je do programu shadera.

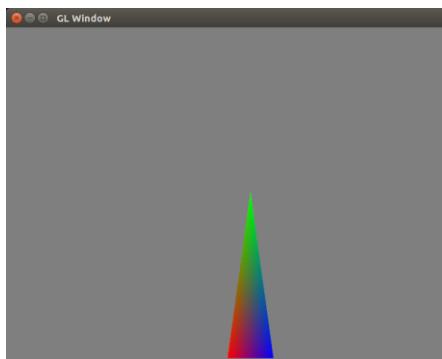
```
translate_matrix = glm::translate(glm::mat4(1.0), glm::vec3(x_trans, y_trans, 0.0));
rotate_matrix = glm::rotate(translate_matrix, angle, glm::vec3(0, 1, 0));
glUniformMatrix4fv(trans_uniform, 1, GL_FALSE, glm::value_ptr(rotate_matrix));
```

W pierwszej kolejności obliczamy macierz translacji. Zaczynamy od pustej macierzy (macierzy jednostkowej) oraz wpisujemy obliczony wcześniej wektor translacji. Następnie obliczamy macierz rotacji. Zauważcie, że tutaj w funkcji `rotate()` wykorzystujemy wyżej obliczoną macierz translacji. Chcemy bowiem nałożyć na siebie dwie transformacje. Wektorem rotacji jest wektor [0, 1, 0], a zatem jest to oś Y.

Na końcu wysyłamy obliczoną wynikową macierz transformacji do programu shadera.

Pamiętajcie o **wyłączeniu *cull face***, bo przez połowę obrotu nie będziecie mieli nic na ekranie.

Po uruchomieniu aplikacji powinien się naszym oczom ukazać wirujący oraz poruszający się po okręgu trójkąt (rys. 18.2).



Rys. 18.2. Wynik działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: **9_Transformacje**

19. Teksturowanie

Do tej pory nasze modele, a bardziej proste figury, kolorowaliśmy poprzez podanie koloru dla każdego wierzchołka. W wyniku interpolacji uzyskaliśmy ładne tęczowe powierzchnie. Było to fajne, ale czas na coś ciekawszego. Dziś zajmiemy się teksturami. Nauczymy się zatem, jak kolorować powierzchnie kolorami pochodząymi z pliku graficznego.

Na początku małe przypomnienie czym jest **piksel** i **fragment**. Pikselem jest najmniejsza jednostka ekranu. Kolor piksela ustalany jest poprzez fragmenty, czyli jednopikselsowe powierzchnie. W jednym miejscu ekranu może na siebie nachodzić kilka fragmentów, a zatem w tym przypadku kolor piksela zostanie obliczony jako pewna kombinacja kolorów nakładających się fragmentów.

Chcąc operować na teksturach musimy wprowadzić nowe pojęcie. Jest nim **teksel**. Teksem nazywamy najmniejszy element tekstury, czyli „piksel” tekstury. Założmy, że mamy teksturę o wymiarach 800x600 i pewną powierzchnię tej samej wielkości, która ma być pokryta tą teksturą. W wyniku na każdy fragment powierzchni przypadnie jeden teksel tekstury. Natomiast, jeśli przykładowo powierzchnia kryta tekturem jest mniejsza od tekstury, to na jeden fragment powierzchni przypadnie więcej niż jeden teksel (tekstura ściśnie się).

Tekstura nigdy nie jest dokładnie dopasowana do powierzchni, na której ma się znajdować. Może być większa, bądź mniejsza od powierzchni docelowej. Dlatego przeprowadza się operację mapowania, czyli dopasowania tekstury do powierzchni. Ogólnie proces ten nazywany jest **mipmappingiem**. Polega on na wygenerowaniu z wejściowej tekstury mniejszych tekstur, czyli przeprowadzeniu operacji skalowania. Mipmapping możliwy jest tylko wtedy, gdy tekstura wejściowa ma rozmiar będący potęgą dwójki, np. 256x256, 64x64. Przykładowo, mając teksturę o wymiarach 128x128, wygenerowane zostaną tekstury 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, 1x1, bo:

$$128 = 2^7 \quad 64 = 2^6 \quad 32 = 2^5 \quad 16 = 2^4 \quad 8 = 2^3 \quad 4 = 2^2 \quad 2 = 2^1 \quad 1 = 2^0$$

Celem mipmapping'u jest zwiększenie prędkości teksturowania, gdyż do bardziej odległych obiektów nie używamy wejściowej tekstury o dużej rozdzielczości. Obiekty odległe można wtedy teksturować mniejszymi teksturami. Mniejsze tekstury posiadają oczywiście mniej szczegółów, ale z daleka jest to i tak niewidoczne.

Tekstury są plikami graficznymi. Wybór formatu tekstur jest ogromny. Może to być PNG, BMP, JPEG, TGA, DDS, itp. Niestety OpenGL nie posiada sam w sobie funkcji ładowania plików graficznych. Musimy do tego użyć specjalnej zewnętrznej biblioteki. Do wyboru mamy kilka bibliotek w zależności od preferencji. Mój wybór padł na **FreeImage**. Wybrałem ją z tego względu, że jest ciągle wspierana i umożliwia obsługę wielu formatów. Instalacja jej jest prosta zarówno na Windows, jak i na Linuksie. Na Linuksie musimy sami sobie ją przebudować ze źródeł.

Niezależnie od tego, z której biblioteki korzystacie, procedura ładowania tekstuury i używania jej jest taka sama. Przedstawię wszystko w kilku krokach, aby łatwiej było zrozumieć.

Procedura ładowania i obsługi tekstuury

1. Ładowanie pliku graficznego - wczytanie do programu bajtów tworzących plik graficzny; proces wczytywania zależy od użytej biblioteki. W przypadku FreeImage, jest to coś takiego:

```
FREE_IMAGE_FORMAT image_format = FIF_UNKNOWN;
FIBITMAP* image_ptr = 0;
BYTE* bits = 0;

image_format = FreeImage_GetFileType(file_name.c_str(), 0);
if(image_format == FIF_UNKNOWN)
    image_format = FreeImage_GetFIFFromFilename(file_name.c_str());

if(FreeImage_FIFSupportsReading(image_format))
    image_ptr = FreeImage_Load(image_format, file_name.c_str());

bits = FreeImage_GetBits(image_ptr);

int image_width = 0;
int image_height = 0;
image_width = FreeImage_GetWidth(image_ptr);
image_height = FreeImage_GetHeight(image_ptr);
```

2. Określenie współrzędnych tekstuury - dla każdego wierzchołka podajemy współrzędne tekstuury, które mają mu zostać przypisane.

Tekstura tak samo jak ekran ma swoje współrzędne. Początek układu współrzędnych tekstuury znajduje się w **lewym dolnym rogu tekstuury**. Oś poziomą przyjęto nazywać się S, a oś pionową T (rys. 19.1).



Rys. 19.1. Układ współrzędnych tekstuury.

Jak już wyżej wspomniałem, współrzędne tekstuury określamy dla każdego wierzchołka. Spójrzmy na poniższy najprostszy przykład.

Założymy, że mamy w kodzie programu zdefiniowany prostokąt.

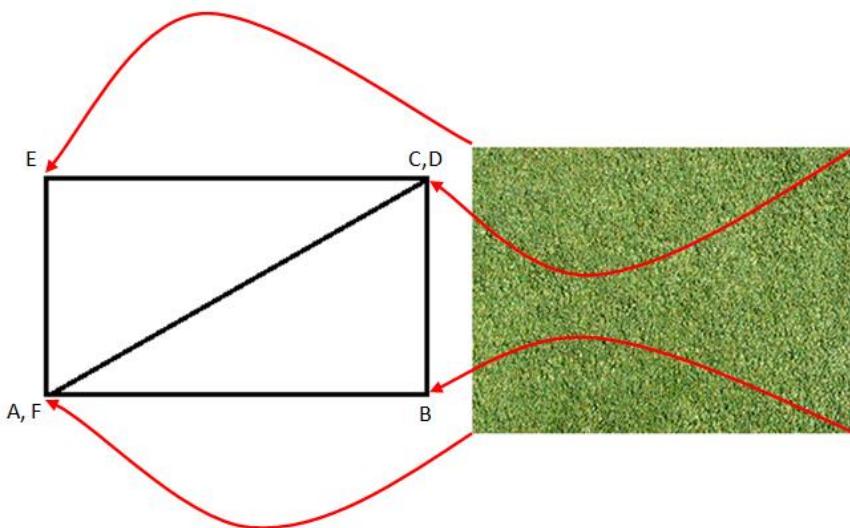
```
GLfloat points[] = {
```

```

-0.5f, -0.5f, 0.0f,
0.5f, -0.5f, 0.0f,
0.5f, 0.5f, 0.0f,
0.5f, 0.5f, 0.0f,
-0.5f, 0.5f, 0.0f,
-0.5f, -0.5f, 0.0f,
};


```

Chcielibyśmy teraz nałożyć na niego teksturę. Musimy zatem dla tych 6 wierzchołków (tworzących dwa trójkąty) podać współrzędne tekstury. Trójkąt dolny ma wierzchołki oznaczone A, B i C, a trójkąt górny wierzchołki oznaczone D, E, i F (rys. 19.2).



Rys. 19.2. Nakładanie tekstuury.

Łatwo możemy zauważyć jakie współrzędne tekstury powinien przyjąć każdy wierzchołek: A = (0,0), B = (1, 0), C = (1, 1), D = (1, 1), E = (0, 1), F = (0, 0).

Teraz musimy stworzyć tablicę, w której będziemy przechowywać te współrzędne.

```

GLfloat tex_coords[] = {
    0.0f, 0.0f, // A
    1.0f, 0.0f, // B
    1.0f, 1.0f, // C
    1.0f, 1.0f, // D
    0.0f, 1.0f, // E
    0.0f, 0.0f, // F
};


```

Każde dwa kolejne elementy tablicy odpowiadają kolejnemu wierzchołkowi.

3. Wpisanie współrzędnych tekstuury do nowego obiektu VBO - należy stworzyć nowy obiekt VBO i wpisać do niego tablicę współrzędnych wierzchołków.

```

GLuint texture_coords_vbo = 0;
 glGenBuffers(1, &texture_coords_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(tex_coords), tex_coords, GL_STATIC_DRAW);


```

Jak widać postępujemy tak samo jak w przypadku wpisywania tablicy współrzędnych wierzchołków, czy tablicy kolorów. Po prostu tworzymy nowy obiekt VBO.

4. Stworzenie obiektu VAO.

Tak samo jak w lekcjach poprzednich tworzymy obiekt VAO i wpisujemy do niego obiekty VBO. Pierwszy obiekt zawiera współrzędne wierzchołków, a drugi współrzędne tekstury. Koniecznie należy pamiętać o aktywacji tych dwóch atrybutów za pomocą funkcji glEnableVertexAttribArray().

```
GLuint vao = 0;
 glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);
 glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo);
 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);
 glEnableVertexAttribArray(1);
```

Zwróćcie natomiast uwagę na drugi parametr funkcji glVertexAttribPointer(), czyli tam, gdzie wpisujemy ile wartości ma być pobranych z tablicy na jeden wierzchołek. Teraz należy tutaj podać wartość 2, gdyż na każdy wierzchołek przypadają dwie wartości: współrzędna S oraz współrzędna T.

5. Stworzenie nowego obiektu tekstury i wpisanie do niego zawartości.

Spójrzmy pierw na kod, a potem omówię go krok po kroku.

```
GLuint texture = 0;
 glActiveTexture(GL_TEXTURE0);
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image_width, image_height, 0, GL_BGR, GL_UNSIGNED_BYTE,
 bits);
```

Na początku definiujemy zmienną typu GLuint, do której zostanie zapisany unikalny identyfikator nowo wygenerowanej tekstury.

Kolejnym krokiem jest wybór jednostki przechowującej teksturę, czyli takiej szufladki na nową teksturę. Służy do tego funkcja glActiveTexture(). Jako parametr przyjmuje ona wartość GL_TEXTURE#, gdzie # jest numerem jednostki. Domyślną wartością jest GL_TEXTURE0, czyli nie musielibyśmy tutaj tej funkcji umieszczać, ale chciałem pokazać jej użycie. Możemy mieć w programie dużo tekstur i chcemy je umieszczać w różnych szufladkach.

Dalszymi funkcjami, tzn. glGenTextures() i glBindTexture(), generujemy identyfikator nowej tekstury i ustawiamy ją jako aktywną. Chcemy, aby nowo wygenerowana tekstura była teksturą 2D, a zatem w funkcji glBindTexture() określamy to w pierwszym parametrze za pomocą parametru GL_TEXTURE_2D.

Najważniejszym etapem jest użycie funkcji glTexImage2D(). Przyjmuje ona 9 parametrów. Omówię je po kolej:

- typ tekstury; korzystamy z tekstury 2D, a zatem podajemy GL_TEXTURE_2D,
- wyjściowy poziom do mipmapping'u; wartość 0 oznacza, że jest to poziom zerowy, czyli poziom posiadający największą ilość szczegółów,
- liczba składowych koloru tekstury; chcemy przechowywać informację o czterech składowych: R, G i B, dlatego podajemy GL_RGB; kolejność składowych ma

znaczenie, tzn. można także podać GL_BGR; istnieje jeszcze dużo innych wartości, które możemy tutaj podać, ale na razie nie jest to nam potrzebne,

- szerokość tekstury,
- wysokość tekstury,
- parametr ten zawsze przyjmuje wartość 0,
- format koloru pliku wejściowego; jeśli plik wejściowy nie posiada zapisanej informacji o przezroczystości (kanale alfa) to stosujemy tutaj format GL_BGR, jeśli natomiast plik wejściowy posiada także informację o przezroczystości to musimy tutaj użyć formatu GL_BGRA.
- typ danych, w którym zapisana jest pobrana zawartość pliku graficznego; u nas jest to BYTE (unsigned long),
- wskaźnik do zawartości pliku graficznego.

6. Wywołanie dodatkowych funkcji obsługi tekstur.

My na razie korzystamy tylko z funkcji glGenerateMipmap(), która generuje mipmap'y dla aktywnej tekstury (tej ustawionej przez glBindTexture()). Nie wywołanie tej funkcji poskutkuje prawdopodobnie nie wyświetleniem się tekstury na ekranie, gdyż tekstura nie będzie mogła być dopasowana.

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Jako parametr przyjmuje ona rodzaj tekstury. My używamy tekstury dwuwymiarowej, a dlatego podajemy jej jako parametr wartość GL_TEXTURE_2D.

7. Modyfikacja vertex shader.

Vertex shader jest pierwszym elementem w potoku przetwarzania, a więc musi on odebrać współrzędne wierzchołków na ekranie oraz współrzędne tekstury i przekazać je dalej do fragment shader.

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;

out vec2 texture_coordinates;

void main()
{
    texture_coordinates = vt;
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
```

Zdefiniowaliśmy dwie zmienne wejściowe do shadera. Pierwszą jest zmienna typu vec3 i przyjmie ona współrzędne wierzchołków z warstwy 0, a drugą jest zmienna typu vec2 i odbierze ona współrzędne tekstury z warstwy numer 1. Zmienna wejściowa przyjmująca współrzędne jest typu vec2, gdyż odbiera ona dwie współrzędne S i T. Wartość "2" także ustalialiśmy w funkcji glVertexAttribPointer() dla współrzędnych tekstury. Tą zmienną wejściową zawierającą współrzędne tekstury przekazujemy dalej w głąb potoku przetwarzania do fragment shader za pomocą zmiennej typu out.

8. Modyfikacja fragment shader.

Fragment shader odbiera współrzędne tekstury oraz teksturę, a następnie wykonuje kolorowanie kolejnych fragmentów.

```
#version 330
in vec2 texture_coordinates;
uniform sampler2D basic_texture;
out vec4 frag_colour;

void main()
{
    vec4 texel = texture(basic_texture, texture_coordinates);
    frag_colour = texel;
}
```

Zmienną wejściową jest zmienna typu `vec2`, czyli są to współrzędne tekstury otrzymane z vertex shader, a zmienną wyjściową jest oczywiście `frag_colour`, czyli kolor danego fragmentu.

Zmienna `basic_texture` typu `uniform` służy do wyboru szufladki z tekstrą, którą chcemy użyć. Wystarczy, że w kodzie programu odwołamy się do tej zmiennej za pomocą:

```
GLuint shaders = LoadShaders("vertex_shader.gls1", "fragment_shader.gls1");
GLint texture_slot = glGetUniformLocation(shaders, "basic_texture");
glUniform1i(texture_slot, 0);
```

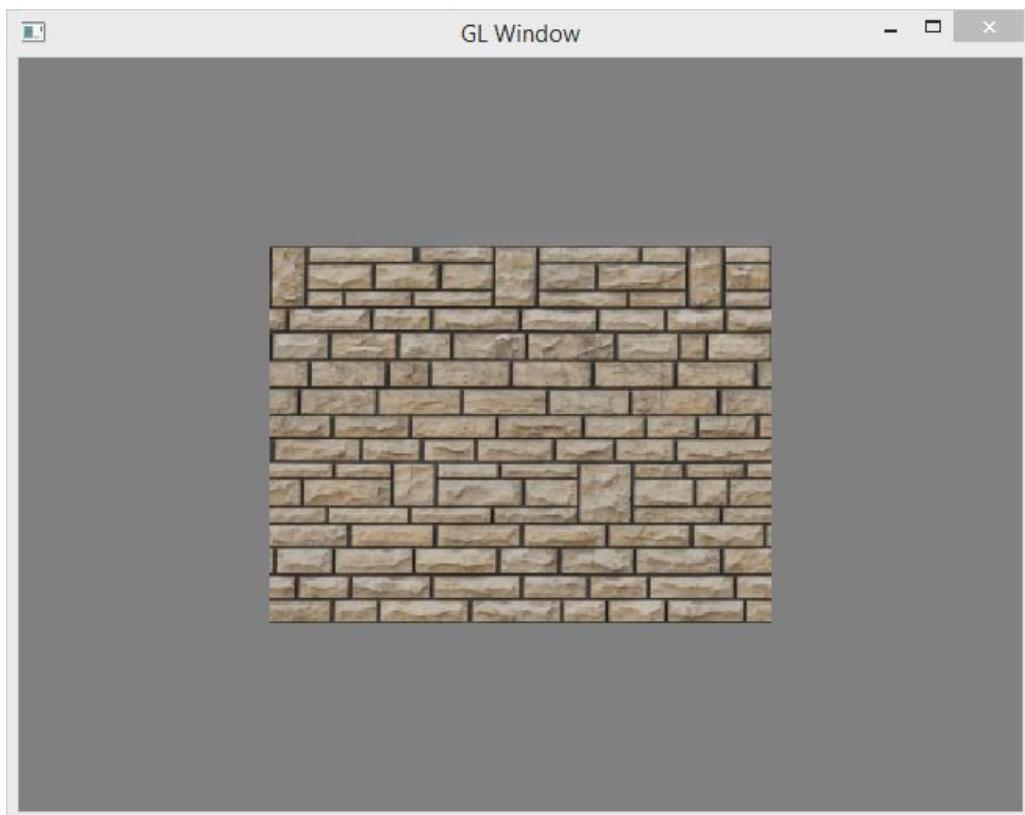
Odnajdujemy lokalizację zmiennej `basic_texture` w programie shadera, a następnie wysyłamy do niej wartość równą 0, czyli chcemy pobrać teksturę z jednostki z indeksem 0 (z szufladki o indeksie 0). Funkcja `glUniform1i()` jak nazwa wskazuje służy do wysłania do programu shadera jednej zmiennej typu `int` (całkowitego).

W przypadku nakładania tekstur mamy także do czynienia z interpolacją tak samo jak w przypadku kolorowania prostymi kolorami. Zauważcie, że podawaliśmy tylko współrzędne tekstury dla wierzchołków, a zatem cała ta przestrzeń pomiędzy wierzchołkami uzupełni się automatycznie pozostałymi współrzędnymi tekstury. Na przykład pomiędzy dwoma dolnymi wierzchołkami powstaną współrzędne tekstury (0.1, 0), (0.2, 0), (0.3, 0), itd. i tak aż do (1, 0).

Funkcja `texture()` użyta w fragment shader przyjmuje dwa parametry. Pierwszym jest zmienna `uniform` określająca aktywną teksturę, a drugim wejściowe współrzędne tekstury. Funkcja ta potrafi dopasować dla aktualnie przetwarzanego fragmentu odpowiedni teksel tekstury. W uproszczeniu zasada działania tej funkcji jest prosta. Jeśli mamy teksturę o wymiarach 800x600 i przykładowo współrzędne tekstury w danym fragmencie równe są (0, 1), to fragmentowi temu przypadnie teksel o współrzędnych (0 x 800, 1 x 600) = (0, 600). A jeśli przykładowo pewien fragment ma wynikające z interpolacji współrzędne tekstury (0.4, 0.7), to narysowany zostanie na nim teksel o współrzędnych (0.4 x 800, 0.7 x 600) = (320, 420).

Koniec procedury ładowania tekstury.

Po uruchomieniu programu zobaczymy efekt jak na rys. 19.3.



Rys. 19.3. Wynik działania programu.

Tips & Tricks:

- Możemy zabezpieczyć się przedomyłkowym przekroczeniem akceptowalnego przedziału wartości współrzędnych tekstury 0.0 - 1.0 poprzez użycie funkcji:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

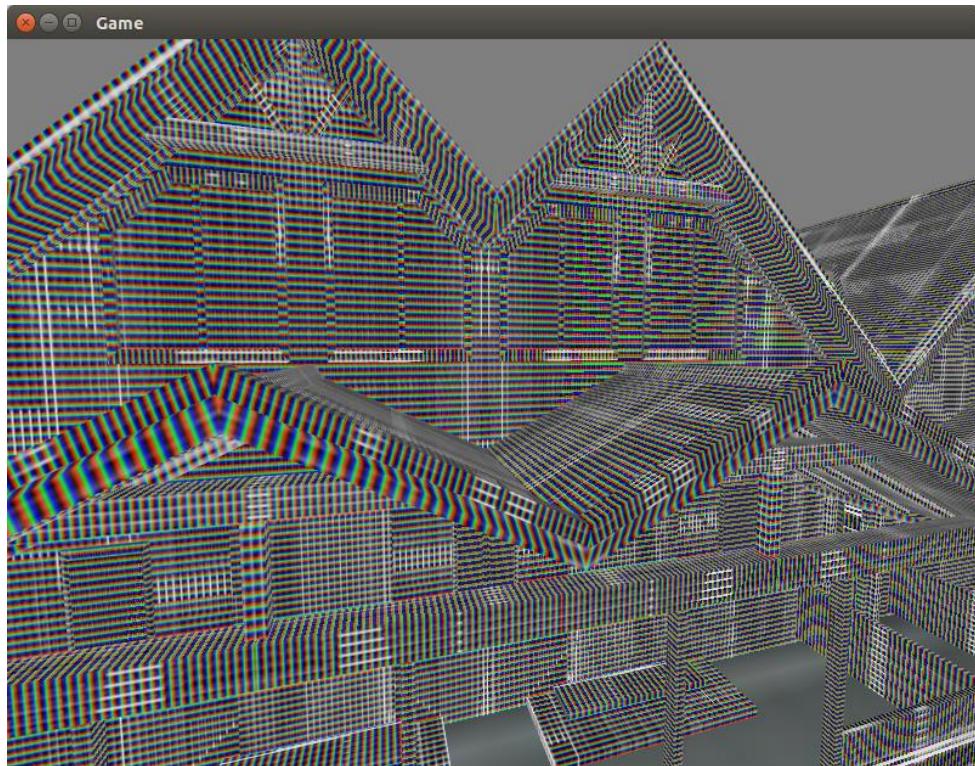
W przypadku, gdy podana wartość współrzędnej tekstury będzie za duża, ustawniona zostanie wartość maksymalna, czyli 1.0. Działanie tych dwóch funkcji odnosi się tylko do aktualnie aktywnej tekstury.

- Aby tekstura mogła być wykorzystana należy pierw umieścić ją w slocie. Wyboru slotu dokonujemy za pomocą funkcji `glActiveTexture()`. Jako parametr tej funkcji podajemy numer slotu, w którym chcemy umieścić teksturę. Należy pamiętać o tym, że mamy ograniczoną ilość slotów, w których możemy przechowywać tekstury. Ile jest tych slotów zależy od karty graficznej. Wartość tą można sprawdzić za pomocą funkcji `glGetIntegerv()` z parametrem `GL_MAX_DRAW_BUFFERS`. Jesteśmy zatem zmuszeni kontrolować ile aktualnie wykorzystujemy tekstur, ile zajmujemy slotów i w razie potrzeby zastępować tekstury w slotach. Najłatwiej jest oczywiście wykorzystywać tylko jeden slot i ciągle przeladowywać w nim teksturę. Nie jest to jednak rozwiązanie zbyt optymalne.

Kod źródłowy z tego rozdziału jest w katalogu: 10_Teksturowanie

20. Błędy odczytu tekstur

Jeśli wczytywałeś już jakieś tekstury i próbowałeś ich użyć, to może doświadczyłeś takich problemów jak na rys. 20.1.



Rys. 20.1. Błędnie wczytana tekstura.

Problem ten wynika z niewłaściwego ustawienia liczby bitów tekstuury przypadających na jeden piksel. To znaczy, jeśli dany plik graficzny zapisany jest tak, że jeden piksel opisany jest za pomocą 32 bitów (cztery kanały: R, G, B i A - każdy kanał ma 8 bitów), a my wczytując tekstuwę założymy 24-bitowy rozmiar piksela, to otrzymamy właśnie takie błędy jak na powyższym rysunku. Sytuacja odwrotna, tzn. wczytywanie 24-bitowej tekstuury, jako 32-bitową, również powoduje takie same problemy.

Rozwiązaniem tego problemu jest detekcja rozmiaru pojedynczego piksela w bitach. W bibliotece FreeImage służy do tego funkcja `FreeImage_GetBPP()`. Jako wynik swojego działania zwraca ona ilość bitów przypadających na jeden piksel tekstuury.

Proces wczytywania tekstuury powinien wtedy wyglądać mniej więcej tak:

```
FREE IMAGE FORMAT image format = FIF UNKNOWN;
FIBITMAP* image ptr = nullptr;
BYTE* bits = nullptr;

image_format = FreeImage_GetFileType(path.c_str(), 0);
if (image_format == FIF_UNKNOWN)
    image format = FreeImage_GetFIFFFromFilename(path.c_str());

if (FreeImage_FIFSupportsReading(image format))
    image_ptr = FreeImage_Load(image format, path.c_str());
```

```
bits = FreeImage_GetBits(image_ptr);

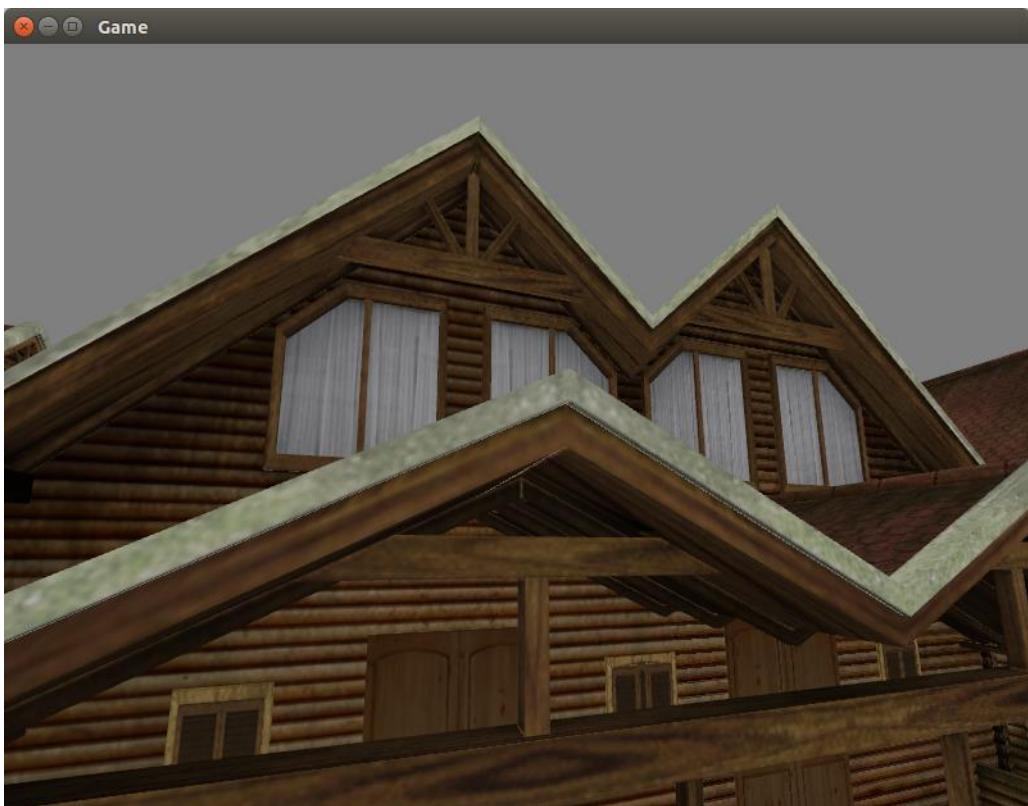
unsigned int image width = 0;
unsigned int image height = 0;
image_width = FreeImage_GetWidth(image_ptr);
image_height = FreeImage_GetHeight(image_ptr);

GLuint texture = 0;
glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);

unsigned int colours = FreeImage_GetBPP(image_ptr);
if (colours == 24)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image_width, image_height, 0,
                 GL_BGR, GL_UNSIGNED_BYTE, bits);
else if (colours == 32)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image_width, image_height,
                 0, GL_BGRA, GL_UNSIGNED_BYTE, bits);

glGenerateMipmap(GL_TEXTURE_2D);
FreeImage_Unload(image_ptr);
```

Jak widać, sprawdzamy tutaj ile bitów przypada na jeden piksel i w zależności od tego, używamy formatu `GL_BGR` lub `GL_BGRA`. Tekstura wtedy powinna wczytać się już poprawnie (rys. 20.2) bez znaczenia, czy jest to tekstura w formacie PNG, TGA czy JPEG.



Rys. 20.2 Poprawnie wczytana tekstura.

21. Licznik FPS

Renderując grafikę przydaje się mieć pewną miarę informującą nas o tym jak wydajnie renderujemy. Do tego celu można wykorzystać między innymi licznik ilości klatek renderowanych w ciągu sekundy (**FPS - Frames Per Second**).

Jak już wiemy, klatki renderujemy w pętli. W każdym obiegu pętli generowana jest nowa klatka obrazu. Im więcej klatek wygenerowanych w krótkim czasie, tym obraz wydaje się płynniejszy. Mała liczba klatek powoduje z kolei przeskoki obrazu, co jest strasznie męczące.

Gracze na pewno doskonale rozumieją to pojęcie. Grając w jakąś grę turową i posiadając małą liczbę klatek na sekundę, jesteśmy w dalszym stanie grać bez problemu. Problemy zaczynają się jeśli posiadamy niski wskaźnik FPS grając w dynamiczne gry. Brak płynności całkowicie uniemożliwia prowadzenie rozgrywki.

Skoro renderujemy w pętli, a więc tam musimy umieścić licznik klatek. Zaczniemy od prostej funkcji zliczającej klatki.

```
void FPSCounter(double& fps)
{
    static double prev_time = glfwGetTime();
    double actual_time = glfwGetTime();

    static int frames_counter = 0;

    double elapsed_time = actual_time - prev_time;
    if (elapsed_time >= 1.0)
    {
        prev_time = actual_time;

        fps = static_cast<double>(frames_counter) / elapsed_time;
        frames_counter = 0;
    }

    frames_counter++;
}
```

Funkcja ta jest naprawdę prosta. Korzystamy w niej z znanej już nam funkcji `glfwGetTime()` zwracającej liczbę sekund, która upłynęła od czasu inicjalizacji biblioteki GLFW. W naszej funkcji sprawdzamy zatem, jaki czas upłynął od ostatniej chwili. Jeśli ten czas jest równy bądź większy od jednej sekundy, to obliczamy liczbę klatek wygenerowanych w ciągu tego czasu. Jeśli z kolei jeszcze nie upłynęła sekunda to inkrementujemy dalej licznik klatek.

Teraz wystarczy, że wrzucimy tą funkcję do głównej pętli renderującej i już mamy działający licznik klatek. Nie umiemy jeszcze wyświetlać tekstu w OpenGL, a zatem liczbę klatek będziemy wyświetlać na górnej belce okna. Do zmiany nazwy okna służy funkcja `glfwSetWindowTitle()`. W pierwszym parametrze przyjmuje ona wskaźnik do okna, a w drugim nową nazwę okna.

```
while(!glfwWindowShouldClose(window))
{
    static double fps = 0;
    FPSCounter(fps);
```

```
    std::string title = "Game @ FPS: " + std::to_string(fps);
    glfwSetWindowTitle(window, title.c_str());
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Skorzystałem tutaj z mechanizmów C++11, czyli z funkcji `to_string()`. Zamianę liczby `double` na `string` można równie dobrze wykonać za pomocą funkcji `sprintf()`.

Podczas renderowania należy pamiętać o jednej kluczowej rzeczy. Każdy monitor ma swoją jakąś częstotliwość odświeżania, np. 75 Hz. Generowanie klatek z częstotliwością większą niż częstotliwość odświeżania monitora jest po prostu bez sensu, bo i tak monitor tego nie nadąży wyświetlać. Ponadto, w głównej pętli renderującej prowadzimy także inne obliczenia, takie jak sztuczna inteligencja. Obliczanie sztucznej inteligencji szybciej niż jesteśmy w stanie na to zareagować (bo monitor się nam wystarczająco szybko nie odświeża) też nie jest poprawnym podejściem. Można wykorzystać oczywiście synchronizację pionową (**VSync - Vertical Sync**), czyli pobranie w aplikacji częstotliwości odświeżania monitora i renderowania z taką samą częstotliwością.

Nie popadajmy teraz w manię optymalizacji kodu tak, aby wycisnąć jak najwięcej FPS. Można zmarnować na tym mnóstwo czasu, a w efekcie otrzymać dwie dodatkowe klatki. Poza tym istnieją lepsze narzędzia analizy wydajności niż licznik FPS, ale o tym jeszcze dopowiem.

Kod źródłowy z tego rozdziału jest w katalogu: 11_FPS

22. Kamera wirtualna

Wszystko to, co do tej pory działało się na naszej scenie, obserwowaliśmy tylko z jednego stałego punktu. Nie mieliśmy możliwości zmiany kąta patrzenia oraz poruszania się kamerą w głąb sceny. Poruszanie się po świecie wirtualnym jest przecież kluczowym elementem gier komputerowych. Czas zatem opuścić kurtynę i zobaczyć jak to wszystko jest zrealizowane.

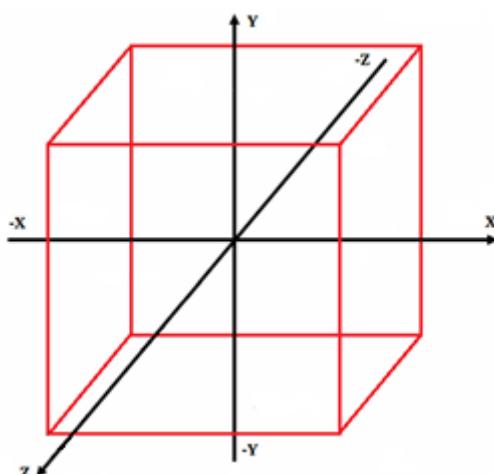
Rzutowanie

Najprostsza definicja rzutowania mówi, że jest to operacja polegająca na takim przedstawieniu pikseli na płaskim ekranie, aby sprawiały wrażenie trójwymiarowości (przestrzeni trójwymiarowej).

Wyróżniamy dwa rodzaje rzutowania:

- rzutowanie ortogonalne (prostokątne),
- rzutowanie perspektywiczne.

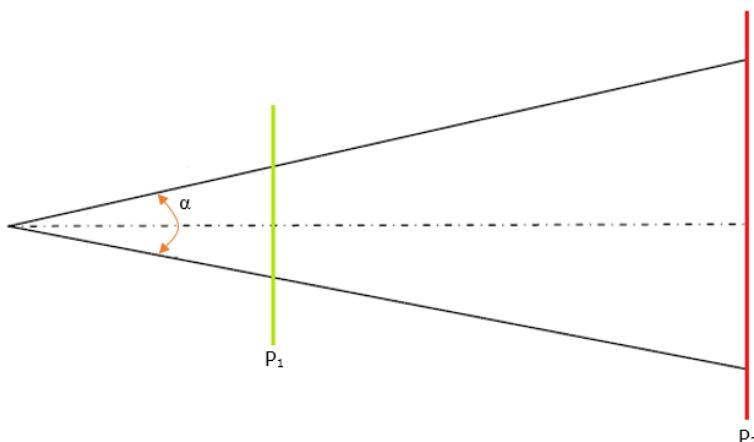
Począwszy od pierwszego rozdziału posługiwaliśmy się rzutowaniem ortogonalnym (rys. 22.1). W rzutowaniu tym bryłą widzenia jest prostopadłościan. Tylko obiekty znajdujące się w obrębie tej bryły zostaną narysowane na ekranie. Każdy obiekt znajdujący się w obszarze rysowania niezależnie od swojej odległości od kamery **ma dokładnie tą samą wielkość**. Ten rodzaj rzutowania wykorzystywany jest z reguły w grach dwuwymiarowych.



Rys. 22.1. Prostopadłościan widzenia w rzutowaniu ortogonalnym.

Dla nas bardziej naturalne jest rzutowanie perspektywiczne (rys. 22.2). Im obiekt znajduje się dalej od oka (kamery) tym jest mniejszy. Powstaje zatem wrażenie głębi. Rzutowanie to wykorzystujemy w grach trójwymiarowych.

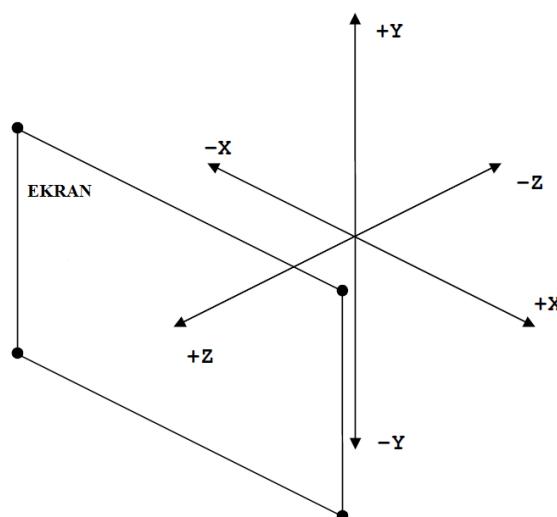
W widoku perspektywicznym występuje stożek widzenia. Stożek ten ograniczony jest przez dwie płaszczyzny: bliższą P_1 oraz dalszą P_2 . Wszystkie obiekty (wierzchołki) znajdujące się pomiędzy tymi dwoma płaszczyznami zostaną wyświetlane na ekranie, natomiast obiekty znajdujące się poza stożkiem nie są już uwzględniane przy rysowaniu. Kąt α zaznaczony na pomarańczowo określa nam kąt widzenia, tzw. FOV - Field Of View.



Rys. 22.2. Stożek widzenia w rzutowaniu perspektywicznym.

Kamera

Zadaniem kamery tak samo jak w rzeczywistości jest obserwowanie sceny. Kamera może mieć swoje własne położenie oraz orientację w przestrzeni. Obiekty, które obserwujemy okiem kamery znajdują się w pewnej odległości oraz orientacji względem niej. Aby wiedzieć, w jakiej odległości od kamery znajduje się jakiś obiekt przypisano kamerze jej własny układ współrzędnych. W OpenGL obowiązuje **prawoskrętny układ współrzędnych** (rys. 22.3).

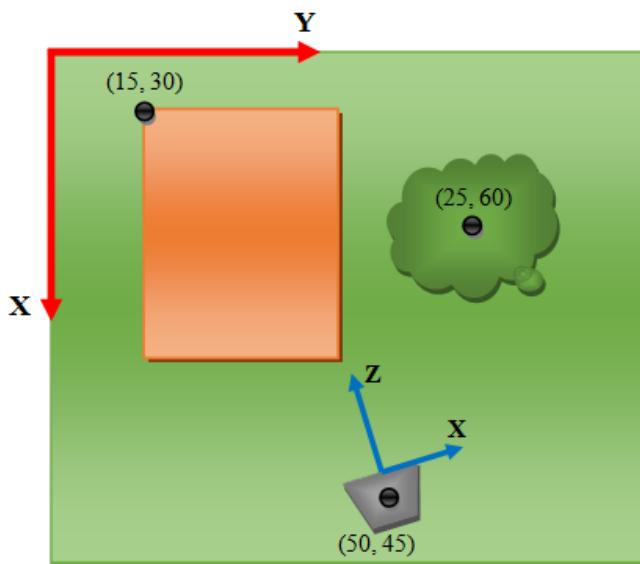


Rys. 22.3. Prawoskrętny układ współrzędnych.

Takim układem postugiwałyśmy się do tej pory. Umieszczony on był w środku ekranu. Definiując jakiś wierzchołek, definiowałyśmy go względem tego układu współrzędnych. Przykładowo wierzchołek o współrzędnych (0.5, 0.3) umieszczony był w prawej górnej ćwiartce ekranu, gdyż os X skierowana jest dodatnio w prawo, a os Y w góre.

Projektując mapę do gry komputerowej nie myśli się w ogóle o kamerze i jej układzie współrzędnym. Przyjmuje się w pewnym punkcie mapy, np. w lewym rogu mapy, początek układu współrzędnych mapy i później względem tego układu współrzędnych określa się położenie różnych obiektów takich jak budynki, drzewa, itp. na mapie. Na przykład zakładamy sobie, że magazyn broni umieszczamy w punkcie o współrzędnych (15, 30), a

drzewo przy tym magazynie w współrzędnych (25, 60). Gdy mapa jest już gotowa do przetestowania, umieszczana jest kamera w jakimś punkcie początkowym na mapie, np. (50, 45). Położenie oraz orientacja kamery wyrażone są zatem w układzie współrzędnym mapy. Załóżmy teraz, że patrzymy się kamerą na ten magazyn broni. Kamera ma swój układ współrzędnych (wychodzący jakby z obiektywu), a obiekty na mapie umieszczone są w układzie współrzędnym mapy. Trzeba zatem obliczyć jakie będzie położenie tych obserwowanych obiektów na mapie w układzie współrzędnym kamery, aby wiedzieć jak daleko je umieścić względem widza patrzącego przez tą kamerę (rys. 22.4). Aby takie obliczenia były możliwe należy stworzyć pewną macierz przekształcenia jednego układu współrzędnych do drugiego.



Rys. 22.4. Wzajemne położenie układu współrzędnych mapy i kamery.

Podsumowując, kamera jest takim wędrującym układem współrzędnych i musimy nauczyć się jak przeliczać współrzędne z pewnego globalnego układu współrzędnych do układu współrzędnych kamery.

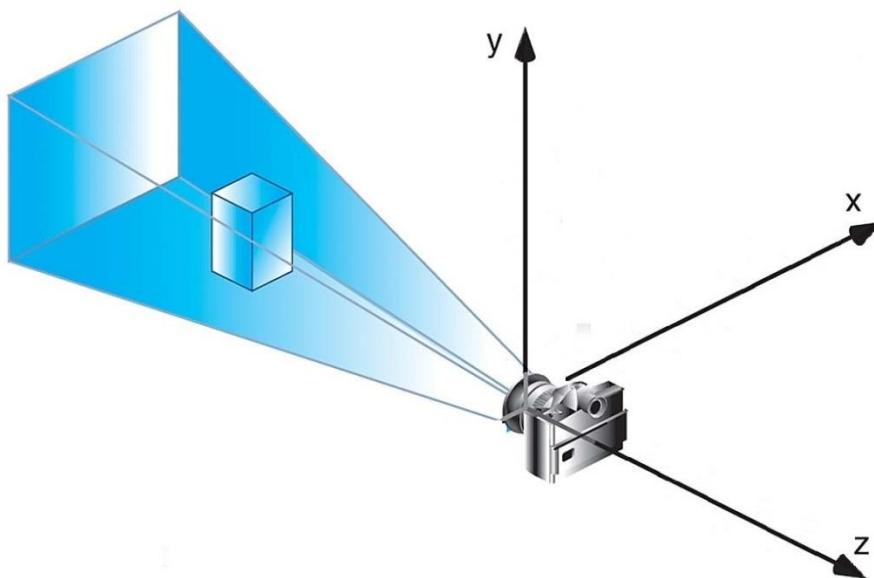
Macierz modelowania i macierz rzutowania (projekcji)

Jak już wspomniałem, musimy mieć pewną macierz przeliczającą nam współrzędne oraz orientacje obiektów na scenie na współrzędne układu kamery. Wykonywane jest to za pomocą macierzy kamery (widoku). Ogólna postać macierzy kamery jest następująca:

$$\begin{bmatrix} R_x & R_y & R_z & -P_x \\ U_x & U_y & U_z & -P_y \\ -F_x & -F_y & -F_z & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

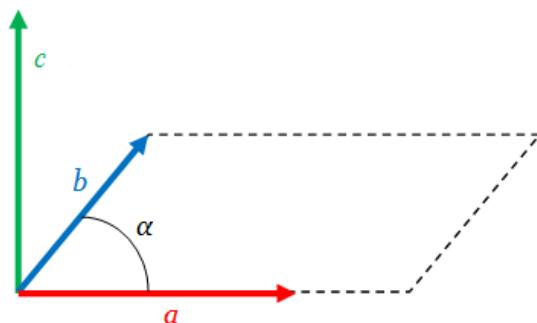
gdzie: R - wektor wskazujący w prawo, U - wektor wskazujący w góre, F - wektor wskazujący do przodu, P - wektor określający położenie kamery.

Wektory R , U oraz P są wektorami określającymi orientację (obrót) kamery. Kierunek wektora R określa kierunek osi X kamery, kierunek wektora U określa kierunek osi Y kamery, a kierunek wektora F określa kierunek osi Z kamery (rys. 22.5).



Rys. 22.5. Układ osi współrzędnych kamery.

Jak wiemy, trójwymiarowy układ współrzędnych tworzą trzy prostopadłe do siebie osie. Nie możemy zatem w pełni dowolnie określić kierunku tych wektorów. Zazwyczaj robi się to w taki sposób, że określa się kierunek dwóch wektorów (dwóch osi), a później oblicza się na tej podstawie kierunek trzeciego wektora (osi). Aby obliczyć wektor prostopadły do dwóch podanych należy skorzystać z **iloczynu wektorowego** (rys. 22.6).

Rys. 22.6. Iloczyn wektorowy. Wektor c jest prostopadły do płaszczyzny wyznaczonej przez wektory a i b .

Nie będę pokazywał jak się go oblicza, gdyż nie jest to nam potrzebne. Zdefiniowane są już specjalne funkcje, które zrobią to za nas.

Na stworzenie macierzy kamery są trzy metody. Pokażę je wszystkie, każdy może wybrać sobie swoją ulubioną. Oczywiście wykorzystamy bibliotekę matematyczną GLM.

Metoda 1:

Definiujemy dwa wektory wskazujące kierunek, nie ma znaczenia które, np. wektor wskazujący do przodu i wektor wskazujący w prawą stronę. Trzeci wektor obliczamy jako iloczyn wektorowy dwóch poprzednich.

```
glm::vec3 direction(0.0, 0.0, -1.0);
glm::vec3 right(1.0, 0.0, 0.0);
glm::vec3 up = glm::cross(right, direction);
```

Dalej definiujemy wektor translacji, który przesunie kamerę w odpowiednie miejsce na mapie (scenie).

```
glm::vec3 camera_pos(0.0, 0.0, 2.0);
```

Ostatecznie tworzymy macierz kamery według podanego wyżej wzoru macierzy. Jeśli odnosimy się do macierzy w bibliotece GLM poprzez operator [] to wpisujemy wtedy kolumnę, a nie wiersz. Należy o tym pamiętać.

```
glm::mat4 view_matrix;
view_matrix[0] = glm::vec4(right.x, up.x, -direction.x, 0);
view_matrix[1] = glm::vec4(right.y, up.y, -direction.y, 0);
view_matrix[2] = glm::vec4(right.z, up.z, -direction.z, 0);
view_matrix[3] = glm::vec4(-camera_pos.x, -camera_pos.y, -camera_pos.z, 1);
```

Metoda 2:

Macierz kamery możemy także zdefiniować jako operację rotacji oraz translacji. Zdefiniujmy zatem te dwie macierze.

```
glm::mat4 t = glm::translate(glm::mat4(1.0), -camera_pos);
glm::mat4 r = glm::rotate(glm::mat4(1.0), 0.0f, glm::vec3(0, 1, 0));
```

Aby teraz złożyć te dwie operacje do jednej macierzy transformacji należy je przez siebie przemnożyć. Kolejność mnożenia ma tutaj duże znaczenie. Musimy w pierwszej kolejności wykonać operację rotacji, a dopiero później translacji.

```
glm::mat4 view_matrix = r * t;
```

Metoda 3:

W celu zdefiniowania macierzy kamery (widoku) można także skorzystać z funkcji zaimplementowanej w bibliotece GLM. Jest to funkcja `lookAt()`. Przyjmuje ona trzy parametry:

- wektor określający położenie kamery,
- kierunek patrzenia,
- wektor określający kierunek "w górę".

```
glm::mat4 view_matrix = glm::lookAt(camera_pos, camera_pos + direction, up);
```

Poznaliśmy trzy metody na stworzenie macierzy kamery. Niedawno też zajmowaliśmy się macierzą przekształceń, która obracała, skalowała oraz przesuwała wierzchołki. Otóż, ta macierz przekształceń oraz przed chwilą poznana macierz kamery tworzą razem **macierz modelowania**.

Teraz zajmijmy się macierzą rzutowania (projekcji). Jak sama nazwa wskazuje, macierz ta pozwala nam określić, w jaki sposób chcemy rzutować. Macierz ta składa się z następujących elementów:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & P_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$D = \tan(0.5 \cdot FOV) \cdot P_1$$

$$S_x = (2 \cdot P_1) / 2 \cdot D \cdot A$$

$$S_y = P_1 / D$$

$$S_z = -(P_2 + P_1) / (P_2 - P_1)$$

$$P_z = -(2 \cdot P_2 \cdot P_1) / (P_2 - P_1)$$

gdzie: FOV - kąt widzenia, $P1$ - odległość płaszczyzny $P1$ od kamery (rys. 22.2), $P2$ - odległość płaszczyzny $P2$ od kamery, A - aspekt ekranu, np. 16:9 lub 4:3.

Za pomocą biblioteki GLM mamy dwie możliwości na stworzenie macierzy rzutowania.

Metoda 1:

W tej metodzie obliczamy ręcznie wszystkie składniki, a następnie tworzymy z nich macierz projekcji.

```
float P1 = 0.1f;
float P2 = 100.0f;
float FOV = 90.0f;
float aspect = 16.0f / 9.0f;

float D = tan(0.5 * FOV * 3.14 / 180.0) * P1;
float Sx = (2.0 * P1) / (2 * D * aspect);
float Sy = P1 / D;
float Sz = -(P2 + P1) / (P2 - P1);
float Pz = -(2.0f * P2 * P1) / (P2 - P1);

glm::mat4 perspective;
perspective[0] = glm::vec4(Sx, 0, 0, 0);
perspective[1] = glm::vec4(0, Sy, 0, 0);
perspective[2] = glm::vec4(0, 0, Sz, -1.0);
perspective[3] = glm::vec4(0, 0, Pz, 0);
```

Metoda 2:

Drugą metodą na stworzenie macierzy projekcji jest skorzystanie z funkcji `perspective()` pochodzącej z biblioteki GLM. Pobiera ona cztery parametry:

- kąt widzenia (FOV),
- aspekt ekranu,
- położenie płaszczyzny $P1$,
- położenie płaszczyzny $P2$.

Jak widać pobiera ona takie same parametry, jakie musielibyśmy podać w metodzie powyżej. Zaletą tej metody jest właśnie to, że nie musimy tych składników macierzy obliczać ręcznie.

```
glm::mat4 perspective = glm::perspective(FOV, aspect, P1, P2);
```

Gdy już zdefiniowaliśmy nasze macierze musimy je przesłać do programu shadera. W pierwszej kolejności jednak trzeba zmodyfikować vertex shader, aby mógł przyjmować poprzez zmienne uniform te dwie macierze. Modyfikacja nie jest trudna. Wystarczy dodać dwie zmienne typu uniform oraz dwie dodatkowe operacje mnożenia macierzy. Przemnażanie przez zmienną `trans_matrix` nie jest konieczne, jeśli nie chcemy w jakiś sposób modyfikować wierzchołków naszego obiektu. Można ten składnik pominąć.

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;

uniform mat4 trans_matrix;
uniform mat4 view_matrix;
uniform mat4 perspective matrix;

out vec2 texture coordinates;

void main()
{
    texture coordinates = vt;
    gl_Position = perspective matrix * view matrix * trans matrix * vec4(position, 1.0);
}
```

Należy koniecznie pamiętać o **odpowiedniej kolejności mnożenia macierzy**. Pierw wierzchołki poddawane są przekształceniom takim jak rotacja, skalowanie oraz translacja. Później następuje obliczenie współrzędnych wierzchołków w układzie wspólczesnym kamery za pomocą macierzy kamery. Na końcu dopiero wprowadzane zostaje przekształcenie pochodzące z macierzy projekcji.

Kod źródłowy z tego rozdziału jest w katalogu: 12_Kamera_wirtualna

23. Klawiatura i mysz

Nauczyliśmy się już, czym jest kamera wirtualna oraz w jaki sposób ustawia się ją na scenie. Przypomnę jeszcze, że do ustawienia kamery w żądanym miejscu na scenie trzeba podać cztery wektory. Przede wszystkim jest to wektor translacji przesuwający kamerę do danego punktu. Dodatkowo, musimy zdefiniować jeszcze trzy wektory określające orientację kamery, czyli wektory określające kierunki (zwroty) osi układu współrzędnych kamery. Jeśli już to wszystko wiemy, to nie będzie problemem teraz wykorzystać klawiaturę i myszkę w celu modyfikacji tych wektorów.

Zacznijmy od klawiatury. Wykorzystywana będzie ona do przesuwania kamery. Zdefiniujmy sobie na początku zmienne, w których będziemy przechowywać wektory opisujące położenie oraz orientację kamery. Trzy z nich od razu możemy zainicjalizować.

```
glm::vec3 camera_position(0.0, 0.0, 2.0);
glm::vec3 camera_direction(0.0, 0.0, -1.0);
glm::vec3 camera_right(1.0, 0.0, 0.0);
glm::vec3 camera_up;
```

Teraz musimy wrócić do funkcji przechwytyjącej naciśnięcia klawiszy klawiatury (definiowaliśmy ją już w pierwszym rozdziale). W jej ciele dodamy obsługę klawiszy W, S, A oraz D, które będą odpowiadały za zmianę wartości odpowiednich wektorów, a tym samym za przesuwanie kamery na mapie. Standardowo, klawisz W będzie przesuwał kamerę do przodu, klawisz S do tyłu, klawisz A w lewo, a klawisz D w prawo.

Ogólnie idea jest prosta. Jeśli poruszamy się do przodu bądź do tyłu, to poruszamy się wzdłuż osi Z kamery, czyli **wzdłuż kierunku wektora wskazującego kierunek patrzenia kamery**. Jeśli natomiast poruszamy się w prawo lub lewo, to poruszamy się **wzdłuż wektora określającego kierunek „w prawo”**. Jest to oczywiście oś X kamery.

A więc, można napisać taki kawałek kodu:

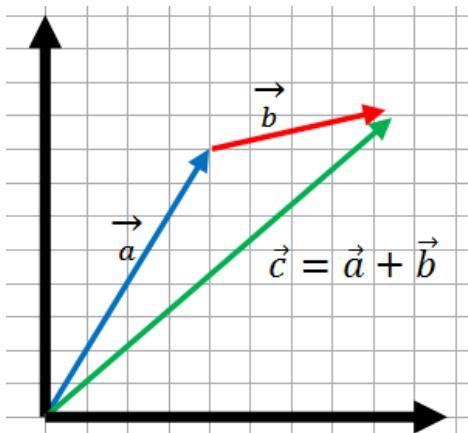
```
// globalnie
glm::mat4 view_matrix;

// ...
float move_speed = 3.0f;

if (key == GLFW_KEY_W)
    camera_position += camera_direction * (float)GetTimeDelta() * move_speed;
if (key == GLFW_KEY_S)
    camera_position -= camera_direction * (float)GetTimeDelta() * move_speed;
if (key == GLFW_KEY_A)
    camera_position -= camera_right * (float)GetTimeDelta() * move_speed;
if (key == GLFW_KEY_D)
    camera_position += camera_right * (float)GetTimeDelta() * move_speed;

view_matrix = glm::lookAt(camera_position, camera_position + camera_direction, camera_up);
```

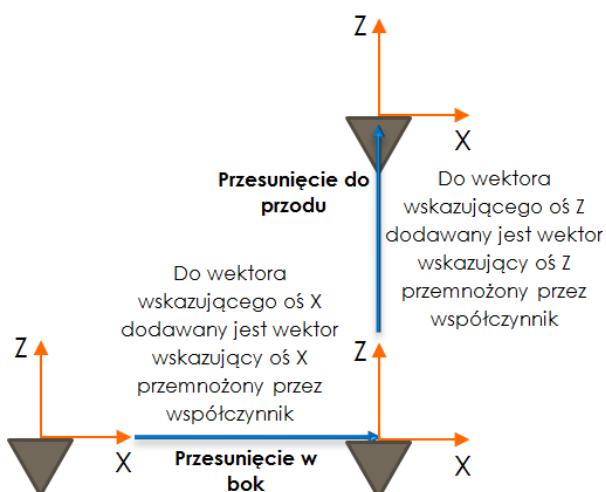
Mamy tutaj cztery instrukcje warunkowe sprawdzające, który klawisz został naciśnięty. W przypadku klawiszy W oraz S, do aktualnej pozycji kamery (wektora pozycji kamery) dodawany lub odejmowany jest wektor kierunku kamery. A zatem mamy tutaj do czynienia z dodawaniem dwóch wektorów. Ogólny schemat dodawania wektorów prezentuje rys. 23.1.



Rys. 23.1. Dodawanie dwóch wektorów. Wektor „a” jest wektorem określającym aktualne położenie kamery, a wektor „b” jest wektorem przesunięcia do nowego miejsca. Wektor „c” określa końcowe (nowe) położenie kamery.

Identycznie sprawa wygląda w przypadku przesuwania kamery w bok, tylko z tą różnicą, że do wektora aktualnego położenia dodawany jest wektor określający kierunek osi X kamery.

Całość może podsumować rys. 23.2.



Rys. 23.2. Przesuwanie kamery.

Wektor przesunięcia przemnażany jest przez pewien współczynnik. Mnożenie wektora przez **dodatni współczynnik** nie zmienia jego zwrotu ani kierunku. Zmienia tylko jego długość. Każda składowa wektora (a są one 3) przemnażana jest przez ten współczynnik. Dzięki temu mamy możliwość regulacji skokowości przesuwania kamery. Jeśli długość wektora przesunięcia będzie mała, to ruchy kamery będą wydawać się płynne. Jeśli natomiast długość tego wektora będzie zbyt duża, obserwowalne będą skokowe ruchy kamery.

Zmienna `move_speed` jest zwykłą zmienną typu `float`. Służy do sterowania długością ruchu. Jest to zwykły współczynnik, przez który przemnażamy wektor.

Dodatkowym współczynnikiem jest funkcja `GetTimeDelta()`. W każdym obiekcie pętli aktualizowane są dwie zmienne:

```
// globalne
double actual_time;
double previous_time;

// wewnatrz petli while
previous_time = actual_time;
actual_time = glfwGetTime();
```

Zmienna `actual_time` przechowuje czas, jaki jest przy wejściu do aktualnego obiegu pętli, a zmienna `previous_time` przechowuje czas, jaki był w poprzednim obiegu pętli. Zadaniem funkcji `GetTimeDelta()` jest obliczenie różnicy pomiędzy tymi dwoma czasami:

```
double GetTimeDelta()
{
    return actual_time - previous_time;
}
```

A zatem, funkcja ta, zwraca czas, jaki upłynął od poprzednio wyrenderowanej klatki obrazu. Przez ten czas przemnażamy także nasz wektor kierunku, jest to więc dodatkowy współczynnik. Musimy skorzystać z tej funkcji, gdyż niektóre komputery są szybsze, a niektóre wolniejsze, a co za tym idzie, na szybszych komputerach przez jedną sekundę trzymania klawisza poruszania, przebędziemy większą odległość, niż na tym wolniejszym. Szybszy komputer w ciągu jednej sekundy wygeneruje na przykład 50 klatek obrazu, a wolniejszy tylko 10. Czyli przez jedną sekundę na szybszym komputerze przemieścilibyśmy się pięć razy dalej, niż na wolniejszym. Wprowadzamy zatem pewne ograniczenie, które ujednolici w pewien sposób przesuwanie kamery.

Po obliczeniu nowego położenia kamery, musimy na nowo przeliczyć macierz widoku. Korzystamy tutaj z dobrzej biblioteki GLM.

W przypadku myszki jest trochę trudniej, ale ogólnie zasada działania jest podobna. Skoro istnieje funkcja przechwytyująca klawisze klawiatury, to istnieje także funkcja przechwytyująca ruchy myszki. W funkcji tej musimy się zatroszczyć o takie elementy jak:

- ciągłe ustawianie kurSORA w środku okna aplikacji tak, aby kurSOR nie wyjechał poza brzegi okna - stracilibyśmy wtedy kontrolę nad kamerą,
- sprawdzanie aktualnego położenia kurSORA myszki - to, że wyżej ustawiamy go w środku ekranu, to nie znaczy, że jest on tam zabetonowany na stałe; jeśli poruszymy kursorem to po prostu powróci on zaraz do środka okna, ale ruch ten zostanie wykryty,
- na podstawie ruchów kurSORA myszki wokół punktu centralnego okna, należy wyznaczyć, w którą stronę kurSOR się przemieścił i obliczyć kąt obrotu,
- zaktualizować orientację kamery.

Kod tej funkcji jest następujący:

```
//globalnie
double horizontal_angle = 0;
double vertical_angle = 0;
```

```

void CursorPositionCallback(GLFWwindow* window, double x cursor pos, double y cursor pos)
{
    glfwSetCursorPos(window, window width / 2, window height / 2);
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

    float mouse speed = 0.5f;

    horizontal angle += mouse speed * (float)GetTimeDelta() *
                        float(window width / 2.0 - x cursor pos);
    vertical_angle += mouse speed * (float)GetTimeDelta() *
                      float(window height / 2.0 - y cursor pos);

    if (vertical angle < -1.57)
        vertical angle = -1.57;
    if (vertical angle > 1.57)
        vertical_angle = 1.57;

    camera direction = glm::vec3(cos(vertical angle) * sin(horizontal angle),
                                 sin(vertical angle),
                                 cos(vertical angle) * cos(horizontal angle));
    camera_right = glm::vec3(-cos(horizontal_angle), 0,
                            sin(horizontal_angle));
    camera_up = glm::cross(camera_right, camera_direction);

    view matrix = glm::lookAt(camera position, camera position +
                                camera direction, camera up);
}

```

Funkcja `glfwSetCursorPos()` służy do ustawiania pozycji kurSORA na ekranie. My ustawiamy go w środku naszego okna aplikacji. Zmienne `window_width` oraz `window_height` zawierają szerokość oraz wysokość okna.

Systemowy kurSOR na tle naszej graficznej aplikacji nie wygląda elegancko. Możemy go ukryć za pomocą funkcji `glfwSetInputMode()`. Parametrem `GLFW_CURSOR_DISABLED` określamy, że nie chcemy widzieć kurSORA na ekranie. Ale w sumie, na początku nie polecam go wyłączać, można zaobserwować jak wyglądają ruchy „zaczeplionej” myszki, żeby nie żyć w przekonaniu, że jest całkiem nieruchoma.

Następnie obliczamy nowe wartości kątów obrotu kamery. Sprawdzana jest różnica położenia kurSORA względem środka ekranu (czyli przesunięcie kurSORA), a następnie, wartość ta przemnażana przez dwa współczynniki. Ich przeznaczenie jest takie samo jak w przypadku klawiatury.

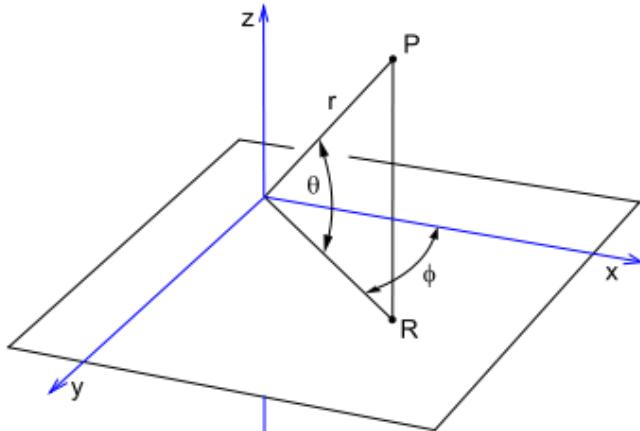
Po obliczeniu kąta pionowego, sprawdzamy czy mieści się on w przedziale -90 i $+90$ stopni. Zapobiegamy tym przekręcaniem się kamery dookoła własnej osi.

Pozostała jeszcze jedna rzecz do wyjaśnienia. Jak obliczyć wektory określające kierunki osi układu współrzędnych kamery na podstawie pionowego oraz poziomego kąta obrotu? Potrzebujemy trochę matematyki do tego. Te dwa kąty wystarczają już do tego, aby określić kierunek (i zwrot) danego wektora (rys. 23.3). Stanowią one, tzw. **współrzędne sferyczne**. W łatwy sposób można przejść z tych współrzędnych sferycznych do **współrzędnych kartezjańskich**, czyli do tych, w których używamy współrzędnych X, Y oraz Z do opisu położenia. Należy skorzystać z takich wzorów:

$$\begin{aligned}
 x &= r \cos \theta \cos \varphi \\
 y &= r \cos \theta \sin \varphi \\
 z &= r \sin \theta
 \end{aligned}$$

Wartość r nas nie interesuje, gdyż chodzi nam tylko o kierunek wektora. Nie interesuje nas to, jaki jest dlugi. Przyjmujemy zatem $r = 1$.

Odnosząc się do rys. 23.3, oś X na rysunku jest osią Z kamery (bo kąt poziomy jest mierzony od tej osi), oś Y na rysunku jest osią X kamery (bo leży na tej samej płaszczyźnie, co kąt poziomy), a oś Z jest osią Y kamery.



Rys. 23.3. Współrzędne sferyczne.

W ten sposób obliczymy kierunek wektora patrzenia kamery. Musimy jeszcze obliczyć wektor określający kierunek osi X lub Y, obojętnie. Nie zamierzamy obracać kamery dookoła osi Z, a więc wektor X będzie zawsze leżał na płaszczyźnie i będzie prostopadły do rzutu wektora Z na tą płaszczyznę. Aby wyznaczyć wektor prostopadły do wektora $[a, b]$ należy zmienić kolejność współrzędnych i dodać znak minus przy jednej z nich. Tak więc, do wektora $[a, b]$ są dwa wektory prostopadłe: $[-b, a]$ oraz $[b, -a]$.

```
camera_direction = glm::vec3(cos(vertical_angle) * sin(horizontal_angle),
                             sin(vertical_angle),
                             cos(vertical_angle) * cos(horizontal_angle));
camera_right = glm::vec3(-cos(horizontal_angle), 0,
                         sin(horizontal_angle));
camera_up = glm::cross(camera_right, camera_direction);
```

Wektor określający kierunek osi Y zostanie obliczony jako iloczyn wektorowy dwóch pozostałych wektorów.

Na sam koniec musimy jeszcze ustawić tą naszą funkcję obsługi ruchu myszy jako aktywną. Służy do tego funkcja `glfwSetCursorPosCallback()`.

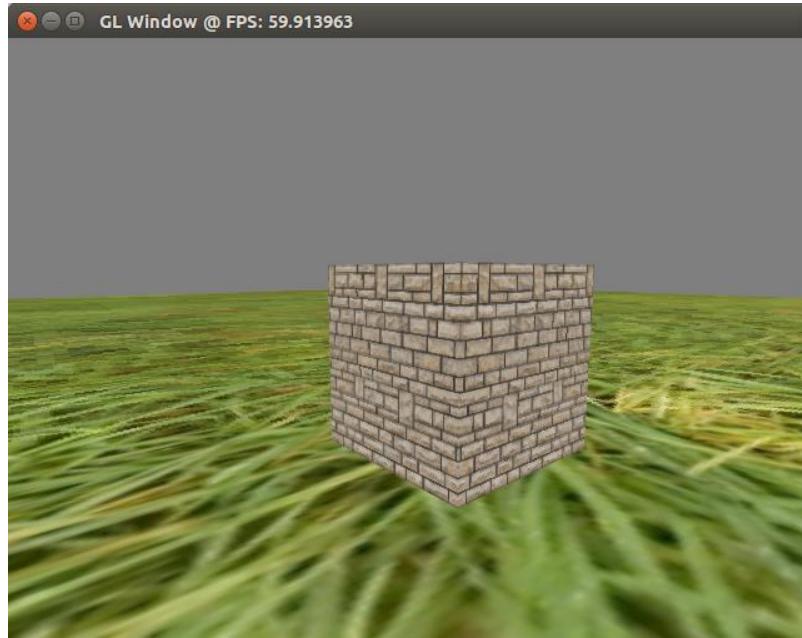
```
glfwSetCursorPosCallback(window, CursorPositionCallback);
```

Wszystko to nie zadziała, jeśli zapomnimy o **wysłaniu zaktualizowanej macierzy widoku do programu shadera**. Aktualną macierz widoku wysyłamy w pętli głównej programu:

```
glUniformMatrix4fv(view_uniform, 1, GL_FALSE, glm::value_ptr(view_matrix));
```

Polecam uruchomić kod źródłowy z tego rozdziału i przyjrzeć się dokładnie jak wszystko jest zrealizowane. Zachęcam do pobawienia się kodem, modyfikuj go śmiało. Pozwoli to zrozumieć większość zagadnień.

Po uruchomieniu programu otrzymamy taką scenę jak na rys. 24.4. Będziemy mogli się później dowolnie przemieszczać, jak w prawdziwej grze trójwymiarowej.



Rys. 23.4. Wynik działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: 13_Klawiatura_mysz

24. Filtrowanie tekstur

Z teksturami jest jeden zasadniczy problem. Ulegają zniekształceniu po nałożeniu na powierzchnię trójwymiarową. Jeśli uruchamiasz program przedstawiony w poprzednim rozdziale to na pewno zauważysz, że tekstury nie wyglądają za ładnie. Były one jakby rozmyte i niewyraźne. W tym rozdziale zajmiemy się właśnie filtrowaniem tekstur, aby uniknąć takich niemyłych dla oka efektów.

Podczas nakładania tekstur na powierzchnię tekstuра jest próbkowana (**sampling**). Procesem tym jest nic innego jak obliczanie koloru dla danego fragmentu. Nazywane jest to próbkowaniem, ponieważ pobierane są próbki obrazu (próbki koloru z tekstuры) i tym pobranym kolorem malowany jest dany fragment.

Jeśli tekstuра jest idealnie dopasowana do powierzchni to problem zniekształcenia nie wystąpi. Mamy na przykład prostokątną tekstuру o wymiarach 800x600 oraz prostokątną powierzchnię o takich samych wymiarach. W tej sytuacji każdy teksel tekstuры trafi do odpowiadającego sobie fragmentu.

Problem pojawia się natomiast, gdy tekstuра jest mniejsza bądź większa od krytej powierzchni. Nastąpi wtedy zwiększenie bądź zmniejszenie liczby teksli przypadających na jeden fragment. Jeśli tekstuра jest mniejsza od powierzchni, to jeden teksel tekstuры zostanie przypisany do kilku fragmentów (rozciagnięcie tekstuры - **magnification**). Zagęszczenie teksli tekstuры na jednym fragmencie następuje wtedy, gdy tekstuра jest większa od powierzchni (ściśnięcie tekstuры - **minification**).

Z tej racji, że prawie nigdy nie będziemy mieli tekstuры idealnie dopasowanej do powierzchni, musimy określić jak OpenGL ma ją filtrować. Filtrować, to znaczy wykorzystać pewien algorytm obliczania koloru, który zostanie przydzielony danemu fragmentowi. Rodzaj filtra należy określić dla dwóch przypadków: rozciagnięcia tekstuры oraz ściśnięcia tekstuры. Oczywiście nie filtrujemy tekstuры o rozmiarze identycznym z rozmiarem powierzchni, gdyż jest ona już odpowiednio dopasowana.

Są trzy główne rodzaje filtrowania tekstuры.

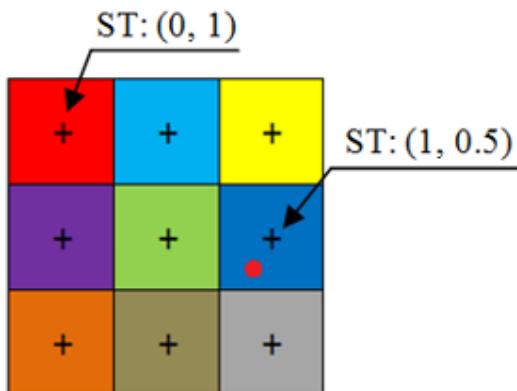
1. Filtrowanie najbliższym sąsiadem.

Jest to najprostszy rodzaj filtra. Pobierany jest teksel znajdujący się najbliżej obliczonej wartości. Jeśli na przykład jakiemuś fragmentowi przypadają współrzędne tekstuры ST: (0.8, 0.6), a tekstuра ta ma wymiary 3x3, to można obliczyć, który teksel przypadnie temu fragmentowi: $(0.8 \cdot 3, 0.6 \cdot 3)$ (rys. 24.1).

Jest to najszybszy filtr nie powodujący obciążenia GPU, gdyż nie wymaga on przeprowadzania żadnych trudnych obliczeń. Niestety, uzyskane efekty nie są najlepsze. Powstaje poszarpany, z widocznymi pikselami, obraz.

Do określenia rodzaju filtrowania w kodzie służy funkcja `glTexParameter()`. Używa się jej do ustawiania parametrów aktywnej tekstuры. Przyjmuje ona trzy parametry:

- rodzaj tekstury,
- nazwa parametru - my będziemy na razie wykorzystywać dwa parametry: `GL_TEXTURE_MAG_FILTER` i `GL_TEXTURE_MIN_FILTER`. Pierwszy parametr określa, że chcemy ustawić filtr w przypadku powiększenia tekstury, a drugi w przypadku ścisnięcia tekstury.
- nowa wartość parametru.



Rys. 24.1. Filtrowanie najbliższym sąsiadem. Czerwona kropka oznacza obliczone współrzędne (2.4, 1.8). Wybrany zostanie zatem ten niebieski teksel, na którym leży kropka. Jest on najbliższy.

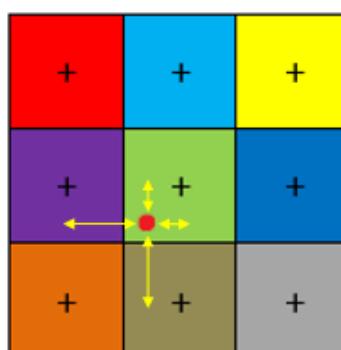
W kodzie programu filtr ten definiujemy następująco:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Wartość `GL_NEAREST` określa, że chcemy skorzystać z filtrowania najbliższym sąsiadem.

2. Filtrowanie liniowe.

W tym rodzaju filtrowania pobrany teksel ma kolor powstały w wyniku zmiksowania kolorów czterech sąsiadujących tekslów. Podczas mikowania kolorów uwzględniana jest odległość do każdego z sąsiadujących tekslów (filtrowanie dwuliniowe) (rys. 24.2).



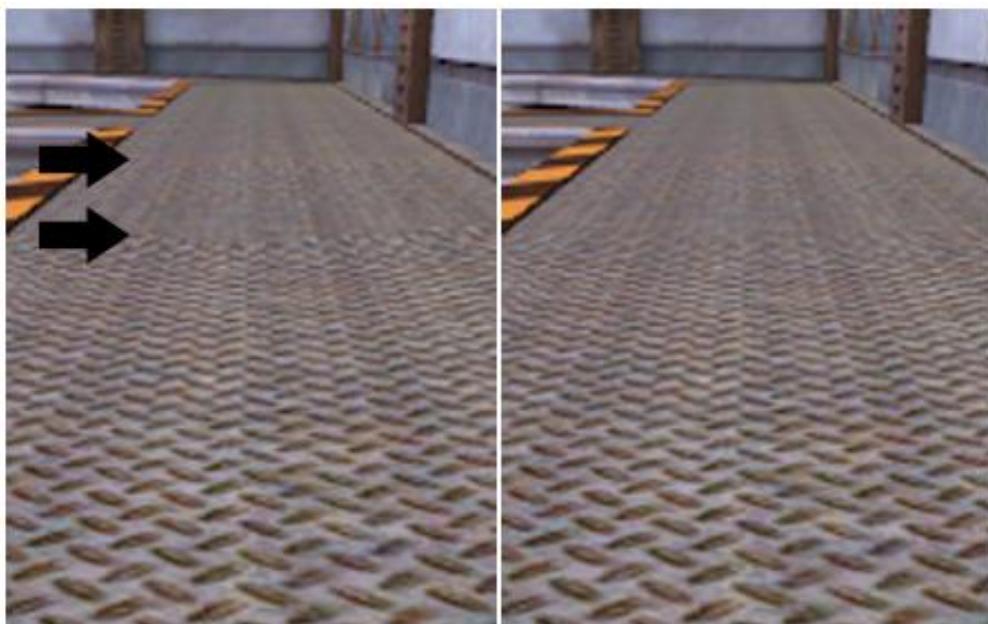
Rys. 24.2. Filtrowanie liniowe. Żółte strzałki prezentują odległość do czterech sąsiadnych tekslów od obliczonego punktu.

W kodzie ten filtr definiujemy następująco:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Wróćmy jeszcze do zagadnienia mipmap. Skrót **MIP** rozwija się do „*multim im parvo*”, co z kolei oznacza „*many in one*”. Mipmap'a generowana jest z wejściowego obrazu poprzez stopniowe (potęgi cyfry 2) zmniejszanie tego obrazu, aż do rozmiaru jednego piksela. Głównym zadaniem mipmap'y jest niwelacja skutków zmniejszania tekstury (ścieśniania - minification). Z tego względu, że generacja mipmap następuje jeszcze przed użyciem danej tekstury, to przyspiesza to proces renderowania w czasie rzeczywistym. Nie jest konieczne obliczanie oraz generowanie obrazów w chwili działania programu. Oczywiście nie ma nic za darmo, wygenerowane obrazy muszą być gdzieś przechowywane. Tekstura wraz z wygenerowanymi mipmap'ami zajmuje o 1/3 więcej pamięci, niż sama tekstura bez mipmap.

Na dużej powierzchni przykrytej teksturą zostanie użytych kilka mipmap. W obszarach bliżej kamery zostaną użyte większe mipmap'y, np. 64x64, a w obszarach dalej kamery, zostaną użyte mniejsze mipmap'y, np. 8x8. W wyniku użycia kilku wielkości mipmap pojawią się widoczne granice, gdzie jedna mipmap'a się kończy, a druga się zaczyna (rys. 24.3). Aby wygładzić moment przejścia jednej mipmap'y w drugą, stosuje się **filtrowanie trójliniowe (tri-linear filtering)**. Jest ono rozszerzeniem filtrowania dwuliniowego o dodatkowy krok, polegający na wykonaniu dodatkowego filtrowania liniowego na sąsiadujących mipmap'ach.



Rys. 24.3. Różnica pomiędzy filtrowaniem dwuliniowym (po lewej), a trójliniowym (po prawej). Widoczne granice mipmap. [Źródło obrazka: wikipedia.pl]

Implementacja tego filtra w kodzie jest następująca:

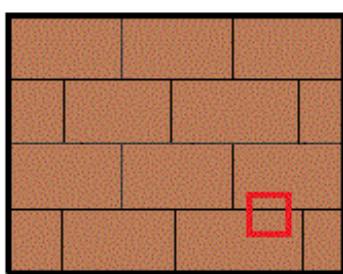
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Pierwsza linia kodu określa, że przy powiększaniu tekstury chcemy korzystać z filtru dwuliniowego. Generowane mipmap'y nigdy nie są większe od wejściowej tekstury, dlatego przy powiększaniu nie mamy możliwości mieszania mipmap (bo ich nie ma). Wejściowa tekstura jest tylko powiększana do odpowiedniego rozmiaru i na niej stosujemy

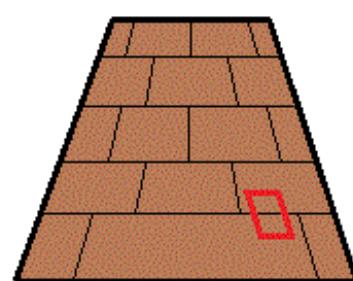
filtrowanie dwuliniowe. Natomiast w przypadku zmniejszania tekstuury, mamy dostęp do mipmap, a zatem warto jest to wykorzystać. Druga linia kodu, a dokładniej wartość `GL_LINEAR_MIPMAP_LINEAR`, określa, że chcemy stosować filtrowanie dwuliniowe, a ponadto chcemy także nałożyć filtr dwuliniowy na sąsiednie mipmap'y, niwelując widoczne granice pomiędzy nimi.

3. Filtrowanie anizotropowe.

Powysze dwie metody filtrowania sprawdzają się najlepiej, gdy teksturowana powierzchnia jest prostopadła do osi kamery. Jeśli natomiast powierzchnia jest pod pewnym kątem nachylona do kamery, to tekstuura ulega rozmyciu. Obliczanie koloru na podstawie tekstuur sąsiadujących nie jest wtedy dobrym rozwiązaniem, gdyż następuje utrata informacji wynikającej z perspektywy (ściśnięcie jednego końca tekstuury). Aby osiągnąć bardziej realistyczne efekty, a nie rozmycie tekstuury, stosuje się **filtrowanie anizotropowe**. Uzgólnia ono kąt widzenia podczas obliczania koloru fragmentu na podstawie tekstuur zgodnych z kierunkiem nachylenia powierzchni (rys. 24.4).



Filtrowanie liniowe



Filtrowanie anizotropowe

Rys. 24.4. Porównanie filtrowania liniowego z filtrowaniem anizotropowym.

Podczas implementacji filtra anizotropowego w kodzie programu należy określić **współczynnik anizotropii** (zależności od kierunku). Jeśli współczynnik ten jest równy 1.0, to mamy do czynienia z normalnym filtrowaniem (nie anizotropowym). Im ta wartość jest większa, tym więcej tekstuur w danym kierunku jest uwzględnianych. Maksymalną wartość tego współczynnika obsługiwanej przez sprzęt można odczytać za pomocą funkcji `glGetFloatv()` z parametrem `GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT`.

```
GLfloat anisotropy_factor = 0.0f;
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &anisotropy_factor);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, anisotropy_factor);
```

Funkcja `glTexParameterf()` służy do ustawiania parametrów tekstuury. Tutaj włączamy filtrowanie anizotropowe poprzez ustawienie współczynnika anizotropii.

Kod źródłowy z tego rozdziału jest w katalogu: **14_Filtrowanie_tekstur**

25. Oświetlenie

Nadszedł czas rzucić trochę światła na naszą scenę. W tym rozdziale wytłumaczę jak stworzyć w OpenGL źródło światła oraz jak przygotować obiekty, aby mogły być oświetlane oraz cieniowane.

Nowoczesny OpenGL nie dysponuje gotowymi funkcjami, które w prosty sposób wprowadziłyby źródło światła do procesu renderowania. Wbrew pozorom jest to zaleta, gdyż mamy możliwość pełnej ingerencji w proces obliczania oświetlenia. Dzięki temu możemy stworzyć bardzo zaawansowany model obliczeń, który uwzględnia odbicie, załamanie oraz rozproszenie światła na obiektach otoczenia uzyskując realistycznie wyglądający obraz. Bardzo dokładny fizyczny model oświetlenia jest oczywiście także bardzo skomplikowany obliczeniowo, a więc nie do końca chcemy używać go podczas renderowania w czasie rzeczywistym, szczególnie na słabszych maszynach. Równie dobrze możemy stworzyć prostszy model obliczeń oświetlenia, który pozwoli nam także uzyskać zadowalające efekty wizualne.

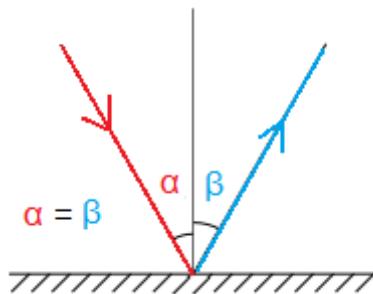
W tym rozdziale zajmiemy się **oświetleniem Phonga**, które jest powszechnie stosowane w grafice komputerowej. Oświetlenie to posiada trzy składowe:

- światło odbite zwierciadlanie (specular),
- światło rozproszone (diffuse),
- światło otoczenia (ambient).

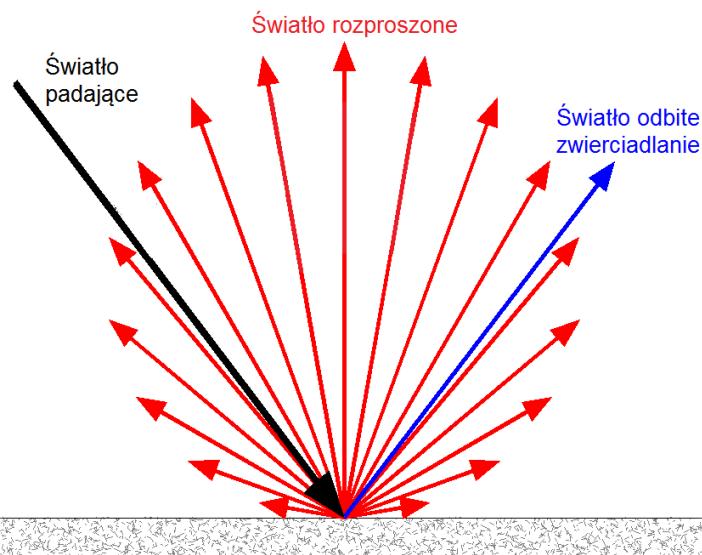
Światło otoczenia jest to nic innego jak światło emitowane przez otoczenie. Ten rodzaj światła nie podlega tłumieniu atmosferycznemu, jest ono jednakowe w każdym miejscu. Można powiedzieć, że im wartość natężenia światła otoczenia jest większa, tym cała scena jest jaśniejsza (jednakowo) oświetlona.

Światło odbite zwierciadlanie powstaje w wyniku odbicia światła od powierzchni. Światło pochodzące z pewnego kierunku odbija się od powierzchni pod tym samym kątem, pod którym padało na tą powierzchnię (rys. 25.1). Intensywność światła odbitego może w jakiś sposób zależeć od kąta padania światła, ale najczęściej wprowadza się uproszczenie i intensywność światła odbitego jest równa intensywności światła padającego. Kolor światła odbitego jest również taki sam jak kolor światła padającego.

Światło rozproszone jest wynikiem chropowatości powierzchni. Światło padające na powierzchnię ulega rozproszeniu na niedoskonałościach powierzchni i ulega odbiciu pod różnymi kątami (rys. 25.2). Źródłem światła rozproszonego są zwykle powierzchnie matowe. Światło rozproszone przyjmuje kolor powierzchni, na którą pada.



Rys. 25.1. Światło odbite zwierciadlanie. Kąt padania i odbicia są sobie równe.



Rys. 25.2. Powstawanie światła rozproszonego na powierzchni.

Oświetlenie Phonga posiada w swoim modelu trzy główne uproszczenia:

- cienie nie są tworzone,
- nie zachodzi zjawisko załamania światła (refrakcji),
- nie modelowane są odbicia pomiędzy obiekty, tzn. światło wychodząc ze źródła, pada na obiekt, a następnie bezpośrednio do oka (kamery). Światło odbite od jednego obiektu nie odbija się już na drugim obiekcie.

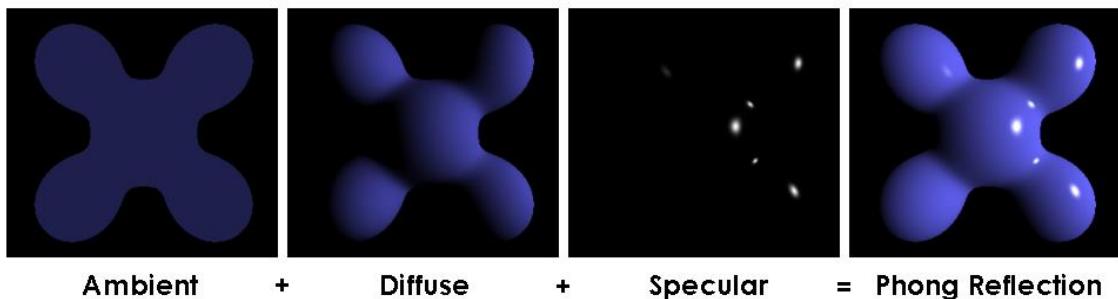
Oświetlenie obliczane jest **na każdy fragment**. Kiedyś obliczenia wykonywane były **na każdy wierzchołek**, a następnie wartość oświetlenia była interpolowana na fragmenty. Obliczenia przeprowadzane dla każdego fragmentu są bardziej kosztowne, ale dają lepsze efekty wizualne. Poza tym dzisiejszy sprzęt nie ma już problemów z przeprowadzaniem takich obliczeń. Ten większy koszt obliczeń wynika przede wszystkim z tego, że fragmentów jest zawsze więcej niż liczby wierzchołków.

Wynikowa intensywność światła przypadająca na fragment jest obliczana z następującego wzoru:

$$I = I_A + I_S + I_D$$

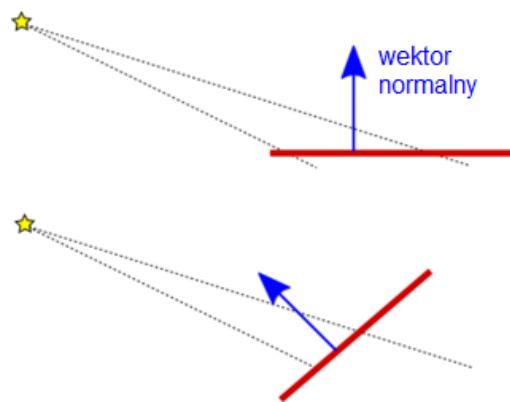
gdzie: I_A - intensywność światła otoczenia, I_S - intensywność światła odbitego zwierciadlanie, I_D - intensywność światła rozproszonego.

Każde z tych świateł posiada swój pewien kolor, a zatem wynikowy kolor fragmentu będzie równy sumie tych trzech kolorów (rys. 25.3).



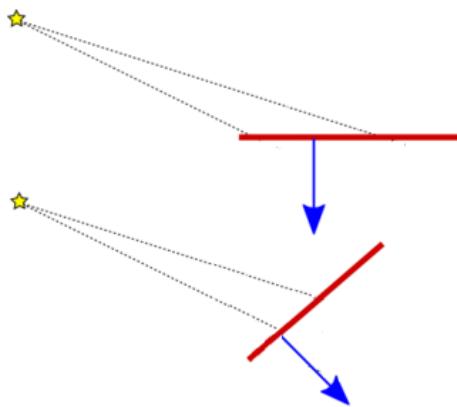
Rys. 25.3. Ilustracja oświetlenia Phong. [Źródło obrazka: wikipedia.pl]

Aby to wszystko miało sens, musimy mieć oczywiście co oświetlać na naszej scenie. Każdy model trójwymiarowy jak wiemy, składa się z wierzchołków oraz płaszczyzn rozciągniętych pomiędzy wierzchołkami (tak jak płótno). Na płótno to jest nakładany jakiś kolor, tekstura oraz oświetlenie. Obliczenie oświetlenia nie byłoby jednak możliwe bez podania, na której stronie tego płótna należy obliczać oświetlenie. Pamiętacie zagadnienie *face culling*, gdzie to nie rysowaliśmy na powierzchni skierowanych do nas tyłem? W przypadku oświetlenia jest tak samo. Tutaj natomiast tą stronę płaszczyzny określamy za pomocą **wektora normalnego**. Wektor normalny jest to taki zwykły wektor, z tą różnicą, że jest on prostopadły do powierzchni, z którą jest związany. Dzięki temu wektorowi światło padające na powierzchnię „wie” pod jakim kątem ustawiona jest do niego powierzchnia (rys. 25.4) i może przeprowadzić odpowiednie obliczenia. Na podstawie tego wektora możemy także obliczyć, pod jakim kątem kamera jest skierowana do powierzchni.



Rys. 25.4. W tym przypadku oświetlenie zostanie obliczone dla tych powierzchni, gdyż światło pada na tą „właściwą” stronę powierzchni.

Jeśli powierzchnia byłaby ustawiona tak, że wektor normalny powierzchni ma przeciwny zwrot, to oznaczałoby to wtedy, że ta powierzchnia (**ta strona powierzchni**) ma nie podlegać obliczeniom oświetlenia (rys. 25.5). Oświetlenie nie jest zatem obliczane dla wnętrza modeli trójwymiarowych. Wektory normalne powierzchni modelu trójwymiarowego skierowane są na zewnątrz tego modelu, a nie do wewnętrz.

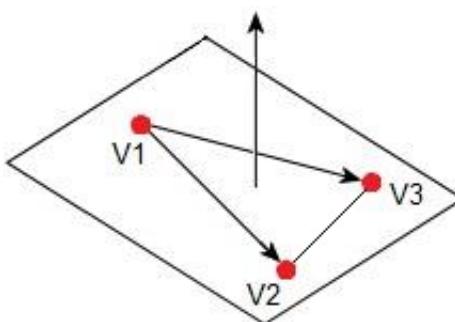


Rys. 25.5. W tym przypadku oświetlenie nie zostanie obliczone. Światło pada na „niewłaściwą” stronę powierzchni, która jest na przykład wewnątrz jakiegoś modelu trójwymiarowego.

Długość wektora normalnego powinna być równa 1. Wektory o długości 1 nazywamy także **wektoram i znormalizowanymi**. Wektor normalny jest zatem wektorem znormalizowanym.

Normalizacja wektora polega na podzieleniu wszystkich składowych wektora przez jego długość.

Wektor prostopadły do powierzchni możemy wyznaczyć jako iloczyn wektorowy dwóch wektorów leżących na tej powierzchni (już o tym pisałem wcześniej). Możemy wyznaczyć dwa wektory wychodzące z jednego wierzchołka trójkąta (prymitywu), które są skierowane do pozostałych dwóch wierzchołków (rys. 25.6). Następnie należy je znormalizować i ostatecznie obliczyć ich iloczyn wektorowy, aby uzyskać wektor normalny.



Rys. 25.6. Wyznaczanie wektora normalnego.

W rzeczywistości tak to nie wygląda. W prostych modelach trójwymiarowych możemy wpisać to wszystko ręcznie, ale budując złożone modele nie byłoby to wygodne. Dlatego programy służące do modelowania 3D, np. Blender, obliczają wektory normalne za nas. My potem możemy wczytać taki model trójwymiarowy wraz z jego wektorami normalnymi do naszego programu (będzie o tym następny rozdział).

Każdy obiekt można także scharakteryzować trzema parametrami:

- współczynnikiem odbicia światła otoczenia,
- współczynnikiem odbicia światła rozproszonego,
- współczynnikiem odbicia światła odbitego zwierciadlanie.

Współczynniki te określają jaka część światła padającego ulega odbiciu. Jest to stosunek ilości światła odbitego od powierzchni do ilości światła padającego na powierzchnię. Mogą zatem być one z przedziału od 0 do 1. Współczynniki te definiujemy dla każdej składowej koloru, tzn. dla R, G i B. Jeśli jakiś obiekt posiada współczynniki odbicia dla światła rozproszonego (1.0, 0.0, 0.0), a światło ma kolor (1.0, 1.0, 1.0), tzn. że obiekt odbije tylko światło koloru czerwonego. Współczynniki te określają zatem materiał, z jakiego wykonany jest obiekt. Jeśli obiekt odbija światło koloru czerwonego to naszym oczom wydaje się on czerwony, jeśli odbija zielone światło to będzie on zielony, itd.

Przystąpmy do kodowania. Na początku musimy opisać parametrami źródło światła. Jak pisałem wyżej, oświetlenie obliczamy na każdy fragment, a zatem parametry źródła światła podajemy z reguły w fragment shader. Można to zrobić bezpośrednio na sztywno w kodzie fragment shader albo wykorzystać do tego celu zmienne typu uniform.

Pokażę wersję najprostszą, tzn. tą bez użycia zmiennych uniform. Zdefiniuję zmienne na sztywno w kodzie shadera. Przerobienie na zmienne uniform nie powinno już sprawiać Ci problemu.

Dodajmy przed funkcją main() w fragment shader następujące zmienne opisujące parametry źródła światła. Jest to położenie źródła światła na scenie oraz kolory poszczególnych światel.

```
vec3 light position = vec3(3.0, 0.0, 1.0);
vec3 ambient color = vec3(0.1, 0.1, 0.1);
vec3 diffuse color = vec3(0.7, 0.7, 0.7);
vec3 specular_color = vec3(1.0, 1.0, 1.0);
```

Są to zmienne typu vec3, bo oczywiście na każdą zmienną musimy przypisać trzy wartości. W przypadku położenia źródła światła jest to położenie X, Y i Z, a w przypadku koloru mamy wartości R, G i B.

Nieco wyżej pisałem także o współczynnikach odbicia obiektu. Musimy je także zdefiniować w kodzie fragment shader. Oczywiście w tym przypadku wygodniej jest użyć zmienne uniform i dla każdego obiektu, który rysujemy, wysyłać do shadera jego indywidualne współczynniki odbicia. Możemy wtedy rozróżnić obiekty bardziej świecące od matowych. W ramach uproszczenia natomiast definiuję te współczynniki na sztywno w kodzie fragment shader.

```
vec3 object_ambient_factor = vec3(1.0, 1.0, 1.0);
vec3 object_diffuse_factor = vec3(0.5, 0.5, 0.5);
vec3 object_specular_factor = vec3(1.0, 1.0, 1.0);
```

Powyżej określiliśmy, że chcemy, aby obiekt odbijał w pełni oświetlenie otoczenia (ambient) oraz oświetlenie odbite zwierciadlanie (specular). Każda składowa koloru w trzech przypadkach jest odbijana z identyczną intensywnością.

Światło otoczenia oświetla jednakowo całą scenę. Jeśli na przykład na scenie ustawiemy jest jakiś obiekt, to z każdej strony jest oświetlony tak samo. Aby zatem używać takiego rodzaju oświetlenia nie musimy definiować wektorów normalnych powierzchni.

Intensywność światła otoczenia obliczamy poprzez pomnożenie koloru tego rodzaju światła i współczynnika odbicia obiektu dla tego światła.

```
vec3 ambient_intense = ambient_color * object_ambient_factor;
```

Jeśli nakładamy na obiekt teksturę, to podczas obliczania wynikowego koloru oświetlonego obiektu, musimy także wziąć pod uwagę kolor danego fragmentu pochodzący z tekstury. Wystarczy, że wykonamy jeszcze jedno dodatkowe mnożenie przez kolor danego fragmentu.

```
vec4 texel = texture(basic_texture, texture_coordinates);
frag_colour = vec4(ambient_intense, 1.0) * texel;
```

Ostatecznie kolor danego fragmentu jest wynikiem „zmieszania” koloru oświetlenia i koloru tekstury. Trzeba także zrzutować zmienną `ambient_intense` na typ `vec4`, aby mnożenie było możliwe, gdyż zmienna `texel` jest właśnie typu `vec4`.

Cały kod fragment shader jest następujący:

```
#version 330
in vec2 texture_coordinates;

uniform sampler2D basic_texture;

out vec4 frag_colour;

vec3 light_position = vec3(3.0, 0.0, 1.0);
vec3 ambient_color = vec3(0.1, 0.1, 0.1);
vec3 diffuse_color = vec3(0.7, 0.7, 0.7);
vec3 specular_color = vec3(1.0, 1.0, 1.0);

vec3 object_ambient_factor = vec3(1.0, 1.0, 1.0);
vec3 object_diffuse_factor = vec3(0.5, 0.5, 0.5);
vec3 object_specular_factor = vec3(1.0, 1.0, 1.0);

void main()
{
    // Ambient
    vec3 ambient_intense = ambient_color * object_ambient_factor;

    vec4 texel = texture(basic_texture, texture_coordinates);
    frag_colour = vec4(ambient_intense, 1.0) * texel;
}
```

Kod vertex shader nie uległ żadnej zmianie:

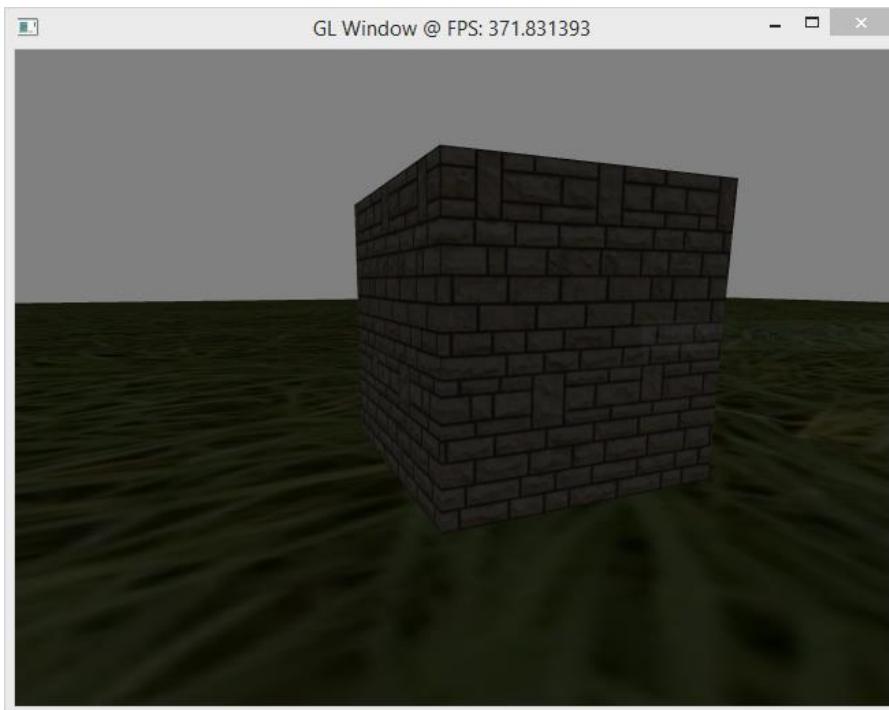
```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;

uniform mat4 view_matrix;
uniform mat4 perspective_matrix;

out vec2 texture_coordinates;

void main()
{
    texture_coordinates = vt;
    gl_Position = perspective_matrix * view_matrix * vec4(position, 1.0);
}
```

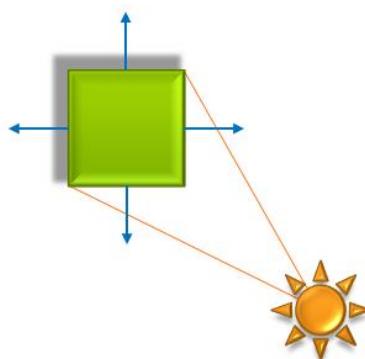
W wyniku otrzymamy taki efekt jak na rys. 25.7.



Rys. 25.7. Nałożone oświetlenie otoczenia na scenę.

Dodajmy teraz kolejny składnik oświetlenia, czyli światło rozproszone. Nadmienię jeszcze, że mamy do czynienia z **punktowym źródłem światła**. Jest to najprostsze źródło światła. Cechą tego rodzaju źródła światła jest to, że świeci w każdym kierunku jednakowo. Taka pseudo żarówka.

Oświetlając jakiś obiekt taką żarówką oświetlamy tylko jego jedną stronę (rys. 25.8). Druga strona pozostaje zacieniona. Musimy zatem mieć informację o tym, z której strony obiekt jest oświetlany, gdzie znajduje się źródło światła względem obiektu oraz jaki jest kąt padania promieni.



Rys. 25.8. Oświetlanie obiektu punktowym źródłem światła. Na niebiesko zaznaczone są wektory normalne powierzchni.

W tym celu definiujemy wektory normalne. Tak samo jak w przypadku położenia wierzchołków, czy współrzędnych tekstury, jest to tablica zmiennych typu `float`. Dla każdego wierzchołka określamy wektor kierunku związanej z nim powierzchni.

Odnosząc się do naszego przykładu, tablica wektorów normalnych dla podłożą (trawy) wygląda następująco:

```
GLfloat normals_floor[] = {
    0.0f, 1.0f, 0.0f,
    0.0f, 1.0f, 0.0f
};
```

Każdemu wierzchołkowi przypisany jest wektor skierowany pionowo w góre zgodnie z kierunkiem osi Y. Przypominam, że musi to być wektor o długości 1, a zatem w miejscu współrzędnej Y wpisujemy wartości 1.0.

Teraz taką tablicę musimy wpisać do obiektu VAO na przykład do indeksu numer 2, gdyż pod indeksem 0 mamy już współrzędne wierzchołków, a pod indeksem 1 umieściliśmy wcześniej współrzędne tekstury. Tego nie pokazuję, bo wiemy już jak to zrobić. Polecam zajrzeć do kodu źródłowego.

Następnym krokiem jest konieczność dodania trochę kodu do vertex shader. Podczas obliczania oświetlenia wszystkie wektory oraz współrzędne wierzchołków **musimy mieć wyrażone w tej samej przestrzeni** (układzie współrzędnym). Dla wygody, najlepiej jest wszystko przeliczyć do układu współrzędnych kamery.

Musimy zatem przeliczyć współrzędne aktualnie przetwarzanego wierzchołka oraz wektor normalny do układu współrzędnych kamery.

```
normal_to_camera = vec3(view_matrix * vec4(normal_vector, 0.0));
vertex_to_camera = vec3(view_matrix * vec4(position, 1.0));
```

Przeliczając wektor normalny musimy pamiętać, że przeliczamy kierunek, dlatego jako ostatnią współrzędną wektora cztero-wymiarowego wstawiamy 0.

Te dwie przeliczone zmienne musimy dalej przekazać do fragment shader, gdzie obliczamy oświetlenie. Definiujemy je zatem jako zmienne typu `out`.

Vertex shader po tych modyfikacjach wygląda następująco:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;
layout(location = 2) in vec3 normal_vector;

uniform mat4 view_matrix;
uniform mat4 perspective_matrix;
out vec2 texture_coordinates;
out vec3 vertex_to_camera;
out vec3 normal_to_camera;

void main()
{
    normal_to_camera = vec3(view_matrix * vec4(normal_vector, 0.0));
    vertex_to_camera = vec3(view_matrix * vec4(position, 1.0));

    texture_coordinates = vt;
    gl_Position = perspective_matrix * view_matrix * vec4(position, 1.0);
}
```

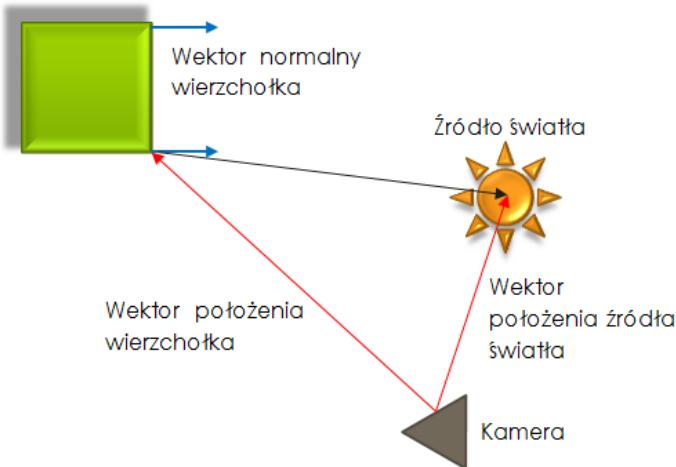
Teraz możemy przystąpić do modyfikacji fragment shader. Musimy przyjmować te dwie zmienne wysyłane z vertex shader. Będziemy jeszcze dodatkowo potrzebowali odczytywać stan zmiennej `view_matrix`, dlatego definiujemy ją jako zmienną `uniform` (tak samo jak w vertex shader).

```
in vec3 vertex_to_camera;
in vec3 normal_to_camera;
uniform mat4 view_matrix;
```

Aby dokonać odpowiednich obliczeń oświetlenia powierzchni, potrzebujemy znać kierunek (wektor) od obliczanego właśnie wierzchołka do źródła światła (rys. 25.9 - wektor czarny). Żeby wyznaczyć ten kierunek, musimy pierw obliczyć różnicę pomiędzy wektorem położenia źródła światła oraz wektorem położenia wierzchołka. Oczywiście, te dwa wektory muszą być wyrażone w układzie współrzednym kamery (rys. 25.9 - wektory czerwone). Otrzymany wektor należy następnie znormalizować. Ostatnim krokiem jest porównanie otrzymanego znormalizowanego wektora oraz wektora normalnego. Porównanie to polega na obliczeniu iloczynu skalarnego tych dwóch wektorów. Iloczyn skalarny informuje nas o tym, pod jakim kątem względem siebie są te dwa wektory ustawione.

```
vec3 distance_to_light = vec3(view_matrix * vec4(light_position, 1.0)) - vertex_to_camera;
vec3 light_direction = normalize(distance_to_light);
float dot_product = dot(light_direction, normal_to_camera);
```

Funkcja `dot()` oblicza iloczyn skalarny. W tym przypadku, jako jej argumenty podajemy dwa wektory o długości 1, a zatem w wyniku otrzymamy **kosinus kąta**, jaki jest pomiędzy tymi dwoma wektorami. Jeśli dwa wektory są do siebie prostopadłe, to iloczyn skalarny wyniesie 0, gdyż kosinus z kąta prostego jest równy 0. Jeśli natomiast kąt pomiędzy nimi jest równy 0° , to w wyniku otrzymamy wartość 1.



Rys. 25.9. Schematyczne ułożenie wektorów podczas obliczania światła rozproszonego.

Przypomnę, że w matematyce geometrycznej reprezentacja iloczynu skalarnego jest następująca:

$$\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos \alpha$$

Wartość otrzymana w wyniku iloczynu skalarnego może wyjść ujemna (wtedy, gdy kąt alfa $> 90^\circ$). Oznacza to, że ta powierzchnia jest skierowana przeciwnie do źródła światła. Nie chcemy dla takiej powierzchni obliczać światła rozproszonego. W tym celu możemy użyć funkcji `max()`, która zwraca maksymalną wartość z podanego przedziału. Jeśli wartość jest ujemna, funkcja zwróci nam 0.

```
dot_product = max(dot_product, 0.0);
```

Gdy już wszystko mamy gotowe, możemy obliczyć ostateczną intensywność światła rozproszonego. Należy przemnożyć przez siebie kolor światła, współczynnik odbicia obiektu dla tego światła oraz uzyskaną w poprzednim kroku wartość iloczynu skalarnego. Jeśli ta ostatnia wartość jest równa 0, to intensywność światła rozproszonego będzie równa 0. A jeśli jest równa 1, to intensywność będzie największa.

```
vec3 diffuse_intense = diffuse_color * object_diffuse_factor * dot_product;
```

Obliczoną intensywność musimy jeszcze dodać do poprzednio obliczonej intensywności światła otoczenia.

```
frag_colour = vec4(ambient_intensity + diffuse_intense, 1.0) * texel;
```

Cały fragment shader wygląda teraz następująco:

```
#version 330
in vec2 texture_coordinates;
in vec3 vertex_to_camera;
in vec3 normal_to_camera;

uniform mat4 view_matrix;
uniform sampler2D basic_texture;

out vec4 frag_colour;

vec3 light_position = vec3(3.0, 0.0, 1.0);
vec3 ambient_color = vec3(0.1, 0.1, 0.1);
vec3 diffuse_color = vec3(0.7, 0.7, 0.7);
vec3 specular_color = vec3(1.0, 1.0, 1.0);

vec3 object_ambient_factor = vec3(1.0, 1.0, 1.0);
vec3 object_diffuse_factor = vec3(0.5, 0.5, 0.5);
vec3 object_specular_factor = vec3(1.0, 1.0, 1.0);

void main()
{
    // Ambient
    vec3 ambient_intensity = ambient_color * object_ambient_factor;

    // Diffuse
    vec3 distance_to_light = vec3(view_matrix * vec4(light_position, 1.0)) - vertex_to_camera;
    vec3 light_direction = normalize(distance_to_light);
    float dot_product = dot(light_direction, normal_to_camera);
    dot_product = max(dot_product, 0.0);
    vec3 diffuse_intense = diffuse_color * object_diffuse_factor * dot_product;

    vec4 texel = texture(basic_texture, texture_coordinates);
    frag_colour = vec4(ambient_intensity + diffuse_intense, 1.0) * texel;
}
```

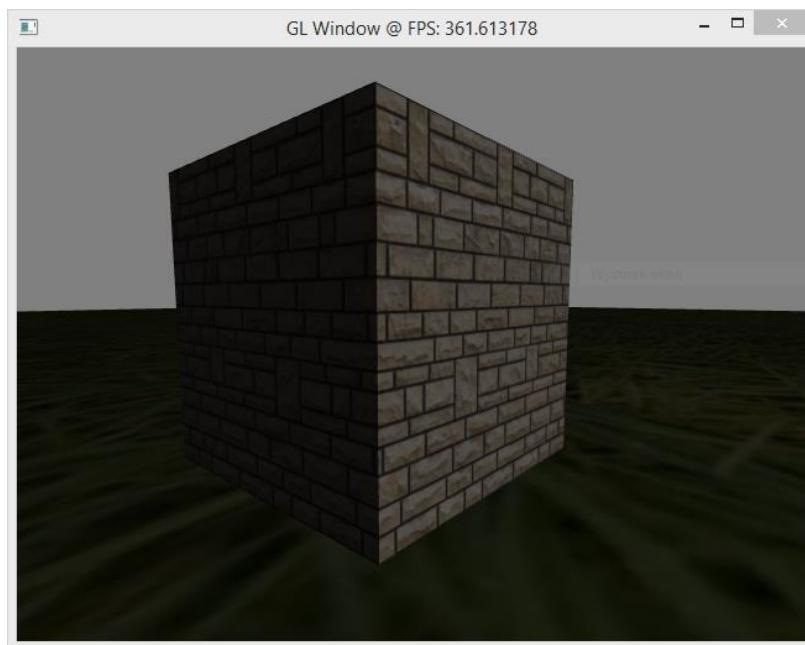
Jeśli uruchomimy program w tym stanie, to otrzymamy efekt jak na rys. 25.10.

Ostatnim krokiem będzie wprowadzenie obliczeń światła odbitego zwierciadlanie. Vertex shader nie wymaga już żadnych modyfikacji. Skupiamy się dalej tylko na fragment shader.

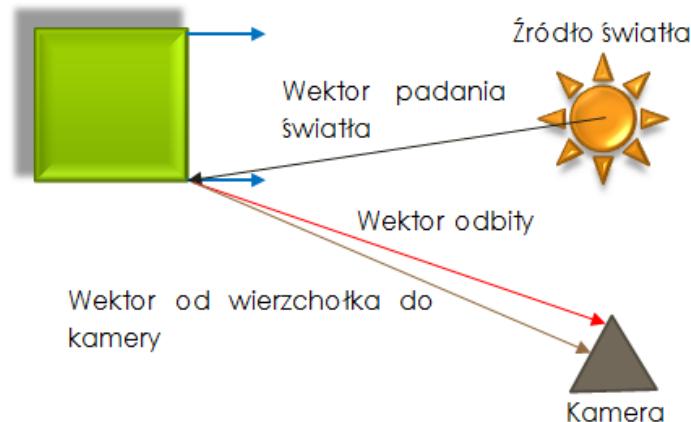
Na początku dodajmy do fragment shader ten kawałek kodu:

```
vec3 reflection = reflect(-light direction, normal to camera);
vec3 surface_to_camera = normalize(-vertex_to_camera);
float dot_specular = dot(reflection, surface_to_camera);
dot_specular = max(dot_specular, 0.0);
```

Światło odbite zwierciadlanie polega na tym, że widzimy je tylko wtedy, gdy patrzymy się na powierzchnię pod takim samym kątem, jak pada na nią światło. W pierwszej kolejności obliczamy zatem wektor kierunku odbitego światła (rys. 25.11 - wektor czerwony). Służy do tego funkcja `reflect()`. W pierwszym parametrze podajemy jej wektor padający (rys. 25.11 - wektor czarny), a w drugim wektor normalny powierzchni. W poprzednim kroku obliczaliśmy wektor kierunku od wierzchołka do światła (rys. 25.9 - wektor czarny) `light_direction`. Wektor padania światła (rys. 25.11 - wektor czarny) będzie wektorem odwrotnym do tego wektora, a zatem stawiamy przed nim minus.



Rys. 25.10. Efekt działania programu po uwzględnieniu światła rozproszenia.



Rys. 25.11. Schematyczne ułożenie wektorów podczas obliczania światła odbitego zwierciadlanie.

Kierunek patrzenia na powierzchnię otrzymamy normalizując wektor określający kierunek z wierzchołka do kamery. Wektor ten także musimy odwrócić, gdyż potrzebujemy wektor wychodzący z kamery w stronę wierzchołka (rys. 25.11 - wektor brązowy).

Ostatecznie porównujemy te dwa wektory za pomocą funkcji `dot()`. W wyniku otrzymujemy współczynnik (kosinus kąta) informujący nas o tym, w jakim stopniu wektor patrzenia jest zbliżony do wektora promienia odbitego. Jeśli kąt pomiędzy tymi wektorami jest równy 0 (leżą na tej samej prostej, mają taki sam kierunek), to otrzymamy wartość 1 (kosinus z kąta 0 stopni jest równy 1).

Musimy jeszcze określić jak duży ma być promień światła odbitego od powierzchni. W tym celu definiujemy zmienną `specular_power`, a następnie obliczamy współczynnik światła odbitego. Do tego celu wykorzystuję funkcję `pow()`, która podnosi pierwszy podany parametr do potęgi podanej w drugim parametrze.

```
float specular_power = 10.0;
float specular_factor = pow(dot_specular, specular_power);
```

No i na koniec pozostało nam obliczyć już tylko intensywność światła odbitego. W tym celu przemnażamy kolor światła odbitego, współczynnik odbicia obiektu dla tego rodzaju światła oraz uzyskany powyżej współczynnik.

```
vec3 specular_intense = specular_color * object_specular_factor * specular_factor;
```

Cały fragment shader prezentuje się następująco:

```
#version 330
in vec2 texture_coordinates;
in vec3 vertex_to_camera;
in vec3 normal_to_camera;

uniform mat4 view_matrix;
uniform sampler2D basic_texture;

out vec4 frag_colour;

vec3 light_position = vec3(3.0, 0.0, 1.0);
vec3 ambient_color = vec3(0.1, 0.1, 0.1);
vec3 diffuse_color = vec3(0.7, 0.7, 0.7);
vec3 specular_color = vec3(1.0, 1.0, 1.0);

vec3 object_ambient_factor = vec3(1.0, 1.0, 1.0);
vec3 object_diffuse_factor = vec3(0.5, 0.5, 0.5);
vec3 object_specular_factor = vec3(1.0, 1.0, 1.0);

void main()
{
    // Ambient
    vec3 ambient_intense = ambient_color * object_ambient_factor;

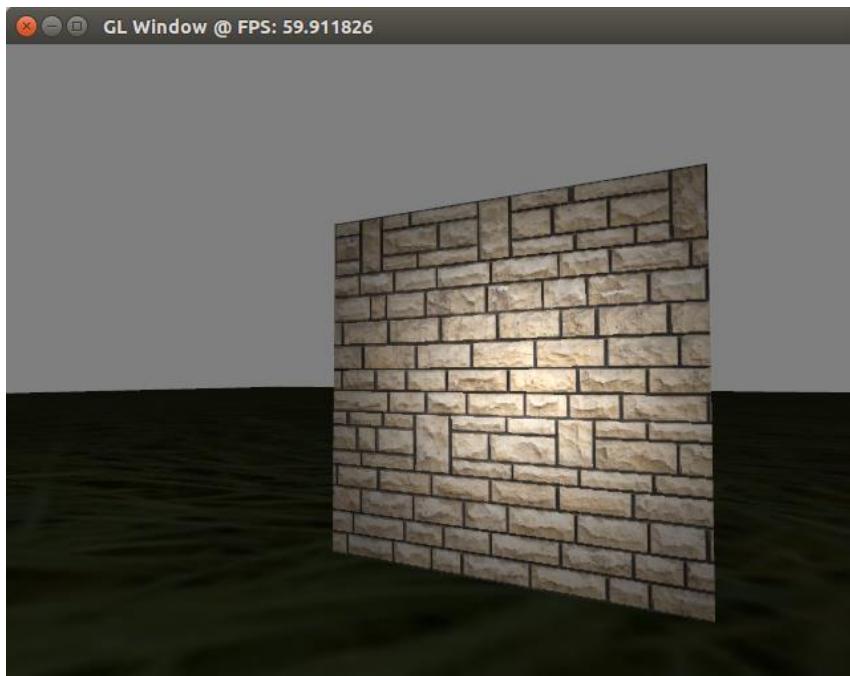
    // Diffuse
    vec3 distance_to_light = vec3(view_matrix * vec4(light_position, 1.0)) - vertex_to_camera;
    vec3 light_direction = normalize(distance_to_light);
    float dot_product = dot(light_direction, normal_to_camera);
    dot_product = max(dot_product, 0.0);
    vec3 diffuse_intense = diffuse_color * object_diffuse_factor * dot_product;

    // Specular
    vec3 reflection = reflect(-light_direction, normal_to_camera);
    vec3 surface_to_camera = normalize(-vertex_to_camera);
    float dot_specular = dot(reflection, surface_to_camera);
    dot_specular = max(dot_specular, 0.0);
    float specular_power = 10.0;
    float specular_factor = pow(dot_specular, specular_power);
```

```
vec3 specular_intense = specular_color * object_specular_factor * specular_factor;  
vec4 texel = texture(basic_texture, texture coordinates);  
frag colour = vec4(ambient intense + diffuse intense + specular intense, 1.0) * texel;  
}
```

Po uruchomieniu uzyskamy coś podobnego jak na rys. 25.12.

Ten rozdział był bardzo trudny. Te wszystkie wektory ... Straszne. Jeśli wszystkiego nie zrozumiałeś od razu, to nie przejmuj się. Z czasem wszystko się wykłaruje. Ewentualnie możesz rozrysować sobie te wektory na kartce papieru i zobaczyć jak to dokładnie działa.

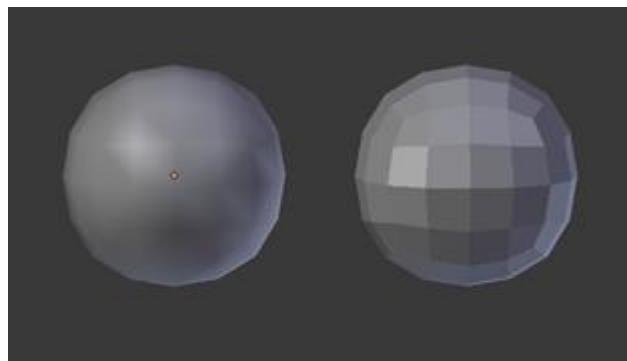


Rys. 25.12. Widoczny efekt odbicia światła na powierzchni.

Tips & Tricks:

- Należy pamiętać o tym, aby nie wykonywać operacji na wektorach, które wyrażone są w innych układach współrzędnych. Można dodawać sobie prefiksy lub postfiksy to nazw zmiennych, określające układ współrzędnych tego wektora, np. `normal_on_camera` lub `position_world`.
- Czterowymiarowy wektor kierunku powinien mieć współrzędną W ustawioną na 0. Natomiast, czterowymiarowy wektor położenia powinien mieć współrzędną W równą 1.
- Zarządzanie kilkoma źródłami światła na scenie jest dość trudne. Obliczanie oświetlenia pochodzącego z kilku źródeł jest także kosztowne do obliczenia. Najlepiej jest obliczać oświetlenie pochodzące tylko z źródeł światła znajdujących się najbliżej kamery.
- To jak bardzo wygięta wydaje się nam oświetlona powierzchnia zależy od wektorów normalnych. W przypadku oświetlonej kuli jej powierzchnia powinna jak najbardziej wydawać się jednolita i gładka. W tym przypadku wektor normalny wierzchołka powinien być obliczony jako średnia z otaczających wektorów normalnych powierzchni (**smooth shading**). Natomiast w przypadku płaskich obiektów, takich jak ściany, stosowane jest cieniowanie płaskie (**flat shading**).

Polega ono na obliczeniu wektora normalnego powierzchni poprzez iloczyn wektorowy dwóch krawędzi wychodzących z tego wierzchołka. Porównanie tych dwóch metod cieniowania jest na rys. 25.13. W programach graficznych służących do modelowania, np. Blender, mamy możliwość wyboru rodzaju cieniowania.



Rys. 25.13. Porównanie smooth shading (lewa strona) i flat shading (prawa strona).

Kod źródłowy z tego rozdziału jest w katalogu: **15_Phong**

26. Wczytywanie modeli z pliku

Ręczne definiowanie wierzchołków jest strasznie uciążliwe. W przypadku bardzo prostych modeli graficznych jak na przykład kostka, jest to możliwe, natomiast chcąc posługiwać się bardziej skomplikowanymi modelami, musimy nauczyć się wczytywać je z pliku.

Stosowanie gotowych modeli zwiększa efektywność oraz bezbłędność pracy. Po pierwsze, projektując model w programie służącym do modelowania grafiki trójwymiarowej, efekty pracy widoczne są od razu. Można na bieżąco śledzić wygląd modelu i go modyfikować. Definiowanie wierzchołków w kodzie programu nie daje takiej możliwości. Po drugie, programista nie musi grzebać się w tworzenie modeli, może tym zająć się grafik. A poza tym, w sieci dostępne są setki darmowych i bardzo dobrych modeli, które można wykorzystywać w celach hobbystycznych.

Plik modelu trójwymiarowego dostarcza takie informacje, jakie byliśmy zmuszeni z reguły podawać ręcznie w kodzie programu. A więc są to:

- współrzędne wierzchołków,
- wektory normalne wierzchołków,
- współrzędne tekstury,
- materiały i tekstury.

Formatów zapisu plików z modelami 3D jest bardzo dużo. Każdy ma swoją indywidualną wewnętrzną strukturę zapisu danych. Najprostszą strukturę mają pliki **OBJ**. Są one także najbardziej popularne z tego względu, że są wspierane przez każdy program do modelowania 3D.

Tabela poniżej przedstawia zestawienie najczęściej używanych formatów plików z modelami trójwymiarowymi.

Format	Animacje	Trudność importu danych
Wavefront (.obj)	nie	bardzo prosto
Lightwave (.lwo)	tak	skomplikowanie
DirectX (.x)	tak	skomplikowanie
Collada (.dae)	tak	przystępnie

Format DirectX (.x) jest nadal popularny mimo tego, że nie jest już oficjalnie wspierany.

Jak wspomniałem wyżej, każdy format ma swoją indywidualną wewnętrzną strukturę zapisu danych, a więc każdy plik musi być odczytywany (parsowany) w inny sposób. Natomiast, sama idea wykorzystania już tych danych w aplikacji zawsze jest taka sama. Zawsze będziemy tymi danymi uzupełniać bufore i wpisywać je do obiektów VBO i VAO.

OpenGL nie jest wyposażony w mechanizm wczytywania modeli z pliku. Z pomocą przychodzi tutaj biblioteka **AssImp**. Obsługuje ona bardzo dużo typów plików i dostarcza

nam dane zawsze w ten sam ujednolicony sposób. Jest to bardzo wygodne, gdyż nie musimy pisać własnych parserów pobierających dane z plików o różnych formatach.

Ja w swoim kursie będę używał najczęściej formatu pliku **Collada**. Został on opracowany przez organizację **Khronos Group** w 2004 roku i wywodzi się z pliku XML. Collada jest plikiem złożonym strukturalnie, przez co jego wczytywanie może trwać dłużej niż pliku OBJ, ale jest to praktycznie niezauważalna różnica.

Napiszmy funkcję, która wczyta model trójwymiarowy z pliku, tak abyśmy potem mogli go wyświetlić na ekranie.

Na początku dołączamy odpowiednie pliki nagłówkowe:

```
#include <assimp/scene.h>
#include <assimp/postprocess.h>
#include <assimp/cimport.h>
```

Nasza funkcja będzie zdefiniowana następująco:

```
int LoadSceneFromFile(std::string file_name, GLuint& vao,
                      std::vector<GLfloat>& mesh_vertices_count,
                      std::vector<GLfloat>& mesh_starting_vertex_index,
                      std::vector<GLuint>& textures)
{
    // ciało funkcji ...
}
```

W pierwszym parametrze podajemy ścieżkę do pliku z modelem trójwymiarowym, a w drugim parametrze przyjmujemy referencję do obiektu VAO. Parametr ten przekazywany jest przez referencję, gdyż obiekt VAO będzie generowany wewnątrz tej funkcji. Pozostałe trzy parametry zostaną omówione później.

Aby wczytać model za pomocą biblioteki Assimp należy skorzystać z funkcji `aiImportFile()`. Przyjmuje ona dwa parametry. Pierwszym jest ścieżka do modelu, który należy wczytać, a w drugim parametrze podajemy operacje, które mają zostać dodatkowo wykonane na tym modelu już po wczytaniu (**post-processing**).

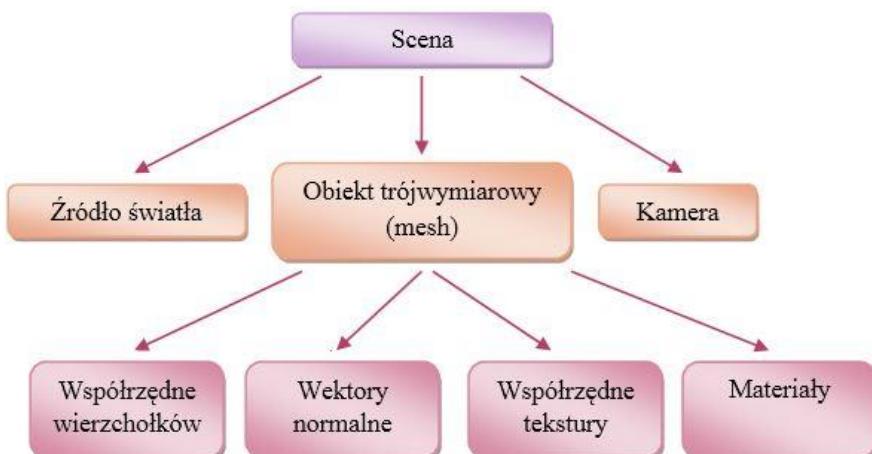
```
const aiScene* scene = aiImportFile(file_name.c_str(), aiProcess_Triangulate);
```

Korzystamy tutaj tylko z jednej opcji, tzn. `aiProcess_Triangulate`, którą jest wykonanie podziału modelu na trójkąty. Niektóre modele mogą być zbudowane z czworoboków jako prymitywów, a zatem, aby zawsze odczytywać wierzchołki w taki sam ujednolicony sposób, ułatwiamy sobie sprawę i dzielimy ewentualne czworoboki na trójkąty. Z nieznanych mi jednak przyczyn, opcja ta nie zawsze działa i niektóre modele mimo to, wczytywane są niepoprawnie (zobacz **Tips & Tricks** na samym końcu tego rozdziału).

Do wyboru jeszcze jest wiele innych opcji post-processing'u, ale na razie nie są nam potrzebne. Ich pełna lista znajduje się w dokumentacji Assimp.

Warto wspomnieć o organizacji plików z modelami trójwymiarowymi. Pliki te zorganizowane są wewnętrznie w pewną hierarchię. Najwyższym stopniem jest **scena**. Na scenie mogą znajdować się takie elementy jak kamery, źródła światła oraz obiekty trójwymiarowe (**mesh**). Kamer oraz źródła światła nie będziemy importować oraz wykorzystywać. Będziemy importować tylko rzeczywiste obiekty trójwymiarowe. Z reguły jest tak, że w jednym pliku zapisanych jest kilka obiektów 3D. Wczytanie wszystkich jest wymagane, gdyż **każdy jest jakąś częścią gotowego ostatecznego modelu 3D**. Na przykład, wczytując z pliku model samochodu, najprawdopodobniej każde koło będzie oddzielnym modelem, a karoseria oddzielnym. W wyniku wczytamy więc pięć modeli, które potem utworzą gotowy samochód. Uprzedzając pytania, nie, nie musimy się martwić o ręczne przesuwanie tych poszczególnych modeli w odpowiednie miejsca gotowego modelu. To już jest zapisane w pliku.

Całą hierarchię pliku przedstawia rys. 26.1.



Rys. 26.1. Hierarchia elementów w pliku z modelem trójwymiarowym.

Dalej definiujemy dwie zmienne:

```
int total_vertices_count = 0;
std::vector<GLfloat> buffer_vbo_data;
```

W pierwszej będziemy przechowywać ile wierzchołków do tej pory wczytaliśmy, a do drugiej zmiennej, będącej tablicą dynamiczną, będziemy dodawać kolejne odczytywane wierzchołki.

Teraz możemy napisać pętlę, która przejdzie po wszystkich obiektach (mesh) na scenie i będziemy mogli zapisać pobrane wierzchołki do bufora. Odczyt liczby obiektów 3D na scenie jest bardzo prosty. Wystarczy tylko odwołanie do zmiennej `mNumMeshes`.

```
for (int i = 0; i != scene->mNumMeshes; i++)
{
    aiMesh* mesh = scene->mMeshes[i];

    int mesh_vertices = 0;

    for (int j = 0; j != mesh->mNumFaces; j++)
    {
        const aiFace* face = &mesh->mFaces[j];
```

```
for (int k = 0; k != 3; k++)  
{  
    aiVector3D vertex position{ 0, 0, 0 };  
    aiVector3D vertex normal{ 0, 0, 0 };  
    aiVector3D vertex_texture_coord{ 0, 0, 0 };  
  
    if (mesh->HasPositions())  
        vertex position = mesh->mVertices[face->mIndices[k]];  
  
    if (mesh->HasNormals())  
        vertex_normal = mesh->mNormals[face->mIndices[k]];  
  
    if (mesh->HasTextureCoords(0))  
        vertex texture coord = mesh->mTextureCoords[0][face->mIndices[k]];  
  
    buffer vbo data.push back(vertex position.x);  
    buffer_vbo_data.push_back(vertex_position.y);  
    buffer_vbo_data.push_back(vertex_position.z);  
  
    buffer vbo data.push back(vertex normal.x);  
    buffer vbo data.push back(vertex normal.y);  
    buffer_vbo_data.push_back(vertex_normal.z);  
  
    buffer_vbo_data.push_back(vertex_texture_coord.x);  
    buffer vbo data.push back(vertex texture coord.y);  
  
    mesh vertices++;  
}  
}  
  
mesh vertices count.push back(mesh vertices);  
mesh starting vertex index.push back(total vertices count);  
total_vertices_count += mesh_vertices;
```

W każdym obiegu pętli pobierany jest wskaźnik do kolejnego modelu, który będzie wczytywany. W zmiennej `mesh_vertices` zliczana będzie liczba wczytyanych wierzchołków tego aktualnego modelu.

Najlepszą metodą na wczytywanie wierzchołków jest przechodzenie po wszystkich powierzchniach modelu, a następnie po każdym wierzchołku tej powierzchni. Liczbę powierzchni odczytujemy za pomocą zmiennej `mNumFaces`, a pobranie kolejnej powierzchni odbywa się za pomocą odwołania do kolejnego elementu tablicy `mFaces[]`.

Na początku wprowadzaliśmy podział modelu na trójkąty, a zatem możemy napisać pętlę, która przejdzie po każdym wierzchołku tego trójkąta. Pętla wykona się trzy razy, czyli tyle, co trójkąt ma wierzchołków.

Za pomocą funkcji `HasPositions()`, `HasNormals()` oraz `HasTextureCoords()` możemy sprawdzić, czy dany wierzchołek posiada zdefiniowane położenie, wektor normalny oraz współrzędne tekstury. Funkcja `HasTextureCoords()` przyjmuje parametr informujący, którą teksturę wierzchołka chcemy pobrać. Wierzchołek bowiem może mieć przypisanych kilka tekstur, które potem są mieszane w celu uzyskania określonych efektów. My pobieramy tylko pierwszą teksturę (indeks 0). Jeśli jakiś element nie zostanie odczytany, to przyjmowane są wartości domyślne, czyli zera.

Ostatecznie dodajemy dane do bufora. Dane dodawane są w takiej kolejności, że pierw dodajemy współrzędne wierzchołka, następnie wektor normalny i współrzędne tekstury. Jeden wierzchołek tworzony jest zatem **przez 8 zmiennych typu zmiennoprzecinkowego float: 3 współrzędne położenia, 3 współrzędne wektora normalnego i 2 współrzędne tekstury**.

Na koniec obiegu pętli wczytywania modelu, zapisujemy liczbę wierzchołków aktualnie wczytanego modelu do tablicy `mesh_vertices_count`. Zwiększamy także całkowitą liczbę wczytanych wierzchołków (zmienna `total_vertices_count`). Do tablicy `mesh_starting_vertex_index` zapisujemy aktualny stan całkowitej liczby wczytanych wierzchołków. Będziemy tego potrzebowali niedługo podczas rysowania modelu.

```
mesh_vertices_count.push_back(mesh_vertices);
mesh_starting_vertex_index.push_back(total_vertices_count);
total_vertices_count += mesh_vertices;
```

Odczytywanie materiałów jest równie proste, co odczytywanie informacji o wierzchołkach. Na razie będziemy odczytywać tylko tekstury.

```
if (scene->mNumMaterials != 0)
{
    const aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];

    aiString texture_path;

    GLuint tex = 0;
    if (material->GetTexture(aiTextureType DIFFUSE, 0, &texture_path) == AI_SUCCESS)
    {
        unsigned int found_pos = file_name.find_last_of("//");
        std::string path = file_name.substr(0, found_pos);
        std::string name(texture_path.C_Str());
        if (name[0] == '/')
            name.erase(0, 1);

        std::string file_path = path + "/" + name;

        if (LoadTexture(file_path, tex))
            std::cout << "Texture " << file_path << " not found." << std::endl;
        else
            std::cout << "Texture " << file_path << " loaded." << std::endl;
    }

    textures.push_back(tex);
}
```

W pierwszej kolejności pobierany jest żądany materiał. Dostęp do wszystkich materiałów zdefiniowanych na scenie mamy dostęp przez pole `scene->mMaterials`, a indeks materiału przypisany do danego obiektu (`mesh`) można odczytać za pomocą `mesh->mMaterialIndex`.

Funkcja `GetTexture()` odczytuje teksturę danego obiektu i zwraca ścieżkę dostępu do niej. W pierwszym parametrze podajemy żądany typ tekstury, który chcemy pobrać (obiekt może mieć kilka rodzajów tekstur, np. mapy wysokości). W drugim parametrze natomiast podajemy indeks tekstury, którą chcemy pobrać. Do trzeciego parametru przekazujemy zmienną typu `aiString` i w wyniku otrzymujemy ścieżkę **względna** do pliku z tekstem.

Jeśli pobranie tekstury zakończy się sukcesem, to należy ją wczytać, tak jak robimy to zwykle, np. za pomocą biblioteki `FreeImage`. Te zabawy z łańcuchami znaków na początku instrukcji warunkowej `if` służą do ustalenia dokładnej lokalizacji tekstury. Nie zawsze ścieżka uzyskana w zmiennej `texture_path` jest poprawna. Można w łatwy sposób to sprawdzić wyświetlając ją sobie na ekranie.

```
std::cout << texture_path.C_Str() << std::endl;
```

Zmienna `tex` jest uchwytem do wczytywanej tekstury. Wewnątrz funkcji `LoadTexture()` następuje generacja nowej tekstury za pomocą funkcji `glGenTextures()`. Jeśli nie uda się odczytać tekstury, uchwyt ten będzie pusty, nie będzie związana z nim żadna tekstura. Ostatecznie wszystkie uchwyty dodawane są do tablicy dynamicznej `textures`, która była parametrem funkcji.

Wczytany wcześniej bufor wierzchołków możemy już przenieść do obiektu VAO.

```
GLuint vbo buffer = 0;
 glGenBuffers(1, &vbo buffer);
 glBindBuffer(GL_ARRAY_BUFFER, vbo_buffer);
 glBufferData(GL_ARRAY_BUFFER, buffer_vbo_data.size() * sizeof(GLfloat),
              buffer_vbo_data.data(), GL_STATIC_DRAW);

int single vertex size = 2 * 3 * sizeof(GLfloat) + 2 * sizeof(GLfloat);

glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, single vertex size, 0);
 glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, single vertex size,
                      reinterpret_cast<void*>(3 * sizeof(GLfloat)));
 glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, single vertex size,
                      reinterpret_cast<void*>(2 * 3 * sizeof(GLfloat)));
 glEnableVertexAttribArray(2);
```

Jak widzisz, wczytujemy cały bufor do jednego obiektu VBO. Może zastanawiasz się, czemu tak robimy. Czy nie byłoby wygodniej, jeśli mielibyśmy trzy oddzielne bufory VBO i każdy byłby na inny rodzaj danych? Otóż niekoniecznie. Przełączanie się pomiędzy buforami VBO jest kosztowne, a więc należy robić to jak najrzadziej. Tworzymy, zatem jeden bufor VBO i wrzucamy do niego wszystkie dane razem.

Ustawiając wskaźniki dla obiektu VAO określamy, że współrzędne wierzchołka zaczynają się od zerowego elementu i współrzędne kolejnego wierzchołka mają być pobierane skokowo, co wartość `single_vertex_size`. Wartość ta określa rozmiar pojedynczego wierzchołka, na którą składa się 8 zmiennych typu `float` (pisałem o tym już wyżej).

Współrzędne wektorów normalnych przesunięte są względem początku o 3 zmienne `float`, a współrzędne tekstury zaczynają się po 6 zmiennych `float`. Ostatnim parametrem funkcji `glVertexAttribPointer()` jest wskaźnik, a zatem konieczne jest przeprowadzenie rzutowania na typ wskaźnikowy.

Schematycznie wygląd bufora prezentuje rys. 26.2.

Wierzchołek nr 1								Wierzchołek nr 2			
P(X)	P(Y)	P(Z)	N(X)	N(Y)	N(Z)	T(X)	T(Y)	P(X)	P(Y)	P(Z)	...

Rys. 26.2. Wygląd bufora z wpisany danymi wierzchołków (P - współrzędne wierzchołka, N - współrzędna wektora normalnego, T - współrzędna tekstury).

Podkreślam jeszcze raz, korzystamy tylko z jednego bufora VBO w celach optymalizacji. Jeśli Ci na tym nie zależy, możesz stworzyć trzy bufory VBO i do pierwszego wpisać współrzędne wierzchołków, do drugiego współrzędne wektora normalnego, a do trzeciego współrzędne tekstuury.

Korzystając teraz z tej funkcji możemy wczytać jakiś model i go narysować:

```
GLuint vao1 = 0;
std::vector<GLfloat> vertices count;
std::vector<GLfloat> starting vertex;
std::vector<GLuint> textures;

LoadSceneFromFile("city/city.obj", vao1, vertices count, starting vertex, textures);

// ...
// Rysowanie
glBindVertexArray(vao1);
for (int i = 0; i < starting_vertex.size(); i++)
{
    glBindTexture(GL_TEXTURE_2D, textures[i]);
    glDrawArrays(GL_TRIANGLES, starting_vertex[i], vertices_count[i]);
}
```

Do naszego obiektu VAO wczytaliśmy wcześniej cały ciąg wierzchołków. Rysując, musimy wiedzieć, od którego wierzchołka zaczyna się kolejny model. Do tego celu wykorzystujemy właśnie tablicę `starting_vertex`, w której zapisywaliśmy numer wierzchołka, gdzie zaczynał się kolejny model. Do funkcji `glDrawArrays()` podajemy teraz w drugim parametrze numer wierzchołka, od którego ma zacząć rysowanie oraz w trzecim parametrze podajemy ile wierzchołków ma narysować.

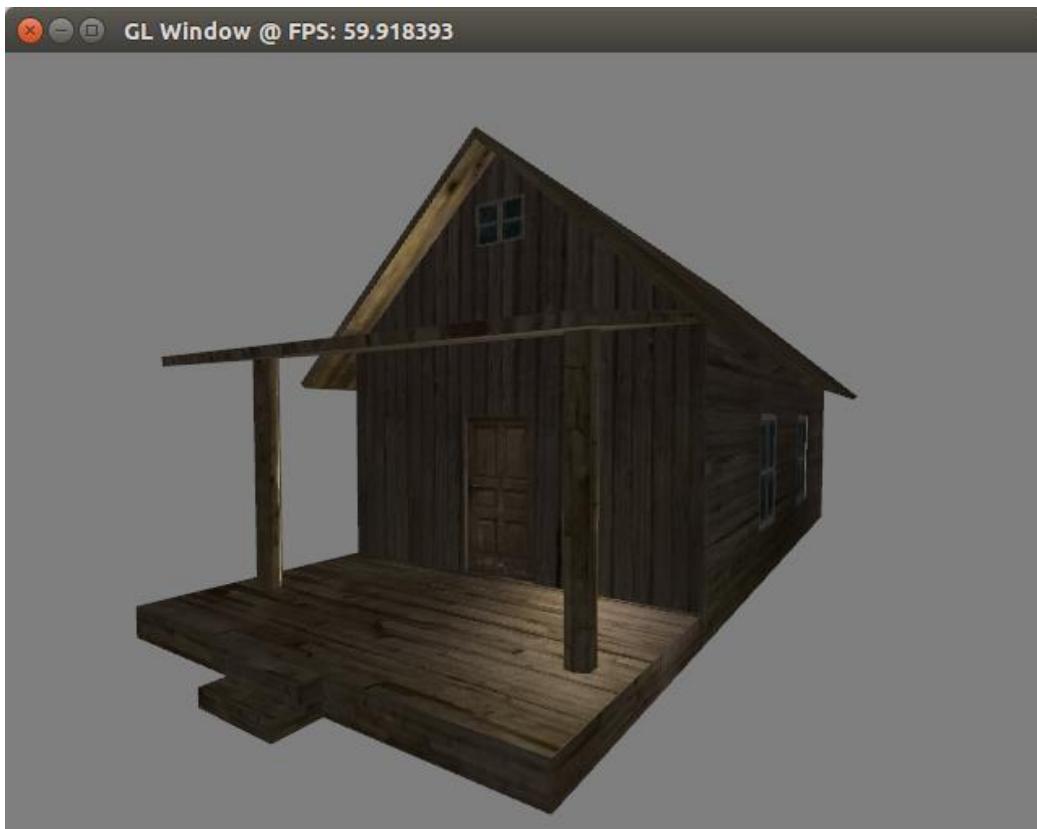
Wczytywanie modeli ogólnie jest skomplikowanym elementem. Nie zawsze pliki z modelami zapisane są poprawnie, a to czasem ścieżka do tekstuury jest źle odczytywana. Trzeba trochę cierpliwości oraz debugowania, aby osiągnąć zamierzony cel. Przeanalizuj wszystko samemu. Spróbuj nawet napisać własną funkcję ładującą modele, wtedy najbardziej zrozumiesz cały ten proces.

Mogemy teraz potestować wczytywanie różnych modeli (rys. 26.3).

Na początku wspomniałem o tym, że w internecie znajdziemy dużo bezpłatnych modeli. Najbardziej polecam stronę <http://tf3dm.com/>. Zawiera całkiem bogatą bazę modeli, a w dodatku mamy możliwość wyboru żądanego formatu pliku.

Kod źródłowy z tego rozdziału jest w katalogu: 16_Wczytywanie_modeli

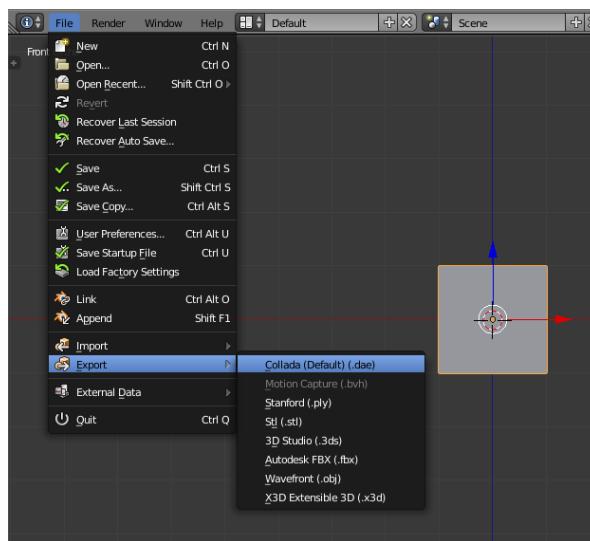
Przykładowy model dostępny jest na stronie „Pobierz”.



Rys. 26.3. Wczytany przykładowy model wraz z teksturami i oświetleniem.

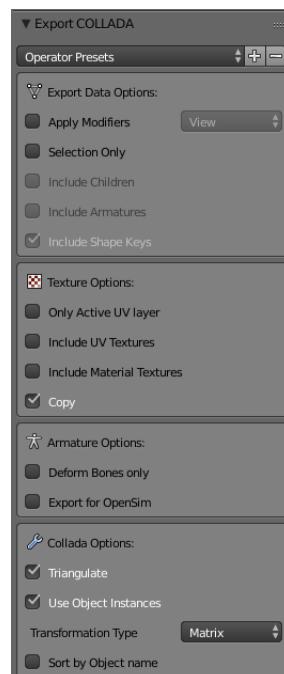
Tips & Tricks:

- Należy pamiętać o tym, że jeśli wczytujemy duże modele (posiadające dużo wierzchołków), proces wczytywania może trwać parę chwil. Jeśli program nie reaguje, to nie znaczy, że się zawiesił, może po prostu trwa właściwe wczytywanie modelu.
- Wczytywanie modeli często może powodować różne problemy. Należy wtedy na spokojnie poszukać przyczyny. Przede wszystkim sprawdzić, czy najprostszy model z sześcianem wczytuje się poprawnie.
- Jeśli umiesz posługiwać się programem Blender, w celu tworzenia swoich własnych modeli, pokażę jak możesz eksportować je do formatu Collada:
 1. Uruchamiamy Blender.
 2. Przeprowadzamy podział modelu na trójkąty: w trybie Edit Mode przechodzimy do menu Mesh > Faces > Triangulate Faces. Obiekt musi być oczywiście pierw zaznaczony.
 3. Wchodzimy w menu File > Export > Collada (rys. 26.4).



Rys. 26.4. Eksport modelu do pliku Collada.

4. Upewniamy się, że ramka po lewej stronie okna wygląda tak jak na rys. 26.5.

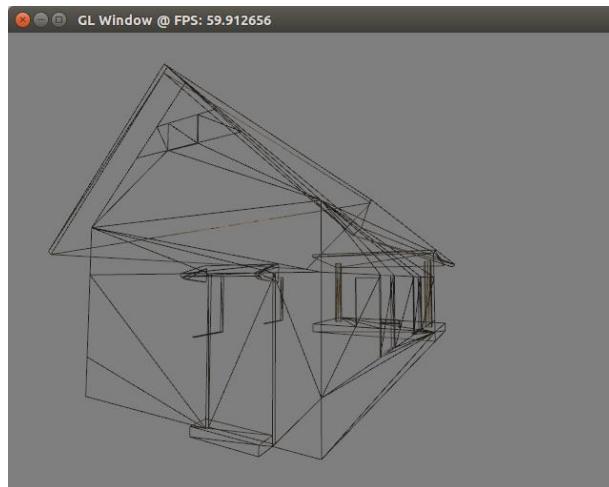


Rys. 26.5. Konfiguracja eksportu pliku Collada.

5. Nadajemy plikowi nazwę i naciskamy przycisk Export COLLADA.

- Program Blender możemy także wykorzystać do podziału na trójkąty już gotowe modele OBJ.
 1. Uruchamiamy Blender.
 2. Jeśli mamy domyślnie już na ekranie model sześcianu to kasujemy go.
 3. Wchodzimy w menu File > Import > Wavefront (.obj).
 4. Wybieramy żądany plik.
 5. Po wczytaniu modelu, zaznaczamy go i wchodzimy w menu File > Export > Wavefront (.obj).

6. W ramce Export OBJ po lewej stronie zaznaczamy opcję Triangulate Faces.
 7. Nadajemy nazwę i klikamy Export OBJ.
- Aby nie wypełniać powierzchni pomiędzy wierzchołkami, a rysować jedynie same wierzchołki i łączące je linie, należy użyć funkcji `glPolygonMode(GL_FRONT, GL_LINE)`. Rysowana będzie tylko siatka trójkątów (rys. 26.6).

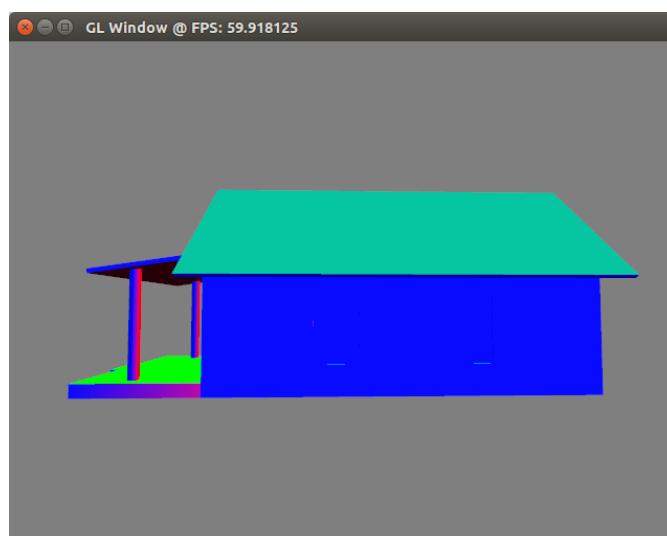


Rys. 26.6. Narysowana siatka trójkątów.

- Jeśli chcemy sprawdzić, czy wektory normalne są prawidłowo skierowane, możemy w fragment shader wprowadzić pewną małą modyfikację:

```
frag_colour = vec4(normal_to_camera, 1.0);
```

W ten sposób kolor powierzchni będzie zależał od jej wektora normalnego. Jeśli powierzchnia będzie pokolorowana na niebiesko, oznacza to, że jej wektor normalny jest skierowany poprawnie, gdyż $RGB = (0, 0, 1)$ i kierunek osi Z kamery pokrywa się z kierunkiem wektora normalnego (rys. 26.7).



Rys. 26.7. Powierzchnie pokolorowane w zależności od ustawienia wektora normalnego względem kamery.

27. Mgła

Fajnie byłoby dodać do sceny jakiś prosty efekt graficzny, żeby wszystko wyglądało ciekawiej. W tym rozdziale zajmiemy się więc mgłą. Nic nie dodaje takiej atmosfery i tajemniczości jak mgła.

Mgła wbrew pozorom jest bardzo łatwym do zrobienia efektem graficznym. Ponadto, nie jest wymagająca obliczeniowo, nie obciążymy nią naszego sprzętu. Mgłę można także śmiało wykorzystać w przypadku, gdy odcinamy rysowanie powyżej pewnej odległości od kamery. Żeby zniwelować nieprzyjemny dla oka efekt nagłego zanikania obiektów, można wprowadzić mgłę, która stopniowo „pochłonie” dalsze, niewidoczne (bo nierenderowane) obszary.

Będziemy zajmować się mgłą liniową, tzn. wzrost gęstości mgły zależy liniowo od odległości od kamery. Zabawy w nieliniowe funkcje logarytmiczne albo wykładnicze nie mają do końca sensu, gdyż i tak oko ludzkie nie zauważa zbytniej różnicy.

Dodanie efektu mgły opiera się tylko na modyfikacji programu fragment shader. No chyba, że mamy zamiar przesyłać różne zmienne typu `uniform` parametryzujące mgłę z aplikacji, wtedy musimy to uwzględnić także w kodzie aplikacji.

Mgłę definiujemy poprzez określenie trzech parametrów:

- kolor mgły,
- w jakiej odległości od kamery mgła ma zacząć się pojawiać,
- w jakiej odległości od kamery mgła ma przyjmować wartość maksymalną – przenikanie w pełni w kolor mgły.

Zaczniemy od zdefiniowania tych parametrów w kodzie shadera:

```
vec3 fog colour = vec3(0.5, 0.5, 0.5);
float min_fog_distance = 5.0;
float max_fog_distance = 80.0;
```

Następnie musimy obliczyć odległość danego renderowanego wierzchołka od kamery. Potrzebujemy uzyskać jedną wartość zmiennoprzecinkową, a zatem korzystamy z funkcji `length()`, która wykorzystuje twierdzenie Pitagorasa w 3D do obliczenia odległości.

```
float distance = length(-vertex_to_camera);
```

Teraz już możemy obliczyć współczynnik mgły:

```
float fog_factor = (distance - min_fog_distance) / (max_fog_distance - min_fog_distance);
```

Jeśli wartość zmiennej `distance` jest większa niż `max_fog_distance`, to współczynnik ten jest wtedy większy od 1. Natomiast w przypadku, gdy `distance` jest mniejsze niż `min_fog_distance`, to współczynnik ten przyjmuje wartość ujemną. Żeby ograniczyć wartość tego współczynnika do przedziału od 0 (zerowy poziom mgły) do 1 (maksymalny

poziom mgły) należy skorzystać z funkcji `clamp()`. Ogranicza ona podaną wartość w określonych przedziałach.

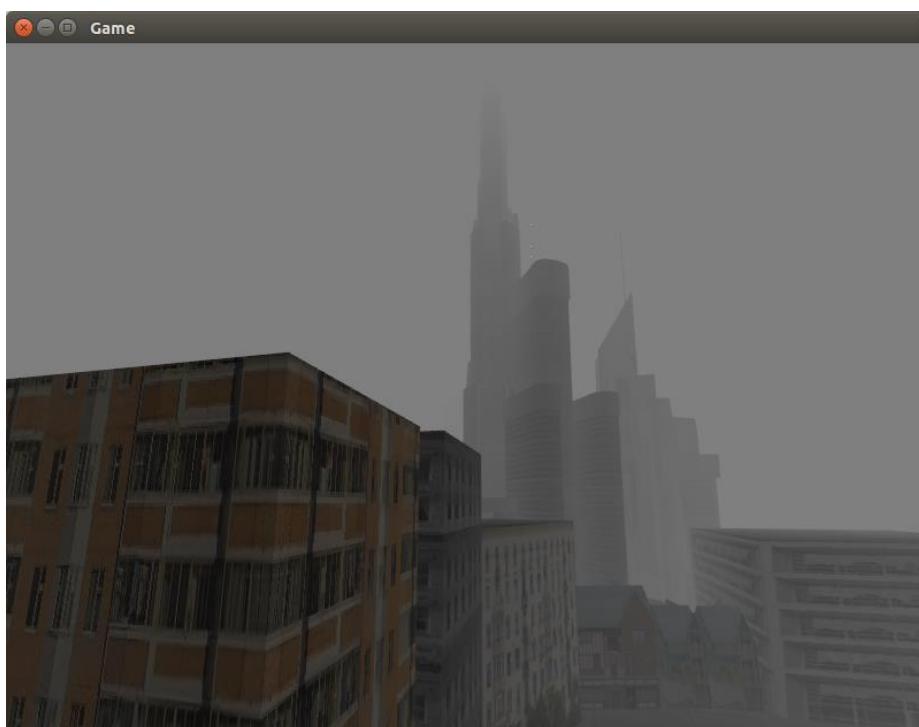
```
fog_factor = clamp(fog_factor, 0.0, 1.0);
```

I to tyle obliczeń. Teraz musimy uzyskany współczynnik uwzględnić podczas obliczania wynikowego koloru fragmentu. Chcemy zmieszać tak naprawdę dwa kolory: pierwszy to kolor wynikający z tekstury oraz oświetlenia, a drugi to kolor mgły. Do tego celu można użyć funkcji `mix()`. Przyjmuje ona dwa kolory składowe oraz parametr ustalający, w jakim stopniu należy te kolory wymieszać.

```
vec4 texel = texture(basic_texture, texture coordinates);
vec4 light_and_texture = vec4(ambient_intense + diffuse_intense + specular_intense, 1.0) *
texel;
frag_colour.rgb = mix(light_and_texture.rgb, fog_colour, fog_factor);
```

Jeśli `fog_factor` przyjmie wartość 0, to wtedy wynikowy kolor będzie w pełni równy kolorowi zmiennej `light_and_texture`. W przypadku współczynnika `fog_factor` równego 1, wynikowy kolor będzie w pełni kolorem mgły `fog_colour`.

Dzięki temu można uzyskać różne ciekawe efekty. Na przykład takie ponure miasto we mgle jak na rys. 27.1.



Rys. 27.1. Nałożony efekt mgły.

Kod źródłowy z tego rozdziału jest w katalogu: **17_Mgla**

28. Multi-teksturowanie

Powierzchnie nie zawsze muszą mieć przypisaną tylko jedną teksturę. Przykładowo, możemy wyobrazić sobie drogę asfaltową, która przy swoich krawędziach pokryta jest piaskiem, tak jak to jest w świecie realnym. Bierzemy zatem teksturę drogi asfaltowej oraz teksturę piasku, mieszamy je ze sobą i w łatwy sposób otrzymujemy efekt zakurzonej drogi. W tym krótkim rozdziale pokażę jak tego dokonać.

Mutli-teksturowanie polega na tym, że dana powierzchnia może posiadać więcej niż jedną teksturę. Tekstury te, można wtedy ze sobą dowolnie mieszać. Można powiedzieć, że są trzy „stopnie wymieszania”:

- powierzchnia w całości pokryta tekstem pierwską,
- powierzchnia pokryta dwoma teksturami, gdzie wzajemny stopień pokrycia tekstu wyrażony jest poprzez współczynnik,
- powierzchnia w całości pokryta tekstem drugą.

Mieszania tekstur dokonujemy w fragment shader. Oczywiście na początku w kodzie programu musimy te tekstury załadować i wrzucić je do oddzielnych slotów (szufladek). Robimy to w sposób następujący:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, first_texture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, second_texture);
```

Za pomocą funkcji `glActiveTexture()` wybieramy aktywną kieszonkę na teksturę, a następnie za pomocą funkcji `glBindTexture()` wpisujemy podaną teksturę do tej aktualnie aktywnej kieszonki.

Dalej musimy rozszerzyć nasz fragment shader o możliwość przyjmowania drugiej tekstury. W tym celu należy dodać kolejną zmienną `uniform sampler2D`:

```
uniform sampler2D first_texture;
uniform sampler2D second_texture;
```

Nie pozostaje teraz już nam nic innego jak ustawienie tego, aby zmienna (`sampler`) `first_texture` odczytywała teksturę ze slotu numer 0, a zmienna `second_texture` ze slotu numer 1.

```
GLint texture_slot_0 = glGetUniformLocation(shaders, "first_texture");
glUniform1i(texture_slot_0, 0);
GLint texture_slot_1 = glGetUniformLocation(shaders, "second_texture");
glUniform1i(texture_slot_1, 1);
```

Do mieszania tekstur wykorzystamy znaną już funkcję `mix()`. Wykonuje ona liniowe mieszanie dwóch kolorów. W rozdziale na temat mgły wykorzystywaliśmy ją w celu zmieszania koloru tekstury z kolorem mgły.

```
#version 330
in vec2 texture_coordinates;
```

```
uniform sampler2D first_texture;
uniform sampler2D second_texture;

out vec4 frag colour;

void main()
{
    vec4 first_sample = texture(first_texture, texture_coordinates);
    vec4 second_sample = texture(second_texture, texture_coordinates);

    frag_colour = mix(first_sample, second_sample, texture_coordinates.s);
}
```

Jak widzimy, współczynnikiem w funkcji `mix()` jest wartość współrzędnej S tekstury. Dzięki temu, uzyskujemy efekt przenikania tekstur od lewej do prawej, zgodnie z narastaniem współrzędnej S. Jeśli wpisalibyśmy stałą wartość współczynnika, np. 0.6, to mieszanie kolorów tekstur nastąpiłoby po całości powierzchni, nie wystąpiłby efekt stopniowego przenikania (rys. 28.1).



Rys. 28.1. Wynik mieszanego teksturowania.

Multi-teksturowanie, jak pisałem na początku, można wykorzystać także do przełączania tekstur. Czasem jakaś powierzchnia pokryta jest jedną tekstonią, a czasem drugą. Najczęściej wykorzystywane to jest, gdy jakiś obiekt w grze zmienia swój stan. Na przykład, tektonura szyby zmienia się w popękaną tektonurę szyby, albo lakier samochodu zmienia się w porysowany lakier samochodu. Kwestia wyobraźni.

Kod źródłowy z tego rozdziału jest w katalogu: 18_Multiteksturowanie

29. Współczynnik odbicia światła na podstawie tekstu

Nie zawsze ładujemy tekstury po to, aby „kolorować” nimi obiekt trójwymiarowy. Nie musimy przecież odczytanego z tekstu koloru teksta, wykorzystywać do pomalowania odpowiedniego fragmentu. Odczytana wartość koloru może postużyć w zupełnie innym celu, np. do obliczenia współczynnika odbicia światła dla danego fragmentu.

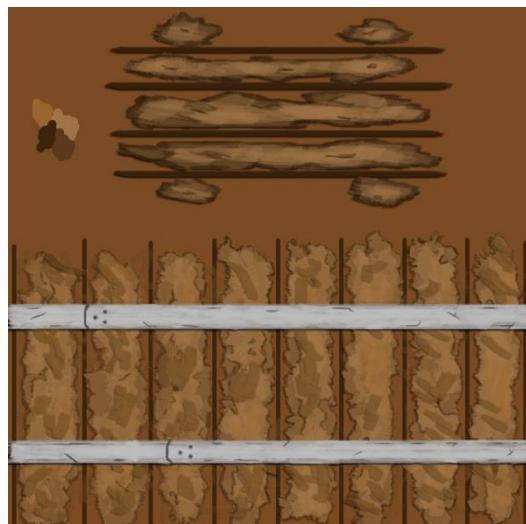
W rozdziale na temat oświetlenia, wspominałem już, że obiekt trójwymiarowy możemy scharakteryzować trzema współczynnikami odbicia światła. Określamy nimi po prostu, jakie składowe światła mają zostać odbite od obiektu i w jakim stopniu. Najczęściej ustawialiśmy takie same współczynniki dla trzech składowych, np. (0.8, 0.8, 0.8), a zatem wszystkie składowe światła były odbijane z jednakową intensywnością.

Co zrobić natomiast wtedy, gdy chcemy, aby dany obiekt posiadał obszary matowe nieodbijające światła oraz obszary metaliczne w pełni odbijające światło? Na przykład, taka drewniana beczka. Zbudowana jest z drewna i przepasana jest metalowymi obręczami. Założmy też, że model beczki jest spójny, tzn. niezbudowany jest z odrębnego modelu samej drewnianej beczki oraz odrębnego modelu metalowych pasów. Jeśli w tym przypadku zdefiniujemy współczynnik odbicia światła dla tego obiektu, to każdy element beczki będzie odbijał światło z taką samą intensywnością. Drewniane deski będą odbijać światło tak samo jak metalowe, co nie będzie wyglądać naturalnie.

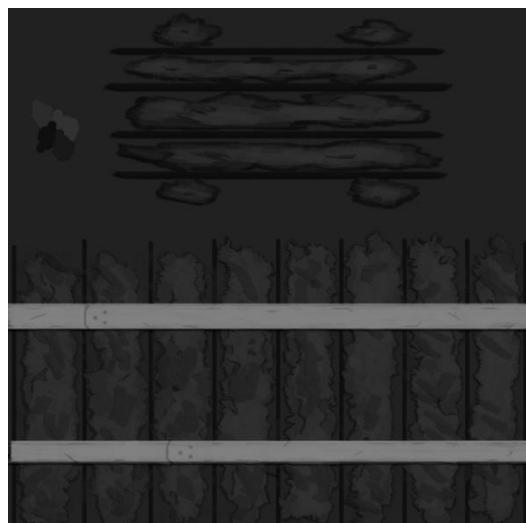
Aby ten problem rozwiązać, należy zastosować co najmniej dwie tekstury. Jedną dla światła rozproszonego (diffuse), a drugą dla światła odbitego zwierciadlanie (specular). Dzięki temu, **każdy fragment będzie miał swoje indywidualne współczynniki odbicia**.

W naszym programie musimy na początku wczytać wierzchołki modelu (bądź wpisać ręcznie) oraz wczytać także dwie tekstury (rys. 29.1 oraz rys. 29.2).

Teksturę dla światła rozproszonego (rys. 29.1) będziemy wykorzystywać w celu kolorowania fragmentów, prawie tak samo jak robiliśmy to do tej pory. Natomiast teksturę dla światła odbitego (rys. 29.2), będziemy wykorzystywać w celu obliczenia współczynnika odbicia dla tego rodzaju światła. Patrząc się na tą tekstrę, można zauważyc, że została ona przyciemniona w porównaniu z tekstrą dla światła rozproszonego oraz metalowe obręcze zostały rozjaśnione. **Współczynnik odbicia światła będzie równy składowym koloru danego fragmentu.** I tak, elementy ciemne, będą miały bardzo mały współczynnik odbicia (kolor czarny ma trzy składowe równe 0), a elementy jasne, będą miały większy współczynnik odbicia (kolor biały ma trzy składowe równe 1).



Rys. 29.1. Tekstura dla światła rozproszonego (diffuse).



Rys. 29.2. Tekstura dla światła odbitego zwierciadlanie (specular).

A zatem, używając tekstury dla światła odbitego, mamy kontrolę nad tym, które elementy mają błyszczeć, a które mają być matowe. Ponadto, możemy kontrolować kolor odbitego światła. W powyższym przykładzie, tekstura jest w skali szarości, czyli będzie odbijała każdą składową światła z taką samą intensywnością. Nie stoi jednak nic nie przeszkodzie, aby ta tekstura była kolorowa i żebyśmy pokolorowali niektóre elementy, np. na czerwono, żeby odbijały tylko kolor czerwony.

Obliczeń oświetlenia oraz nakładania tekstur dokonujemy w fragment shader, a więc tylko tam czekają nas zmiany. Przede wszystkim, musimy móc obsługiwać dwie tekstury:

```
uniform sampler2D diffuse_texture;
uniform sampler2D specular_texture;
```

Musimy także mieć parametry źródła światła, takie jak położenie oraz kolor poszczególnych składowych:

```
vec3 light position = vec3(5.0, 6.0, 0.0);
vec3 ambient_color = vec3(0.1, 0.1, 0.1);
```

```
vec3 diffuse_color = vec3(0.8, 0.8, 0.8);
vec3 specular_color = vec3(1.0, 1.0, 1.0);
```

Dalej w funkcji `main()` dokonujemy obliczeń światła rozproszonego:

```
// Diffuse
vec3 distance_to_light = vec3(view matrix * vec4(light position, 1.0)) - vertex to camera;
vec3 light direction = normalize(distance to light);
float dot_product = dot(light direction, normal to camera);
dot_product = max(dot_product, 0.0);
vec4 texture_sample = texture(diffuse_texture, texture_coordinates);
vec3 Kd = texture sample.rgb;
vec3 diffuse_intense = diffuse_color * Kd * dot_product;
```

Większość z tych obliczeń już znamy. Nowość zaczyna się dopiero od zmiennej `texture_sample`. Pobieramy do niej kolor z aktualnych współrzędnych tekstuury `diffuse_texture`. Dalej, do zmiennej `Kd` przypisujemy składową `rgb` zmiennej `texture_sample`. Zmienna `Kd` jest typu `vec3`, gdyż `rgb` posiada także trzy składowe. Zmienna `Kd` jest współczynnikiem odbicia światła (a zarazem kolorem) danego fragmentu, a więc ostateczna intensywność światła rozproszonego na aktualnym fragmencie jest równa iloczynowi kolorowi światła, kolorowi danego fragmentu wynikającego z tekstuury oraz „intensywności” dochodzącego światła do tego fragmentu.

Identycznie rzecz się ma w przypadku obliczenia światła odbitego zwierciadlanie.

```
// Specular
vec3 reflection = reflect(-light direction, normal to camera);
vec3 surface_to_camera = normalize(-vertex to camera);
float dot_specular = dot(reflection, surface_to_camera);
dot_specular = max(dot_specular, 0.0);
float specular_power = 10.0;
float specular_factor = pow(dot_specular, specular_power);
vec3 Ks = texture(specular texture, texture coordinates).rgb;
vec3 specular_intense = specular_color * Ks * specular_factor;
```

Zostało już tylko na sam koniec zsumować dwie uzyskane intensywności.

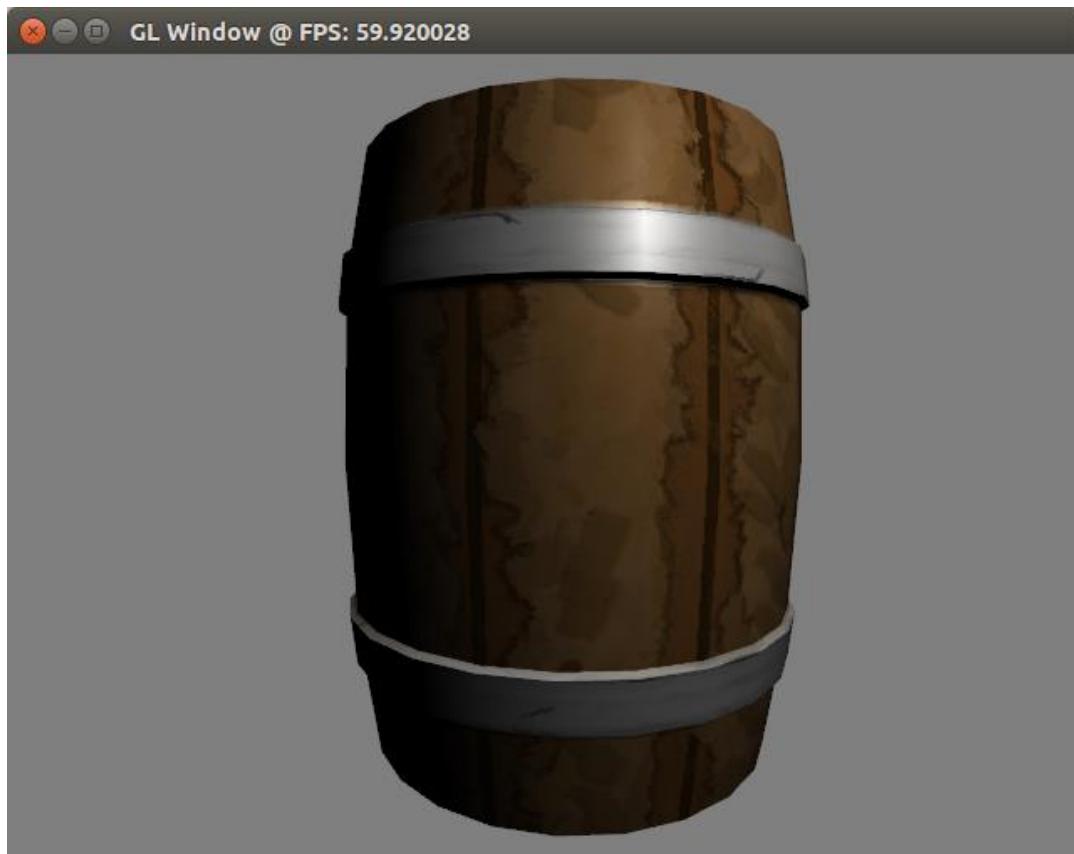
```
frag_colour = vec4(diffuse_intense + specular_intense, 1.0);
```

W wyniku uzyskamy efekt jak na rys. 29.3. Widać, że metalowa obręcz błyszczy się w świetle, a matowe drewno nie odbija zwierciadlanie światła.

W przypadku światła otoczenia można zrobić tak samo jak w przypadku światła otoczenia (ambient). Pobrać tą samą tekstrurę oraz przemnożyć ją przez kolor tego rodzaju światła. Na końcu uzyskaną intensywność, dodać do pozostałych składowych:

```
// Ambient
vec4 texture sample = texture(diffuse texture, texture coordinates);
vec3 Ka = texture sample.rgb;
vec3 ambient intense = ambient color * Ka;
// ...
frag_colour = vec4(diffuse_intense + specular_intense + ambient_intense, 1.0);
```

Jednak z reguły robi się to inaczej. Stosuje się, tzw. [ambient occlusion maps](#). Jest to tekstura uwzględniająca zacienienie powierzchni. Przygotowuje się ją z reguły w programach służących do modelowania 3D poprzez „wypiekanie” (bake). Jest to osobne skomplikowane zagadnienie zasługujące na odrębny rozdział.



Rys. 29.3. Rezultat działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: **19_Wspolczynniki_odbicia**

Przykładowy model dostępny jest na stronie „[Pobierz](#)”.

30. Odrzucanie fragmentów

Wyświetlając jakąś teksturę, OpenGL (a dokładniej GLSL) umożliwia nam pominięcie rysowania jakiegoś koloru. Korzystamy najczęściej z takiej możliwości wtedy, gdy chcemy uzyskać w prosty sposób przezroczystość tekstuury w pewnych jej miejscach.

Wyobraźmy sobie sytuację, że modelujemy drzewo. Jeśli chcielibyśmy dokładnie zamodelować każdy liść, otrzymalibyśmy mnóstwo wierzchołków. Dla każdego wierzchołka obliczane jest przecież także oświetlenie, więc im więcej wierzchołków do obliczenia, tym większe obciążenie komputera. Można zatem w takiej sytuacji posłużyć się pewną sztuczką. Każdy liść modelowany jest jako prostokąt posiadający cztery wierzchołki, a następnie nakładana jest na niego tekstura, np. taka jak na rys. 30.1.



Rys. 30.1. Przykładowa tekstura liścia.

Dzięki temu redukujemy ilość wierzchołków do minimum. Wystarczy, że podczas renderowania, nie będziemy rysować tła otaczającego liść, a wszystko będzie wyglądało w porządku.

Zacznijmy od wyświetlenia tekstuury na ekranie (rys. 30.2).



Rys. 30.2. Wyświetlona tekstura razem z tłem.

Na początku musimy ustalić kolor tła tekstuury. W tym celu otwieramy tekstuwę w jakimś programie graficznym, może być nawet MS Paint i sprawdzamy kolor tła. W powyższym

przypadku jest to $\text{RGB} = (128, 135, 101)$. Po przeliczeniu tych składowych na przedział od 0.0 do 1.0, wartości te będą wynosić kolejno około (0.50, 0.53, 0.40).

Przechodzimy teraz do edycji fragment shader, gdyż tylko w nim mamy możliwość działania na teksturach. Wpisujemy taki kod:

```
void main()
{
    vec4 texel = texture(basic texture, texture coordinates);

    if (texel.r > 0.46 && texel.r < 0.54 &&
        texel.g > 0.49 && texel.g < 0.57 &&
        texel.b > 0.36 && texel.b < 0.44)
    {
        discard;
    }

    frag colour = texel;
}
```

Jak widać, na początku pobieramy próbkę koloru z tekstury, a następnie badamy jej kolor. Kolor tła jest równy około (0.50, 0.53, 0.40), ale na pewno nie będzie to taka sama wartość na całej powierzchni tekstury. Dlatego musimy dodać pewien margines. W tym przypadku jest to 0.04 dla każdej składowej. Najlepiej jest dobrać ten margines doświadczalnie, gdyż w różnych teksturach będą to różne wartości.

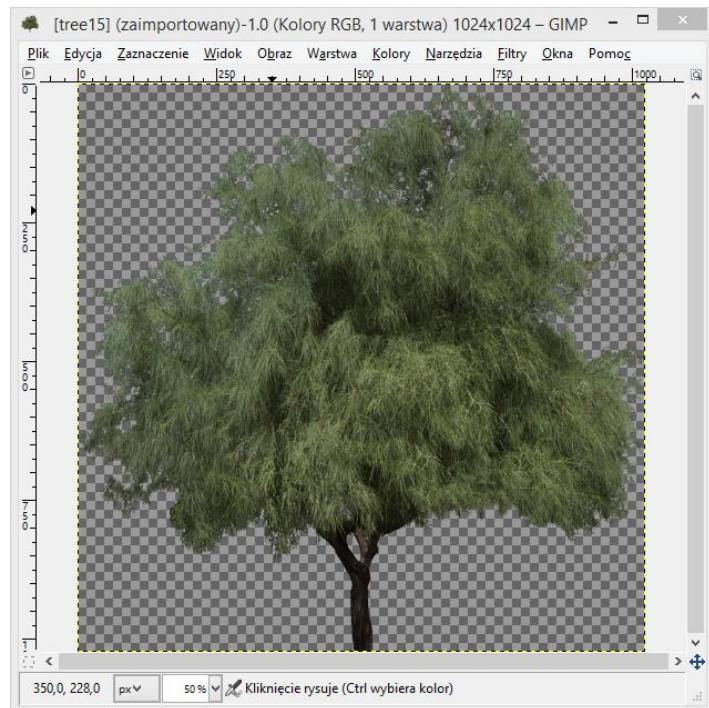
W celu pominięcia aktualnego przetwarzanego fragmennu korzystamy z słowa kluczowego `discard`. Powoduje ono, że aktualny fragment jest ignorowany i nie jest przekazywany dalej w potok graficzny.

Teraz możemy uruchomić już program i zobaczyć rezultaty (rys. 30.3).



Rys. 30.3. Tekstura z usuniętym tłem.

Czasem tło tekstury zamiast jednolitego koloru, jest po prostu przezroczyste (rys. 30.4). Taka tekstura posiada wtedy kanał alfa. W przypadku nie uwzględniania przezroczystości tekstur podczas rysowania, uzyskamy efekty jak na rys. 30.5.



Rys. 30.4. Tekstura z przezroczystym tłem.



Rys. 30.5. Błędnie wyświetcone tekstury razem z tłem.

Wystarczy, że w tym przypadku dodamy małą modyfikację do fragment shader:

```
if (texel.a < 0.9)
{
    discard;
}
```

Po tej zmianie już wszystko powinno być w porządku (rys. 30.6).



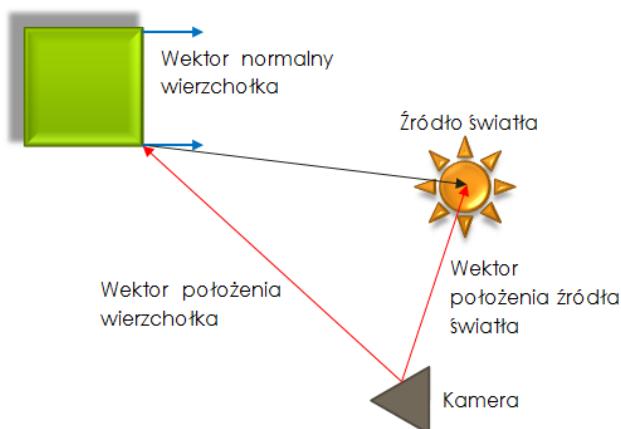
Rys. 30.6. Usunięcie kanału alfa z tekstur.

Kod źródłowy z tego rozdziału jest w katalogu: **20_Odrzucanie_fragmentow**

31. Kierunkowe źródło światła - reflektor

Do tej pory używaliśmy tylko punktowego źródła światła. Działa ono jak klasyczna żarówka, czyli światło emitowane jest w każdym kierunku jednakowo. W życiu często jednak mamy także do czynienia z kierunkowymi źródłami światła, takimi jak latarki oraz reflektory. W tym rozdziale pokażę jak zaimplementować takie źródło światła w OpenGL.

Przypomnijmy sobie jak to było w przypadku oświetlenia Phonga. Dla światła rozproszonego porównywaliśmy dwa wektory: wektor normalny wierzchołka oraz wektor kierunku światła wychodzący z tego wierzchołka (rys. 31.1). W zależności od iloczynu skalarnego tych dwóch wektorów, otrzymywaliśmy stopień intensywności światła rozproszonego na danym wierzchołku.



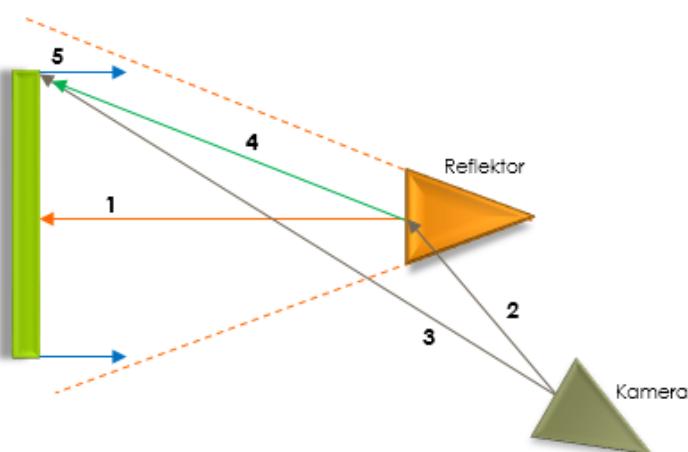
Rys. 31.1. Schematyczne ułożenie wektorów podczas obliczania światła rozproszonego.

W przypadku kierunkowego źródła światła, także będziemy porównywać dwa wektory. Cechą charakterystyczną tego rodzaju światła jest to, że świeci **tylko w ustalonym kierunku** (jak latarka), a zatem musimy mieć wektor kierunku tego światła. Drugim wektorem będzie natomiast wektor zaczepiony w tym źródle światła i wskazujący kierunek do aktualnego wierzchołka (rys. 31.2). Wektory te będziemy oczywiście porównywać ze sobą za pomocą iloczynu skalarnego, tak jak robiliśmy to poprzednio.

Przejdźmy do fragment shader i zacznijmy kodować:

```
vec3 spot position = vec3(0.0, 5.0, 30.0);
vec3 spot position camera = vec3(view matrix * vec4(spot position, 1.0));
vec3 distance from spot camera = vertex to camera - spot position camera;
vec3 direction from spot camera = normalize(distance from spot camera);
```

Na początku określamy położenie reflektora (zmienna `spot_position`) w układzie globalnym, a następnie musimy przeliczyć je na układ współrzędnych kamery (zmienna `spot_position_camera`). Przypomnę, że wszystkie wektory musimy mieć w jednym wspólnym układzie współrzędnych. Z reguły jest to układ kamery albo układ świata. Proponuję dodawać do zmiennych wektorowych przyrostki określające układ współrzędnych, w którym ten wektor jest wyrażony, np. `_world` lub `_camera`.



1 – Wektor kierunku światła, 2 – Wektor położenia źródła światła w układzie współrzędnym kamery, 3 – Wektor położenia wierzchołka w układzie współrzędnym kamery, 4 – Wektor kierunku do wierzchołka, 5 – Wektor normalny

Rys. 31.2. Prezentacja wektorów oświetlenia kierunkowego.

Dalej obliczamy wektor wychodzący z źródła światła i wskazujący kierunek do aktualnie przetwarzanego wierzchołka (wektor 4, rys. 31.2). Wystarczy, że obliczymy różnicę pomiędzy wektorem położenia wierzchołka (3), a wektorem położenia źródła światła (2). Następnie należy otrzymany wektor znormalizować, aby otrzymać wektor jednostkowy.

Wektor kierunku światła (1) określamy za pomocą jednej zmiennej wektorowej, a następnie przeliczamy go na układ współrzędnych kamery. Przypominam o tym, że przeliczamy kierunek, a zatem ostatnią wartością wektora czterowymiarowego musi być 0.

```
vec3 target = normalize(vec3(0.0, 0.0, -1.0));
vec3 target_camera = vec3(view_matrix * vec4(target, 0.0));
```

Potrzebujemy jeszcze dwóch zmiennych. Przede wszystkim koloru emitowanego światła oraz kąta rozwarcia stożka emitowanego światła (rys. 31.3).

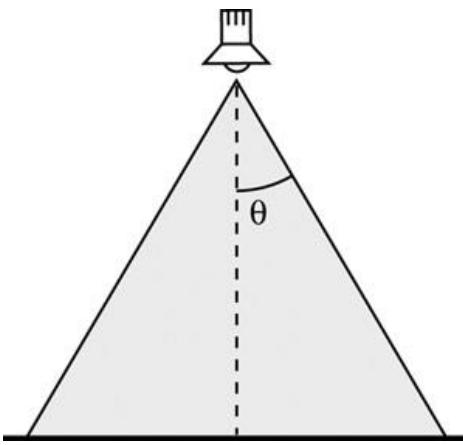
```
vec3 spot_colour = vec3(0.7, 0.7, 0.7);
float spot_arc = 1.0 - 30.0 / 90.0;
```

W tym przypadku jest to kąt 30° . Zmienna `spot_arc` będzie równa ok. 0.67, a zatem jeśli iloczyn wektorowy będzie mniejszy od tej wartości, to znaczy, że ten wierzchołek nie jest już objęty tym źródłem światła.

Mogliśmy teraz już obliczyć iloczyn wektorowy:

```
float spot_dot = dot(target_camera, direction_from_spot_camera);
```

Gdy już mamy to wszystko, możemy obliczyć współczynnik źródła światła. Będzie potrzebny on w celu obliczenia wynikowej intensywności tego źródła światła.



Rys. 31.3. Stożek oświetlenia.

Najprostszą metodą jest sprawdzanie, czy iloczyn wektorowy jest większy bądź mniejszy od podanego kąta stożka oświetlenia, o czym pisałem już wyżej.

```
float spot factor = 1.0;
if (spot_dot < spot_arc)
{
    spot_factor = 0.0;
}
```

Na sam koniec należy obliczyć wynikową wartość oświetlenia. Poniższy przykład pokazuje także, jak uwzględnić oświetlenie Phonga.

```
vec4 texel = texture(basic texture, texture coordinates);
vec4 phong intense = vec4(ambient intense + diffuse intense + specular intense, 1.0);
vec4 spot intense = vec4(spot factor * spot colour, 1.0);
frag_colour = (phong_intense + spot_intense) * texel;
```

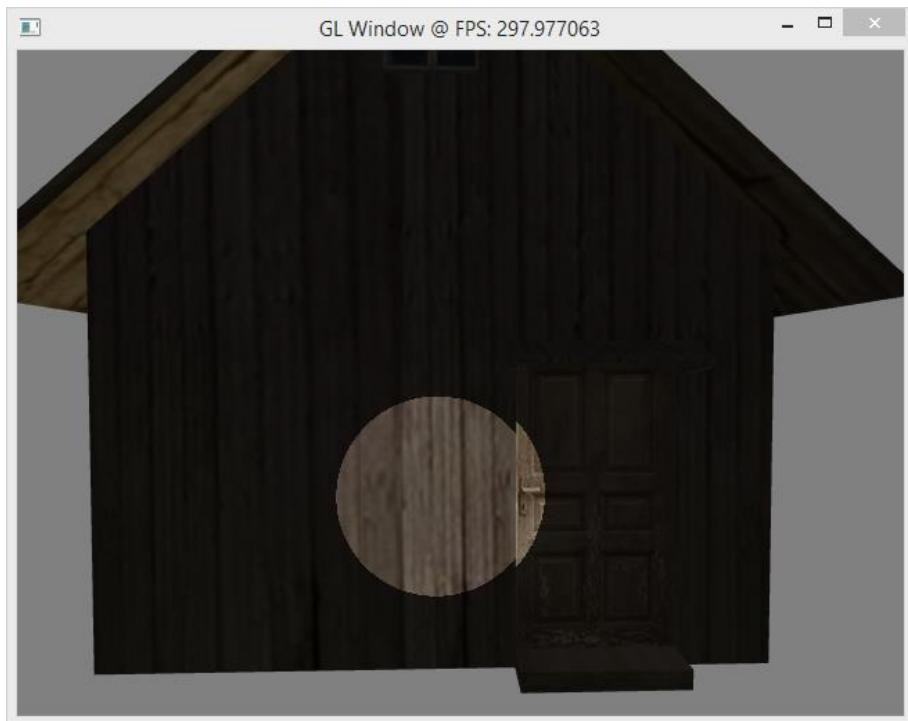
W wyniku otrzymamy taki efekt jak na rys. 31.4. Widoczne jest ostre przejście z strefy oświetlonej do strefy zacienionej.

Te ostre „krawędzie” można udoskonalić. Wystarczy zmodyfikować metodę obliczania współczynnika oświetlenia. Możemy zrobić tak, aby największa intensywność była w punkcie centralnym okręgu i żeby malała wraz z oddalaniem się od środka.

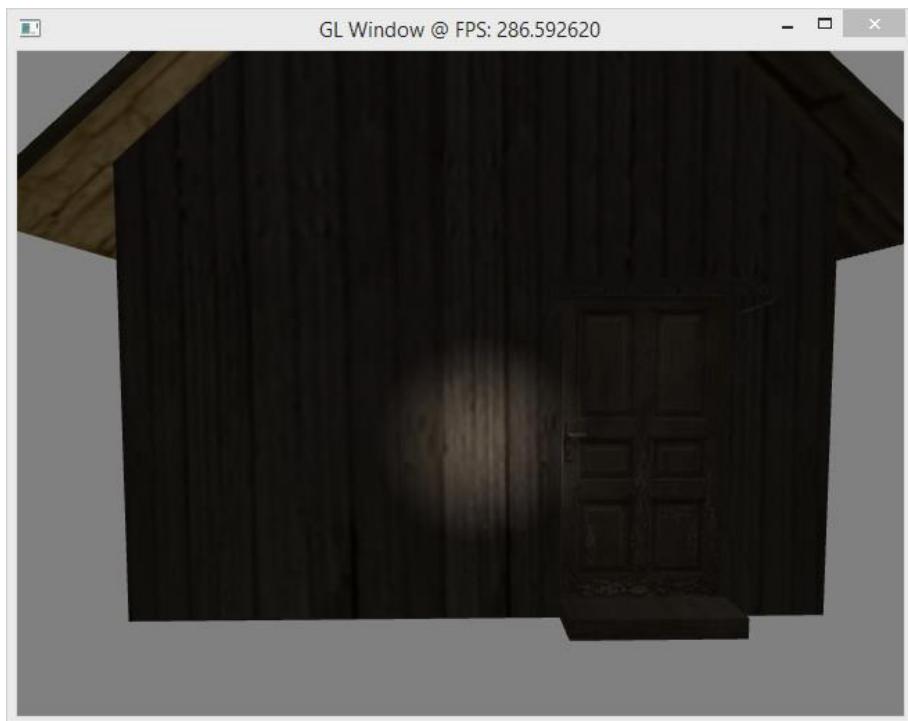
```
float spot factor = (spot dot - spot arc) / (1.0 - spot arc);
spot_factor = clamp(spot_factor, 0.0, 1.0);
```

Korzystamy tutaj jeszcze z funkcji `clamp()`, aby ograniczyć współczynnik do przedziału od 0 do 1.

W wyniku tej modyfikacji, otrzymamy bardziej miękki okrąg światła jak na rys. 31.5.



Rys. 31.4. Wynik oświetlenia kierunkowego.



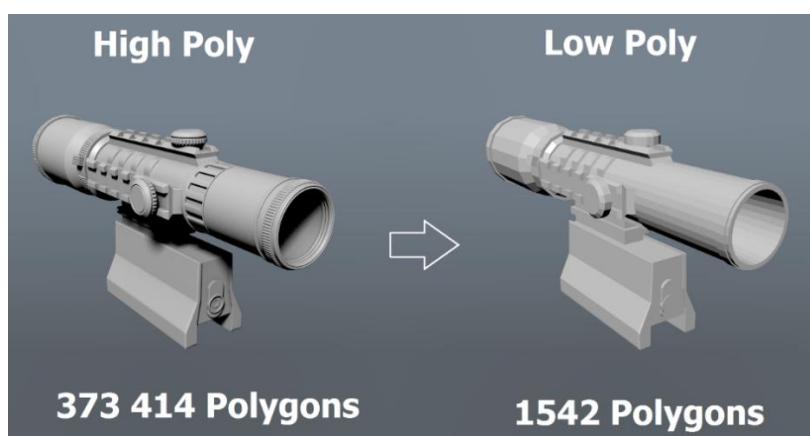
Rys. 31.5. Światło kierunkowe po modyfikacji.

Kod źródłowy z tego rozdziału jest w katalogu: 21_Reflektor

32. Mapowanie normalnych

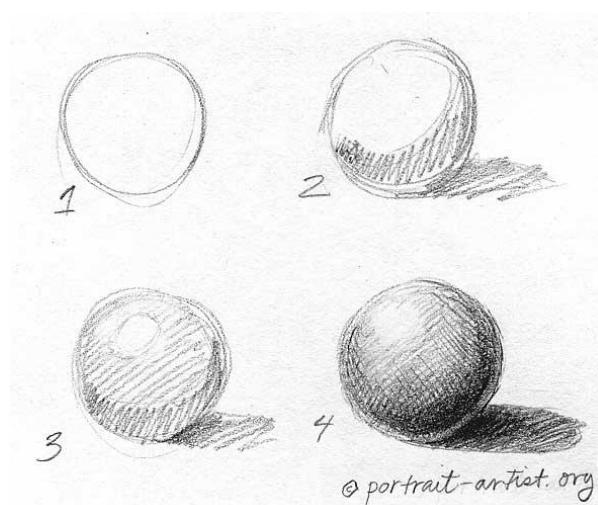
Mapowanie normalnych należy do tych technik, które nie tak dużym kosztem, pozwalają uzyskać oszałamiające efekty. Za jej działanie odpowiedzialna jest skomplikowana matematyka, ale my nie będziemy na razie wchodzić w szczegóły. Przebrniemy przez te morze zagadnień tak do pasa zanurzeni.

Renderując (rysując) dany model, na każdym jego wierzchołku wykonywane są różne obliczenia. Są to przede wszystkim transformacje oraz obliczenia oświetlenia. Im więcej wierzchołków posiada dany model graficzny, tym dłużej trwa proces renderowania. Czasem więc projektanci muszą iść na kompromis. Zmniejszają liczbę wierzchołków uproszczając geometrię modelu. Zmniejsza się wtedy szczegółowość (rys. 32.1), ale dzięki temu następuje znaczny wzrost szybkości obliczeń.



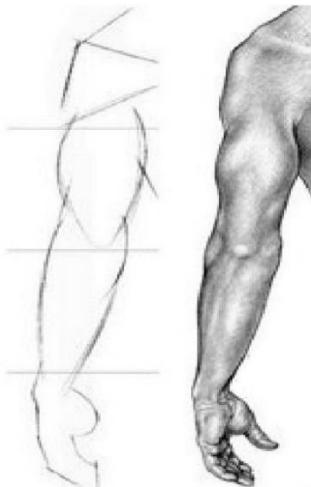
Rys. 32.1. Ograniczanie ilości wierzchołków zmniejsza szczegółowość modelu.

Idealnym rozwiązaniem byłoby zmniejszenie ilości wierzchołków, ale nie tracąc na szczegółowości. Najlepiej byłoby tą szczegółowość w jakiś sposób zasymulować. Przypomnijmy sobie teraz lekcje rysunku ze szkoły podstawowej (chyba każdy to miał), gdzie to musielismy narysować jakąś figurę, a następnie tak ją pocieniować ołówkiem, aby stała się przestrzenna (rys. 32.2).



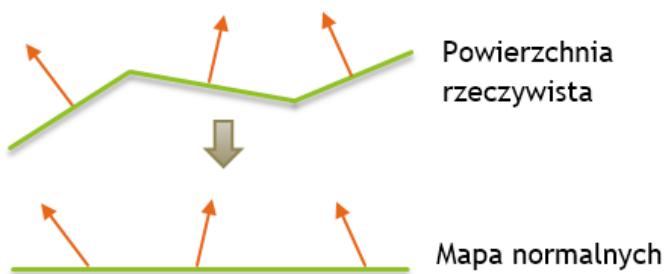
Rys. 32.2. Cieniowanie nadaje charakteru przestrzennego obiektom.

Spójrzmy dalej na przykład na rys. 32.3. Wyobraźmy sobie, że rysunek po lewej stronie jest modelem posiadającym małą liczbę wierzchołków, a zarazem małą szczegółowość. Jeśli odpowiednio go pocieniujemy, uzyskamy dużo większą szczegółowość. Szczegółowość ta nie będzie wynikała z zwiększenia ilości wierzchołków, a tylko i wyłącznie z zabiegu artystycznego, jakim jest właściwe pocieniowanie powierzchni. Jeśli obszar jasny przechodzi w obszar ciemniejszy lub odwrotnie, mamy złudzenie wypukłości bądź wklęstości powierzchni.



Rys. 32.3. Cieniowanie powierzchni wprowadza szczegółowość.

Skoro możliwe jest to na kartce papieru, to dlaczego by nie zrobić tak także na wirtualnym płótnie, czyli na ekranie. Jeśli nierówności na powierzchni nie są duże, nie musimy wcale wprowadzać dodatkowych wierzchołków, aby zamodelować te nierówności. Możemy w tym celu wykorzystać wektory normalne. Jeśli ustawiemy wektory normalne na płaskiej powierzchni pod różnymi kątami i oświetlimy tą powierzchnię, otrzymamy niejednorodne oświetlenie na tej powierzchni (rys. 32.4). Intensywność oświetlenia zależy przecież od kąta pomiędzy wektorem normalnym, a wektorem kierunku światła. Im kąt pomiędzy tymi dwoma wektorami będzie większy, tym mniejsza będzie intensywność światła. Będziemy wtedy mieli złudzenie, że skoro jest to obszar zacieniony, to musi być on położony niżej względem pozostałych obszarów. To jest właśnie technika nazywana **mapowaniem normalnych** (normal mapping).



Rys. 32.4. Schematyczne uzyskiwanie mapy normalnych.

Możemy zatem powiedzieć, że zadaniem mapowania normalnych jest symulowanie nierówności na płaskich powierzchniach. Nierówności te symulowane są właśnie za pomocą wektorów normalnych. **Nieużywane są jednak te same wektory normalne, przyczepione**

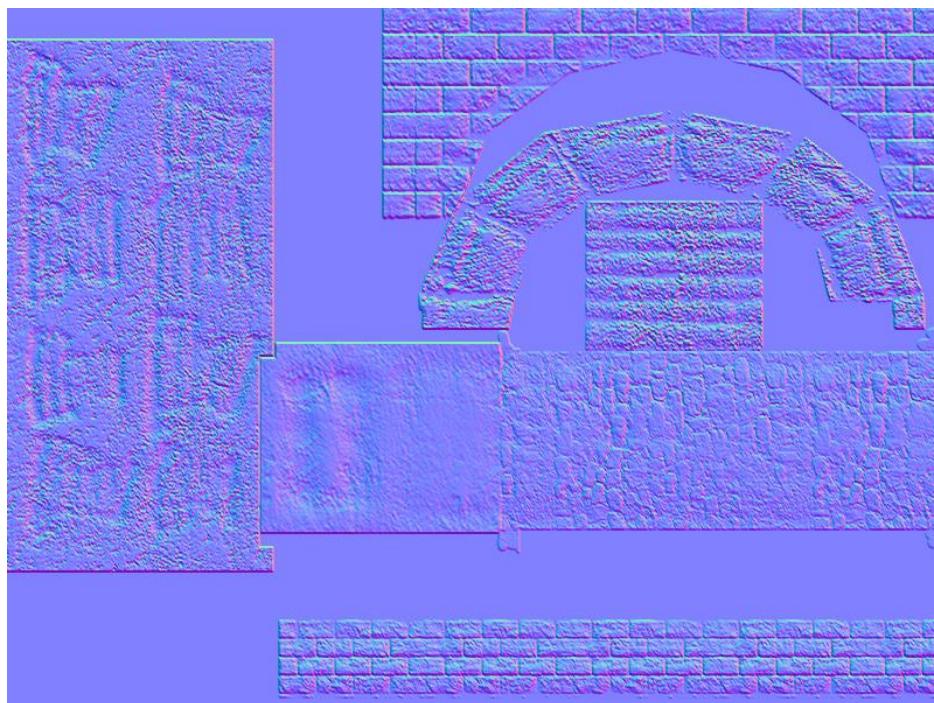
do wierzchołków, których używaliśmy do tej pory. Problemem z tymi wektorami normalnymi jest taki, że jest ich tyle samo co wierzchołków. Czyli chcąc zwiększyć szczegółowość, musielibyśmy zwiększyć liczbę wektorów normalnych, co wiąże się z zwiększeniem liczby wierzchołków i wracamy tym samym do punktu wyjścia.

W technice mapowania normalnych, wektory normalne uzyskuje się z tekstuury. **Nie obchodzi nas już wtedy wektory normalne przywiązane do wierzchołków.** Tekstura, która opisuje wektory normalne nazywana jest **mapą normalnych**. Każdy teksel takiej tekstuury opisuje kierunek i zwrot wektora normalnego przyczepionego do tego teksla. Kolor RGB opisywany jest za pomocą trzech składowych, to tak samo jest wektor normalny (XYZ). Możemy więc interpretować dane składowe koloru RGB jako kolejne współrzędne XYZ wektora normalnego.

Wykorzystywane są dwa rodzaje map normalnych:

- **mapa normalna w przestrzeni stycznej** - najlepiej nadaje się do zastosowań, gdzie obiekt podlega modyfikacjom, np. animacjom. Tego rodzaju mapy są więc używane w grach komputerowych.
- **mapa normalna w przestrzeni obiektu** - stosowana wszędzie tam, gdzie obiekt nie ulega modyfikacjom kształtu, a jedynie podlega transformacjom, np. rotacji. Jest ona szybsza niż mapa normalna w przestrzeni stycznej.

Jak na razie będziemy używać mapy normalnej w przestrzeni stycznej (**tangent space**). Jest to z reguły tekstura w odcieniach niebieskiego (rys. 32.5).

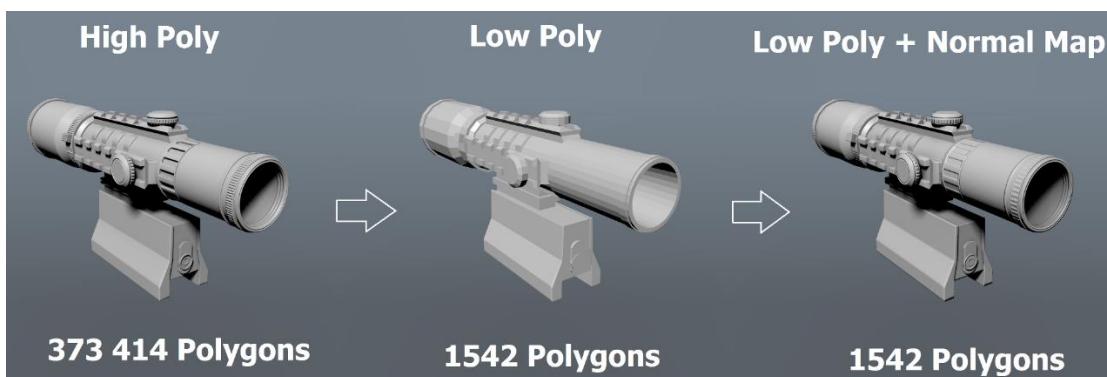


Rys. 32.5. Przykładowa mapa normalnych.

Jak wspomniałem wyżej, każdy teksel opisuje współrzędne wektora normalnego. Składowa R koloru odpowiada współrzędnej X, składowa G współrzędnej Y, oraz składowa B współrzędnej Z.

Wektor normalny prostopadły do powierzchni opisywany jest za pomocą współrzędnych $[0; 0; 1]$. Natomiast na mapie normalnych, za wektor prostopadły uważa się wektor o współrzędnych $[0.5; 0.5; 1]$, co odpowiada kolorowi fioletowemu $[0.5; 0.5; 1] = \text{RGB} [127; 127; 255]$. Wynika to stąd, że jeśli za wektor prostopadły do powierzchni przyjęłoby się wektor $[0; 0; 1]$, to nie moglibyśmy opisać odchylenia wektora w jedną stronę („na minus”). Nie możliwy jest na przykład wektor $[-0.5; 0.5; 1]$, gdyż składowa koloru na tekstuze nie może mieć przecież wartości ujemnej. Ustalając ten wektor na $[0.5; 0.5; 1]$, wektor może odchylać się w każdą stronę, ponieważ składowe koloru mogą się zmieniać od 0 do 1. Wektor normalny natomiast może być opisywany za pomocą składowych ujemnych (przedział od -1 do 1), więc trzeba przeskalać ten przedział od 0 do 1, na przedział od -1 do 1. Robimy to poprzez przemnożenie całości przez 2 i odejmując 1.

Powstaje jeszcze pytanie, jak uzyskać taką teksturę - mapę normalnych. Przy produkcji gier komputerowych, graficy pierw modelują bardzo szczegółowy model trójwymiarowy jakiegoś obiektu, a następnie na jego podstawie „wypiekają” teksturę (**texture baking**), która jest już gotową mapą normalnych. Następnie tworzą taki sam model, redukując wierzchołki oraz usuwając szczegóły. Wtedy na taki ograniczony model nakładana jest uzyskana wcześniej mapa normalnych. Ostatecznie otrzymany efekt jest prawie identyczny jak w przypadku modelu z dużą liczbą wierzchołków (rys. 32.6).

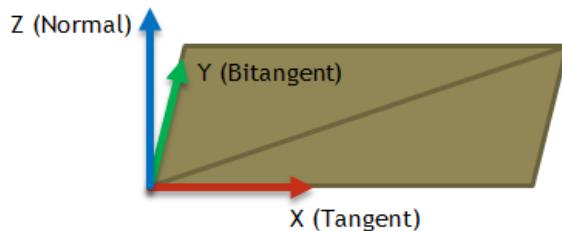


Rys. 32.6. Powstawanie modelu korzystającego z mapy normalnych.

W domowym zaciszu, gdzie nie mamy takich możliwości (i zdolności graficznych), możemy korzystać z gotowych modeli. Duża większość modeli znalezionych w internecie posiada także teksturę z mapą normalnych. Mapę normalnych możemy także wygenerować sobie sami na podstawie normalnej tekstury przekształconej do skali szarości (mapy wysokości). Służą do tego różnego rodzaju narzędzia, na przykład: <http://cpetry.github.io/NormalMap-Online/>. Można także znaleźć plugin'y do GIMPa lub Photoshopa.

Jeszcze została jedna kwestia do wyjaśnienia odnośnie map normalnych. Pobrane współrzędne wektora normalnego z tekstury, wyrażone są w układzie współrzędnym związanym z tą tekstrurą, tzw. tangent space. Tangent space jest to nic innego jak kolejny układ współrzędnych, który jest związany z konkretnym wielokątem (face), a każdy wielokąt ma taki swój indywidualny układ współrzędnych (rys. 32.7). Układ ten opisywany jest przez trzy wektory: wektor normalny prostopadły do danego wielokąta (kierunek i zwrot osi Z) (nie jest to wektor normalny odczytany z tekstrury normalnej, jest to zwykły wektor normalny wierzchołka), wektor wyznaczający kierunek rosnących współrzędnych tekstrury S (osi X) oraz wektor wyznaczający kierunek rosnących

współrzędnych tekstuury T (osi Y). Wektor osi X nazywany jest także jako **tangent**, a wektor osi Y jako **bitangent**.



Rys. 32.7. Układ współrzędnych związany z wielokątem.

Jak już wiemy, aby możliwe było wykonywanie obliczeń, wszystkie wartości musimy mieć wyrażone w jednym wspólnym układzie współrzędnych. W przypadku używania normal mapping, obliczenia powinny być wykonywane właśnie w **tangent space**. Robimy tak z prostej przyczyny, dzięki temu ograniczmy ilość wykonywanych obliczeń na macierzach. Na jednym wielokącie z tekstuury normalnej, uzyskamy mnóstwo wektorów normalnych. Będzie ich tyle co fragmentów do pokolorowania w fragment shader. Jeśli teraz mielibyśmy każdy ten wektor normalny przeliczać do innego układu współrzędnych, np. do view space, to stracimy wówczas sporo czasu obliczeniowego. Do wykonania obliczeń cieniowania powierzchni potrzebujemy z reguły wektora kierunku światła, kierunku patrzenia, położenia kamery i może jeszcze kilku jakiś dodatkowych wektorów. Widać różnicę od razu, bez sensu przeliczać setki wektorów normalnych, skoro możemy przeliczyć kilka wektorów do tangent space i nie tracić czasu.

Do wykonania transformacji z jednego układu współrzędnych do drugiego potrzebujemy oczywiście jakieś macierzy. A dokładniej, potrzebujemy macierzy, która umożliwi nam przeliczanie wektorów z **tangent space do local (model) space i odwrotnie**. Jako local (model) space mam na myśli układ współrzędnych, w którym opisywane są współrzędne wierzchołków modelu 3D (mesh). Macierz ta jest następująca:

$$\begin{bmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{bmatrix}$$

Macierz ta nazywana jest z reguły macierzą **TBN**, gdyż jak widać, składa się ona z wektora tangent, bitangent oraz normal. Powyższa forma macierzy umożliwia dokonanie transformacji z **tangent space do local space**. Macierz odwrotna do tej macierzy, umożliwia wykonanie transformacji z **local space do tangent space** i właśnie taka transformacja będzie nas interesować.

Już wiemy wszystko, co potrzebujemy, na temat mapowania normalnych. Możemy po woli zacząć pisać kod. Chciałbym tylko jeszcze raz podkreślić, technika mapowania normalnych **nie modyfikuje** położenia wierzchołków, ona to tylko **symuluje**.

Pracując w fragment shader, pracujmy nad jakimś punktem (fragmentem). Nie mamy żadnego pojęcia, gdzie ten punkt się znajduje. Tutaj z pomocą przychodzi nam wyżej wspomniany już tangent space. Tworzony jest on poprzez wektor normalny, tangent oraz

bitangent. Wektor normalny mamy (potrafimy już pobrać klasyczny wektor normalny), ale jak uzyskać tangent oraz bitangent? Możemy zrobić to ręcznie, czyli napisać sobie funkcję, która je wygeneruje. Jest to oczywiście metoda skomplikowana i jak na razie nie polecam. O wiele łatwiej jest skorzystać z gotowych dobrodziejstw biblioteki Assimp. Wystarczy, że podczas ładowania modelu dodamy parametr `aiProcess_CalcTangentSpace`.

```
const aiScene* scene = aiImportFile(file name.c_str(), aiProcess_CalcTangentSpace |  
aiProcess_Triangulate);
```

Polecam korzystać z gotowych ustawień ładowania Assimp, np. `aiProcessPreset_TargetRealtime_Fast`, które już ustawia odpowiednie parametry ładowania.

```
const aiScene* scene = aiImportFile(file_name.c_str(),  
                                    aiProcessPreset_TargetRealtime_Fast);  
  
// Assimp postprocess.h  
#define aiProcessPreset_TargetRealtime_Fast ( \  
    aiProcess_CalcTangentSpace | \  
    aiProcess_GenNormals | \  
    aiProcess_JoinIdenticalVertices | \  
    aiProcess_Triangulate | \  
    aiProcess_GenUVCoords | \  
    aiProcess_SortByPType | \  
    0 )
```

Żeby proces generacji tangent oraz bitangent przebiegł prawidłowo, model musi posiadać zdefiniowane wektory normalne.

Teraz możemy pobrać te dwa wektory z biblioteki Assimp. Robimy to tak samo jak w przypadku współrzędnych wierzchołka, czy wektora normalnego.

```
for (unsigned int v = 0; v != 3; v++)  
{  
    aiVector3D position{0, 0, 0};  
    aiVector3D normal_vector{0, 0, 0};  
    aiVector3D texture_coords{0, 0, 0};  
    aiVector3D tangent{0, 0, 0};  
    aiVector3D bitangent{0, 0, 0};  
  
    // ...  
  
    if (mesh->HasPositions())  
        position = mesh->mVertices[face->mIndices[v]];  
  
    if (mesh->HasNormals())  
        normal_vector = mesh->mNormals[face->mIndices[v]];  
  
    if (mesh->HasTextureCoords(0))  
        texture_coords = mesh->mTextureCoords[0][face->mIndices[v]];  
  
    if (mesh->HasTangentsAndBitangents())  
    {  
        tangent = mesh->mTangents[face->mIndices[v]];  
        bitangent = mesh->mBitangents[face->mIndices[v]];  
    }  
  
    // ...
```

Zostaje jeszcze niestety jeden problem. Wektor normalny nie koniecznie jest prostopadły do płaszczyzny wyznaczonej przez wektor tangent oraz bitangent. Wszystko będzie w porządku, gdy dana powierzchnia będzie płaska (jak na przykład na sześcianie). Ale weźmy pod uwagę na przykład sferę, która ma posiadać gładkie oświetlenie (smooth lighting). W

takiej sferze, wektor normalny wierzchołka jest uśredniany na podstawie otaczających wielokątów. Ostatecznie więc, wektor normalny nie będzie prostopadły do osi wyznaczających współrzędne tekstury. Musimy zatem dodatkowo skorygować te wektory.

```
glm::vec3 n(normal_vector.x, normal_vector.y, normal_vector.z);
glm::vec3 t(tangent.x, tangent.y, tangent.z);
glm::vec3 b(bitangent.x, bitangent.y, bitangent.z);

glm::vec3 tangent_corrected = glm::normalize(t - n * glm::dot(n, t));

float det = glm::dot(glm::cross(n, t), b);
if (det < 0.0f)
    det = -1.0f;
else
    det = 1.0f;

glm::vec3 bitangent_corrected = glm::cross(n, tangent_corrected) * det;
```

W pierwszej kolejności obliczamy nowy wektor tangent, który będzie prostopadły do wektora normalnego. Do tego celu wykorzystywany jest algorytm [Gram-Schmidta](#). Spokojnie, jego zrozumienie nie jest wymagane. Wektor bitangent obliczamy jako iloczyn wektorowy wektora normalnego oraz wektora tangent po korekcji. Musimy być tylko pewni, że obliczony wektor bitangent posiada taki sam znak, jak wektor bitangnet wyznaczony przez Assimp. W tym celu sprawdzamy iloczyn skalarny (jego znak) i w zależności od wartości, przemnażamy nowy wektor bitangent przez jeden lub minus jeden.

Następnym krokiem jest wrzucenie nowych danych do obiektów VBO, a następnie do VAO.

```
std::vector<GLfloat> tangent_container;
std::vector<GLfloat> bitangent_container;

tangent_container.push_back(tangent_corrected.x);
tangent_container.push_back(tangent_corrected.y);
tangent_container.push_back(tangent_corrected.z);

bitangent_container.push_back(bitangent_corrected.x);
bitangent_container.push_back(bitangent_corrected.y);
bitangent_container.push_back(bitangent_corrected.z);

// ...

GLuint tangent_vbo = 0;
glGenBuffers(1, &tangent_vbo);
glBindBuffer(GL_ARRAY_BUFFER, tangent_vbo);
glBufferData(GL_ARRAY_BUFFER, tangent_container.size() * sizeof(GLfloat),
             tangent_container.data(), GL_STATIC_DRAW);

GLuint bitangent_vbo = 0;
glGenBuffers(1, &bitangent_vbo);
glBindBuffer(GL_ARRAY_BUFFER, bitangent_vbo);
glBufferData(GL_ARRAY_BUFFER, bitangent_container.size() * sizeof(GLfloat),
             bitangent_container.data(), GL_STATIC_DRAW);

// ...

glBindBuffer(GL_ARRAY_BUFFER, tangent_vbo);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, bitangent_vbo);
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, 0, 0);

 glEnableVertexAttribArray(3);
 glEnableVertexAttribArray(4);
```

Pozostaje jeszcze kwestia tekstury z mapą normalnych. Możemy ją załadować w dowolny sposób. Możemy zrobić to ręcznie, czyli po prostu samemu podać ścieżkę do tekstury.

Możemy wykorzystać także do tego celu bibliotekę Assimp, aby uzyskać ścieżkę do tekstury z mapą normalnych (oczywiście jeśli dany model taką posiada).

```
const aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex];
if (material->GetTexture(aiTextureType_HEIGHT, 0, &texture_path) == AI_SUCCESS)
{
    // Zaladuj teksture
}
```

Zauważylem, że Assimp błędnie w plikach OBJ interpretuje teksturę z mapą normalnych, jako mapę wysokości, dlatego tutaj użyłem `aiTextureType_HEIGHT`. W przypadku, gdyby tekstura nie była odczytywana poprawnie, należy zmienić ten parametr na `aiTextureType_NORMALS`.

Dalej postępujemy tak samo jak w przypadku multiteksturowania. Bindujemy tą teksturę do jakiegoś slotu, np. 1:

```
// Diffuse texture
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, diffuse_texture);
// Normalmap texture
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, normalmap_texture);
```

Nasz program jest już gotowy, teraz musimy zmodyfikować shadery. Zaczniemy od vertex shader.

Zadaniem vertex shader jest przeliczenie wszystkich potrzebnych wektorów do tangent space. Tangent space jest tym „najniższym” układem współrzędnym. Całe drzewo wykorzystywanych układów współrzędnych można przedstawić tak jak na rys. 32.8.



Rys. 32.8. Zależności między układami współrzędnych.

Pozycję kamery w układzie świata możemy uzyskać z macierzy widoku. Położenie kamery w układzie kamery jest równe $(0.0, 0.0, 0.0)$, a więc jeśli ten punkt przetransformujemy poprzez odwrotną macierz widoku, uzyskamy położenie kamery w world space. Następnie transformujemy je na model space. Transformujemy pozycję, a zatem czwarta składowa wektora wynosi 1.0.

Pozycja źródła światła wyrażona jest w world space, a więc musimy przetransformować ją na model space. Dzięki temu będziemy mogli obliczyć kierunek światła poprzez różnicę pozycji wierzchołka oraz pozycji źródła światła.

Podobnie postępujemy w przypadku kierunku patrzenia, który będzie potrzebny do obliczenia światła odbitego (specular).

Dalej możemy wyznaczyć także macierz TBN. Odwrotność tej macierzy używamy, aby przetransformować kierunek patrzenia oraz kierunek światła na tangent space. Te dwie ostatnie wartości przekazujemy dalej w potok do fragment shader.

```
#version 330
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;
layout(location = 2) in vec2 texture_coord;
layout(location = 3) in vec3 tangent;
layout(location = 4) in vec3 bitangent;

uniform mat4 view_matrix;
uniform mat4 perspective_matrix;
uniform mat4 model_matrix;

out vec2 texture_coordinates;
out vec3 view_direction_tangent;
out vec3 light_direction_tangent;

vec3 light pos world = vec3(-4.0, 5.0, -10.0);

void main()
{
    gl_Position = perspective_matrix * view_matrix * model_matrix * vec4(vertex_position,
1.0);
    texture coordinates = texture coord;

    vec3 camera_pos_world = (inverse(view_matrix) * vec4(0.0, 0.0, 0.0, 1.0)).xyz;
    vec3 camera pos local = vec3(inverse(model matrix) * vec4(camera pos world, 1.0));

    vec3 light pos local = (inverse(model matrix) * vec4(light pos world, 1.0)).xyz;
    vec3 light dir local = normalize(vertex position - light pos local);

    vec3 view_dir_local = normalize(camera_pos_local - vertex_position);
    mat3 tbn = mat3(tangent, bitangent, vertex normal);

    view_direction_tangent = inverse(tbn) * view_dir_local;
    light_direction_tangent = inverse(tbn) * light_dir_local;
}
```

Fragment shader wygląda następująco:

```
#version 330
in vec2 texture_coordinates;
in vec3 view_direction_tangent;
in vec3 light direction tangent;

uniform mat4 view_matrix;
uniform sampler2D basic_texture;
uniform sampler2D normal texture;

out vec4 frag colour;

vec3 ambient_color = vec3(0.3, 0.3, 0.3);
vec3 diffuse_color = vec3(0.8, 0.8, 0.8);
vec3 specular color = vec3(1.0, 1.0, 1.0);

vec3 object ambient factor = vec3(1.0, 1.0, 1.0);
vec3 object diffuse factor = vec3(1.0, 1.0, 1.0);
vec3 object specular_factor = vec3(1.0, 1.0, 1.0);

void main()
{
    vec3 normal tangent = texture(normal texture, texture coordinates).rgb;
    normal tangent = normalize(normal tangent * 2.0 - 1.0);

    // Ambient
    vec3 ambient intense = ambient color * object ambient factor;

    // Diffuse
    vec3 direction_to_light_tangent = normalize(-light_direction_tangent);
```

```
float dot_product = dot(direction_to_light_tangent, normal_tangent);
dot product = max(dot product, 0.0);
vec3 diffuse intense = diffuse color * object diffuse factor * dot product;

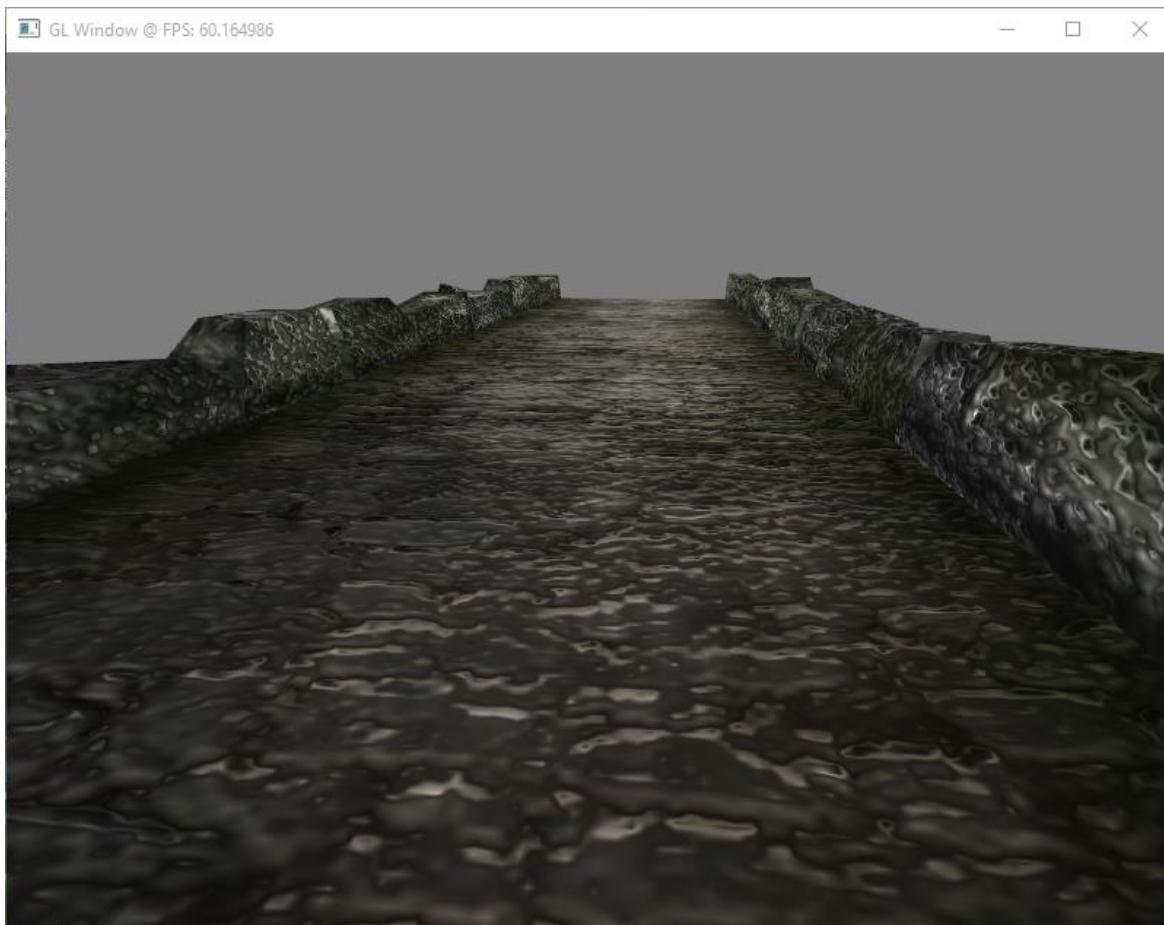
// Specular
vec3 reflection_tangent = reflect(normalize(light_direction_tangent), normal_tangent);
float dot specular = dot(reflection tangent, normalize(-view direction tangent));
dot specular = max(dot specular, 0.0);
float specular factor = pow(dot specular, 10.0);
vec3 specular intense = specular color * object specular factor * specular factor;

vec4 texel = texture(basic_texture, texture_coordinates);
frag colour = vec4(diffuse intense + specular intense + ambient intense, 1.0) * texel;
}
```

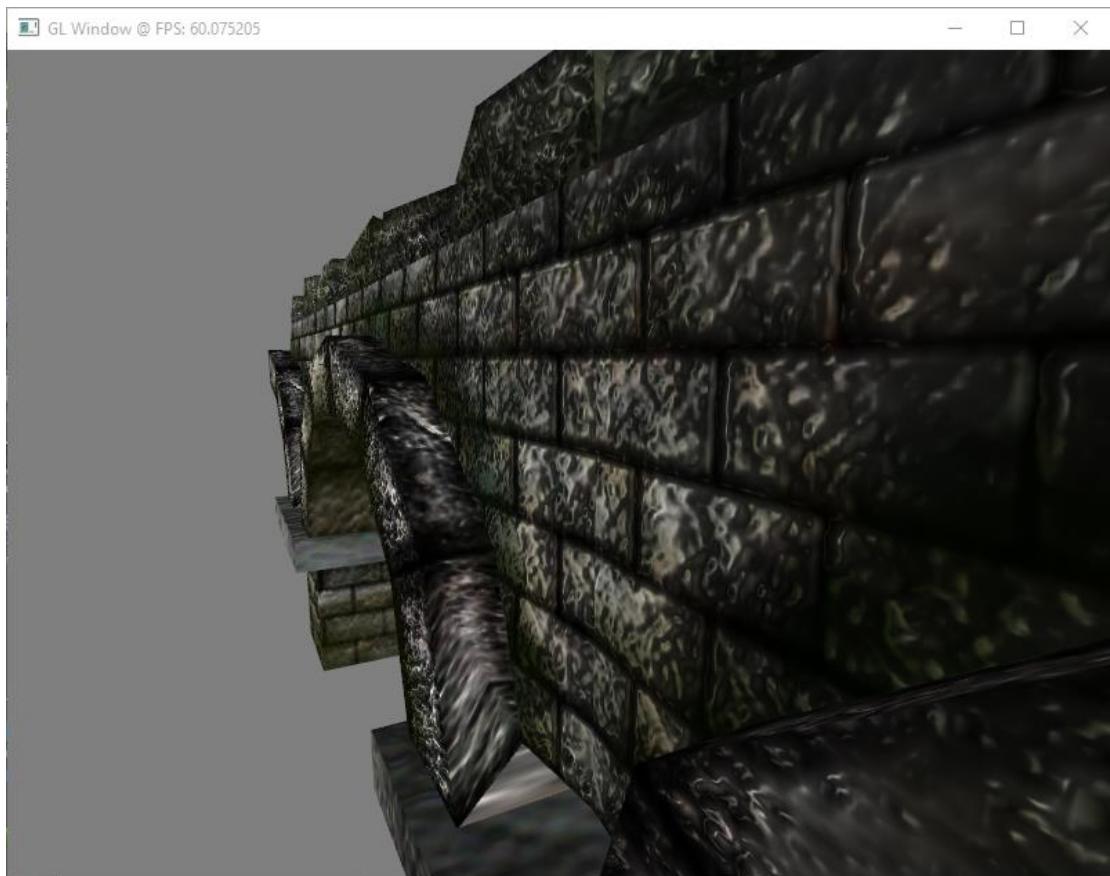
W pierwszej kolejności pobieramy kolor danego tekstuła z mapą normalnych, czyli jest to zmienienna `normal_texture`. Pobieramy składową RGB, gdyż interesują nas tylko trzy wartości (XYZ). Następnie skalujemy uzyskany wektor na przedział od -1 do 1 i normalizujemy go. W ten sposób uzyskaliśmy nowy wektor normalny, który będzie wykorzystywany do obliczania oświetlenia.

Dalej w fragment shader już nic ciekawego się nie dzieje. Dokonujemy obliczeń oświetlenia Phonga, tak samo jak robiliśmy to przedtem. Z tą tylko różnicą, że wykorzystujemy tutaj nasz nowy wektor normalny uzyskany z tekstuły.

Mogemy już uruchomić nasz program i zobaczyć efekty (rys. 32.9 i 32.10). Wyraźnie widać, że powierzchnia wydaje się nierówna, co zwiększa efekt realizmu.



Rys. 32.9. Wynik działania programu.



Rys. 32.10. Wynik działania programu.

Kod źródłowy z tego rozdziału jest w katalogu: **22_Normal_mapping**

Przykładowy model dostępny jest na stronie „[Pobierz](#)”.

33. Przechwytywanie obrazu

Czasem chcemy pochwalić się naszą przepięknie zaprojektowaną grafiką i robimy zrzut ekranu za pomocą klawisza PrtSc albo podobnych narzędzi. Nie zawsze jest to dobre rozwiązanie. Jeśli chcemy pokazać same wnętrze okna, to musimy ręcznie usuwać w programie graficznym obramowania i pasek tytułu okienka. Mogą także zdarzyć się takie sytuacje, że po zrobieniu zrzutu ekranu, pojawią się na nim poziome linie wynikające z przeplatania się różnych klatek obrazu, np. podczas animacji. Możemy więc napisać swoją własną funkcję przechwytywania obrazu, która zapisze gotowy plik graficzny do pliku.

Rysując coś na ekranie, tak naprawdę rysujemy na tzw. **framebuffer**. Po skończonym procesie rysowania, gotowa ramka jest dopiero wyświetiana na ekranie. Możemy pobrać zawartość framebuffer, a następnie wykorzystując jakąś bibliotekę obsługi plików graficznych, np. FreeImage, zapisać ją do pliku.

Stwórzmy sobie zatem następującą funkcję:

```
void takeScreenshot(std::string file_name)
{
    unsigned char *screen_buffer = new unsigned char[window_width *
    window_height * 3];

    glReadPixels(0, 0, window_width, window_height, GL_BGR, GL_UNSIGNED_BYTE,
    screen_buffer);

    FIBITMAP *image = FreeImage_ConvertFromRawBits(screen_buffer, window_width,
    window_height, 3 * window_width,
    24, 0xFF0000, 0x00FF00,
    0x0000FF, false);

    FreeImage_Save(FIF_PNG, image, file_name.c_str());

    std::cout << "Screenshot \" " << file_name << "\" saved." << std::endl;
}
```

Na początku tworzymy bufor oraz alokujemy odpowiednią ilość pamięci. Chcemy zapisywać cały obraz w formacie RGB, a zatem musimy przemnożyć przez siebie trzy wartości: szerokość obrazu, wysokość obrazu oraz ilość składowych koloru (zapisujemy w RGB, dlatego tutaj jest wartość 3).

Następnie korzystamy z funkcji `glReadPixels()`, która pobiera piksele z framebuffer. W pierwszych dwóch parametrach podajemy współrzędne XY piksela, od którego ma zacząć się pobieranie wartości. Kolejnymi dwoma parametrami określamy rozmiar pobieranego bloku pikseli. Pobieramy cały obraz, więc wpisujemy tutaj szerokość oraz wysokość obrazu. Piąty parametr określa format obrazu. Korzystamy z formatu `GL_BGR`, gdyż składowe koloru w OpenGL przechowywane są w odwrotnej kolejności. Pozostało nam jeszcze sprecyzować typ danych, w jakim zapisane są piksele oraz przekazać nasz bufor na piksele w ostatnim parametrze.

Aby zapisać pobrane piksele do pliku, korzystamy z biblioteki FreeImage. Posiada ona zgrabną funkcję `FreeImage_ConvertFromRawBits()`, która robi to w sposób niezwykle łatwy. Jako pierwszy parametr przekazujemy jej bufor, z którego ma odczytywać. Następnie szerokość oraz wysokość obrazu. Dane w buforze zapisane są w sposób

następujący RGBRGBRGB ... RGB ..., a więc w czwartym parametrze podajemy, że „nowy wiersz” pikseli zaczyna się od wartości szerokości obrazu przemnożonej przez 3 (bo 3 składowe koloru RGB). Piąty parametr określa ilość bitów na jeden piksel (kolor zapisujemy na 8 bitach, gdyż maksymalna wartość składowej koloru to 255, a więc $8 \text{ razy } 3 = 24$). Kolejne trzy parametry określają maski koloru R, G oraz B. Ostatni parametr określa, czy kierunek osi Y obrazu ma być od góry do dołu, czy odwrotnie.

Na samym końcu wywołujemy funkcję `FreeImage_Save()`, która zapisuje uzyskany wyżej obraz do pliku. W pierwszym parametrze podajemy żądanego formatu pliku. Tutaj jest to `FIF_PNG`. Można także wpisać tutaj np. `FIF_BMP`, `FIF_JPEG`, itp.

Teraz zostało nam już tylko w jakimś miejscu w naszym programie, np. w funkcji obsługi klawiszy, wywołać tą funkcję.

```
if (key == GLFW_KEY_T && action == GLFW_RELEASE)
    takeScreenshot("test.png");
```

I gotowe!

Kod źródłowy z tego rozdziału jest w katalogu: 23_Screenshot

34. Skybox

Nadszedł czas porzucić nudne kolorowanie tła na jednolity kolor. W tym rozdziale pokażę jak zrobić to o wiele ciekawiej.

Grając w gry komputerowe, na pewno kiedyś słyszałeś o czymś, co nazywa się **skybox**. Za jego pomocą najczęściej tworzy się tło (otoczenie) sceny, np. chmury, gwiazdy, horyzont, itp. (rys. 34.1). Innymi słowy, potęguje on wrażenie efektu 3D w tle, za sceną.



Rys. 34.1. Chmury w oddali to efekt uzyskany za pomocą skybox.

Skybox najprościej można zdefiniować jako sześcian, w którym znajduje się kamera. Na każdą ścianę tego sześcianu nałożona jest jakaś tekstura, np. chmury. Wraz z poruszaniem się kamery, porusza się także ten sześcian, utrzymując kamerę zawsze w samym centrum. Ponadto, sześcian ten, nigdy nie ulega obrotom. Jeśli kamera się obraca, to sześcian pozostaje nieruchomy.

Wyżej pisałem o sześcianie, ale skybox nie musi koniecznie być sześcianem. Czasem lepszy efekt daje sfera. Wtedy mówimy o **skydome**. Na razie jednak pozostaniemy tylko przy sześcianie, gdyż łatwiej jest go teksturować.

Zacznijmy od zdefiniowania sześcianu. Możemy albo zrobić to w sposób klasyczny, poprzez ręczne zdefiniowanie wierzchołków, albo wczytać go z pliku. Ja zrobiłem to w sposób ręczny, aby uprościć sprawę.

```
GLfloat skybox[] = {
    -1.0f,  1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
```

```

-1.0f, -1.0f, 1.0f,
-1.0f, -1.0f, -1.0f,
-1.0f, 1.0f, -1.0f,
-1.0f, 1.0f, -1.0f,
-1.0f, 1.0f, 1.0f,
-1.0f, -1.0f, 1.0f,

1.0f, -1.0f, -1.0f,
1.0f, -1.0f, 1.0f,
1.0f, 1.0f, 1.0f,
1.0f, 1.0f, 1.0f,
1.0f, 1.0f, -1.0f,
1.0f, -1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,
-1.0f, 1.0f, 1.0f,
1.0f, 1.0f, 1.0f,
1.0f, 1.0f, 1.0f,
1.0f, -1.0f, 1.0f,
-1.0f, -1.0f, 1.0f,

-1.0f, 1.0f, -1.0f,
1.0f, 1.0f, -1.0f,
1.0f, 1.0f, 1.0f,
1.0f, 1.0f, 1.0f,
-1.0f, 1.0f, 1.0f,
-1.0f, 1.0f, -1.0f,

-1.0f, -1.0f, -1.0f,
-1.0f, -1.0f, 1.0f,
1.0f, -1.0f, -1.0f,
1.0f, -1.0f, -1.0f,
-1.0f, -1.0f, 1.0f,
1.0f, -1.0f, 1.0f
};

}

```

Musimy pamiętać o tym, że znajdujemy się wewnątrz sześcianu, a zatem musimy uwzględnić prawidłowy **winding order**. Domyślnie ustawione jest tak, że przód ściany jest tam, gdzie wierzchołki podawane są przeciwnie do ruchu wskazówek zegara. Jakby co, możemy to zawsze zmienić za pomocą funkcji `glFrontFace()`.

Następnie tworzymy obiekt VAO na skybox:

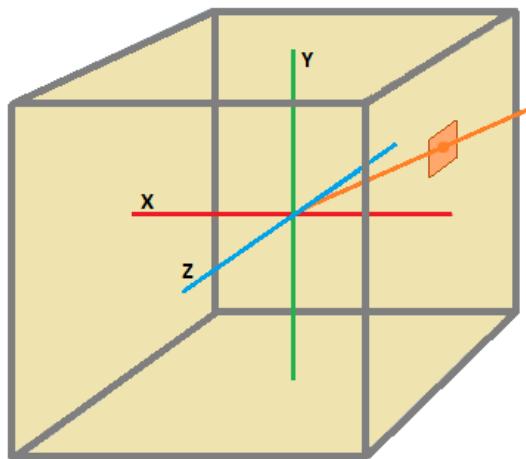
```

GLuint skybox_vbo = 0;
 glGenBuffers(1, &skybox_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, skybox_vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(skybox), &skybox, GL_STATIC_DRAW);

GLuint skybox_vao = 0;
 glGenVertexArrays(1, &skybox_vao);
 glBindVertexArray(skybox_vao);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);

```

Jak może zauważyłeś, nie zdefiniowaliśmy współrzędnych tekstury. Podczas teksturowania płaskiej powierzchni zdefiniowany był w narożniku tekstury układ współrzędnych S-T, który pozwalał odnosić się do konkretnego teksla. W skybox jest nieco inaczej. Układ współrzędnych zaczepiony jest wewnątrz, w samym centrum sześcianu. Kolor danego teksla tekstury pobierany jest poprzez wektor kierunkowy wychodzący z tego układu współrzędnych (rys. 34.2). To skąd wziąć w takim razie ten wektor kierunkowy? Zauważ, że ten układ współrzędnych, to jest ten sam układ, w którym definiujemy położenie wierzchołków sześcianu. Definiując tablicę wierzchołków, zdefiniowaliśmy także od razu tablicę wektorów umożliwiających pobranie odpowiedniego teksla tekstury (czyli tak jakby zdefiniowaliśmy przy okazji współrzędne tekstury).



Rys. 34.2. Pobieranie próbki tekstury w skybox.

I tak, na przykład, wektor [0, 1, 0] pobierze kolor z samego środka górnej ściany sześcianu, natomiast wektor [1, 1, 1] pobierze kolor z prawego górnego narożnika sześcianu.

Jeszcze co do współrzędnych wierzchołków kostki. My w tablicy zdefiniowaliśmy wierzchołki używając wartości -1, 0 oraz +1. Równie dobrze możemy użyć wartości -10, 0, +10, itp. Nie ma to żadnego znaczenia. Rozmiar skybox i tak będzie zawsze taki sam. Wynika to z odpowiedniego sposobu renderowania, o którym będzie napisane nieco niżej. Jeśli zdefiniujemy kostkę o współrzędnych, np. -10, 0, +10, to i tak nie ma potrzeby normalizacji wektorów próbkowania tekstury. Wszystko będzie działać tak samo, jak w przypadku wartości -1, 0, +1, gdyż tutaj nie jest ważna długość tego wektora, chodzi tylko o pobranie koloru z danego kierunku.

Są dwie metody zapisu tekstuury, którą można użyć do skybox. Albo tworzymy jedną dużą teksturę, która umożliwia nałożenie tekstuury na wszystkie ściany sześcianu, albo tworzymy 6 oddzielnych tekstuur i ładujemy je kolejno. My skorzystamy z tej drugiej metody. W tym celu możemy napisać sobie funkcję, która przyjmie jako parametr sześć ścieżek do plików z tekstem.

```
void loadSkybox(std::string front, std::string back, std::string left,
                std::string right, std::string up, std::string down,
                GLuint &texture_handle)
{
    glGenTextures(1, &texture_handle);
    glBindTexture(GL_TEXTURE_CUBE_MAP, texture_handle);

    std::string textures[] = {right, left, down, up, back, front};

    for (int i = 0; i < 6; i++)
    {
        Texture texture;
        loadTexture(textures[i], texture);

        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, texture.width,
                    texture.height, 0, GL_BGR, GL_UNSIGNED_BYTE, texture.bits);
    }

    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}
```

Na samym początku generujemy nową teksturę za pomocą funkcji `glGenTextures()`, a następnie aktywujemy ją (bindujemy) za pomocą funkcji `glBindTexture()`. W przypadku zwykłej tekstury, jako parametr funkcji `glBindTexture()` używaliśmy wartości `GL_TEXTURE_2D`. Tutaj natomiast używamy wartości `GL_TEXTURE_CUBE_MAP`. Parametr ten ustawia, że w jednym obiekcie tekstury `texture_handle`, możemy tak jakby przechować sześć tekstur tworzących skybox.

Kolejnym krokiem jest uruchomienie pętli, która posiada 6 iteracji. W każdej iteracji ładowana jest następna tekstura z tablicy, a następnie jest „zapisywana” w pamięci za pomocą funkcji `glTexImage2D()`. Używamy jej tak samo jak w przypadku zwykłej tekstury, z jedną różnicą. Pierwszym parametrem w przypadku tekstur 2D było `GL_TEXTURE_2D`, a teraz mamy coś innego, jakieś `GL_TEXTURE_CUBE_MAP_POSITIVE_X + i`. Otóż, pierwszy parametr tej funkcji określa cel, czyli do jakiego elementu chcemy przypisać tą teksturę. Dla skybox mamy następujące opcje:

```
// Plik glew.h
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X 0x8515
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_X 0x8516
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Y 0x8517
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Y 0x8518
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Z 0x8519
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Z 0x851A
```

Jak spojrzymy się ponownie na rys. 34.2, to od razu zobaczymy o co chodzi. Wartości te określają, do której ze ścian sześcianu ma zostać przypisana dana tekstura. Konkretna ściana sześcianu jest określana za pomocą osi układu współrzędnych: albo dodatni kierunek osi, albo ujemny.

Zamiast tworzyć sześć wywołań funkcji `glTexImage2D()` dla każdego parametru: `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, itd., możemy skorzystać z wartości tych parametrów. Parameter `GL_TEXTURE_CUBE_MAP_POSITIVE_X` ma wartość `0x8515`. Jeśli dodamy jeden, to uzyskamy wtedy wartość parametru `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, a następnie po kolejnym dodaniu jedynki otrzymamy `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, itd.

Na samym końcu, już za pętlą, musimy jeszcze ustawić filtrowanie tekstur oraz musimy zmusić teksturę, żeby przylegała do samych krawędzi. Bez tego widoczne będą brzydkie „szwy” na krawędziach sześcianu. Tekstura w skybox jest teksturą trójwymiarową, dlatego musimy ustawić parametr `GL_CLAMP_TO_EDGE`, dla trzech współrzędnych S, T oraz R. Dla tekstur skybox nie generujemy także mipmap, z tego względu, że zawsze będą znajdowały się w takiej samej odległości od kamery.

Gdy już nasza funkcja jest gotowa, możemy ją wywołać.

```
GLuint skybox_texture = 0;
loadSkybox("hills_ft.tga", "hills_bk.tga", "hills_lf.tga", "hills_rt.tga",
           "hills_up.tga", "hills_dn.tga", skybox_texture);
```

Renderowanie skybox wymaga oddzielnych shaderów. Ale nie bój się, będą to najprostsze shadery jakie widziałeś w życiu. Zaczniemy od vertex shader.

```
#version 330
layout(location = 0) in vec3 position;

uniform mat4 view_matrix;
uniform mat4 perspective_matrix;

out vec3 texture_coordinates;

void main()
{
    texture coordinates = vec3(position.x, -position.yz);
    gl_Position = perspective_matrix * view_matrix * vec4(position, 1.0);
}
```

Vertex shader dla skybox przyjmuje na wejście tylko jedną wartość, którą jest pozycja wierzchołka. Standardowo, przyjmuje także macierz widoku oraz perspektywy. Zadaniem tego shadera jest zwykłe wyświetlenie wierzchołka na ekranie oraz przekazanie do fragment shader współrzędnych tekstury. Według tego, o czym była mowa wyżej, w skybox nie mamy współrzędnych tekstury, mamy tylko wektory kierunku. Są one równe oczywiście współrzędnym danego wierzchołka, musimy tylko odwrócić znak współrzędnej Y oraz Z. Jeśli tego nie zrobimy, tekstury na bokach sześcianu skybox, będą do góry nogami. No chyba, że przedtem obrócimy je sami w jakimś programie graficznym.

Fragment shader jest również prosty, co vertex shader.

```
#version 330
in vec3 texture_coordinates;

uniform samplerCube cube_texture;

out vec4 frag_colour;

void main()
{
    frag_colour = texture(cube_texture, texture_coordinates);
```

Pojawił się w nim nowy typ zmiennej `samplerCube`. Jest to ten specjalny typ tekstury dla sześcianu, w którym upakowanych jest sześć tekstur. Potrafi on oczywiście działać na współrzędnych trójwymiarowych tekstury, czyli RST.

Poza tym, fragment shader pobiera tylko próbkę tekstury oraz koloruje nią dany fragment. Naprawdę nie ma tu nic skomplikowanego.

Skoro mamy już nasze nowe shadery, musimy je załadować oraz odczytać lokalizację zmiennych `uniform`.

```
Gluint skybox_shader = 0;
if (createShaderProgram(skybox_shader, "vertex_shader_skybox.glsl",
                       "fragment shader skybox.glsl"))
    return -1;

Gluint view_uniform_sky = findUniform(skybox_shader, "view_matrix");
Gluint perspective_uniform_sky = findUniform(skybox_shader, "perspective_matrix");
```

Wszystko jest już gotowe, możemy przystąpić do renderowania naszego skybox.

```
activateShaderProgram(skybox_shader);
setUniform(view uniform sky, view matrix);
setUniform(perspective uniform sky, perspective);
glDisable(GL_DEPTH_TEST);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, skybox_texture);
glBindVertexArray(skybox_vao);
glDrawArrays(GL_TRIANGLES, 0, 36);
 glEnable(GL_DEPTH_TEST);
```

W pierwszej kolejności musimy aktywować program shadera dla skybox oraz przesyłać zaktualizowane macierze widoku oraz perspektywy.

Skybox ma być renderowany jako tło, a nie jako klasyczny obiekt trójwymiarowy. W tym celu należy wyłączyć testowanie głębi za pomocą funkcji `glDisable(GL_DEPTH_TEST)`. Od tej pory wyłączony jest bufor głębokości, więc renderując, nieuwzględniana jest w żaden sposób głębokość kostki skybox. Jest po prostu „spłaszczana” do tła.

Dalsze operacje nie są już niczym nowym. Aktywujemy slot tekstury, bindujemy teksturowę skybox, bindujemy VAO oraz rysujemy za pomocą funkcji `glDrawArrays()`. Na samym końcu włączamy z powrotem testowanie głębi, aby dalej renderowane obiekty były wyświetlane prawidłowo, czyli z zachowaniem głębokości na scenie.

Podkreślę jeszcze, że skybox ma być tłem. Rysujemy go z wyłączeniem testowania głębi, a zatem **musimy go rysować na samym początku**. Dopiero później rysujemy pozostałe obiekty już z włączonym testowaniem głębi.

Jeśli uruchomimy teraz nasz program, zauważymy, że jest coś nie tak (rys. 34.3). Możemy wyjść poza obszar kostki skybox.



Rys. 34.3. Nieprawidłowo wyświetlony skybox.

Zapomnieliśmy o jednej rzeczy, o usunięciu translacji kamery z macierzy widoku. Kamera musi być umieszczona zawsze w centrum skybox. Położenie kamery w układzie współrzędnych kamery jest równe (0, 0, 0) (w view space), a zatem skybox musi posiadać zawsze takie współrzędne. W tym celu należy usunąć translację kamery z macierzy widoku (rys. 34.4).

$$\begin{bmatrix} R_x & R_y & R_z & -P_x \\ U_x & U_y & U_z & -P_y \\ -F_x & -F_y & -F_z & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rys. 34.4. W macierzy widoku zerujemy położenie kamery - trzy składowe wewnętrz czerwonej pętli.

Można to zrobić na kilka sposobów. Najbardziej obrazowo jest moim zdaniem w ten sposób:

```
activateShaderProgram(skybox shader);
glm::mat4 view_static = view_matrix;
glm::vec3 pos(0.0, 0.0, 0.0);
view_static = glm::lookAt(pos, pos + camera_direction, camera_up);
setUniform(view_uniform_sky, view_static);
setUniform(perspective_uniform_sky, perspective);
```

Od tej pory już wszystko powinno działać prawidłowo (rys. 34.5).



Rys. 34.5. Efekt końcowy.

Czasem jest tak, że trzeba trochę pomanewrować teksturami, aby były dobrze dopasowane. Jeśli są obrócone, możemy naprawić to programowo, albo po prostu obrócić je w programie graficznym. Częstym problemem jest także to, że kostka skybox jest do góry nogami. Należy wtedy upewnić się, czy współrzędne przekazywane są z właściwym znakiem.

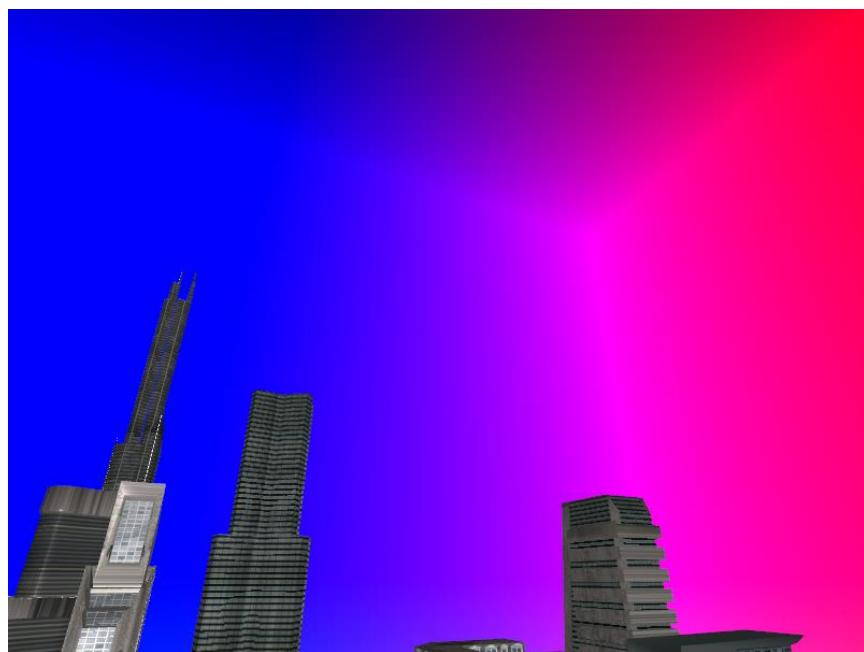
Dobrym źródłem skybox'ów do ściągnięcia jest strona:
<http://www.custommapmakers.org/skyboxes.php>.

Tips & Tricks:

- Jeśli mamy jakieś problemy z źle wyświetlonymi teksturami w skybox, możemy to zdebugować za pomocą fragment shader. Wystarczy, że fragmenty będziemy kolorować za pomocą współrzędnych tekstury:

```
frag_colour = vec4(texture_coordinates, 1.0);
```

W uzyskanych kolorach powinna być zachowana ciągłość (rys. 34.6).



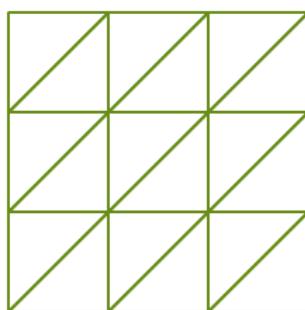
Rys. 34.6. Debugowanie skybox za pomocą fragment shader.

Kod źródłowy z tego rozdziału jest w katalogu: 24_Skybox

35. Prosty generator terenu

Teren jest powszechnie wykorzystywany między innymi w grach komputerowych. Jest to po prostu podłoże, po którym porusza się gracz. Generacja terenu jest ogólnie bardzo skomplikowanym zagadnieniem. Istnieje wiele narzędzi oraz algorytmów, które to umożliwiają, ale my nie będziemy z nich korzystać. Stworzymy własny prosty generator, który w zupełności wystarczy do zastosowań hobbystycznych.

Tak samo jak każdy inny obiekt trójwymiarowy, teren jest także zbudowany z siatki trójkątów. W najprostszym przypadku, gdy teren nie posiada żadnych nierówności, siatka takiego terenu wygląda tak jak na rys. 35.1.



Rys. 35.1. Najprostsza siatka terenu.

Można zauważyć, że cechą charakterystyczną terenu jest to, że tworzy go bardzo regularny układ wierzchołków. I z tego właśnie względu bardzo łatwo jest go generować programowo. Przeważnie razem z współrzędnymi wierzchołków generowane są także wektory normalne oraz współrzędne tekstury.

Każdemu wierzchołkowi można także przypisać jakąś wysokość. Podczas procesu generowania wystarczy, że przesuniemy go do góry bądź do dołu wzdłuż osi Y. W ten sposób uzyskamy nierówności terenu, takie jak pagórki, czy doliny.

Oczywiście teren można także wygenerować w jakimś zewnętrznym programie, np. Blender, i importować go do naszego programu tak samo jak importujemy inne modele graficzne. Ale w tym rozdziale skupiamy się na generacji, import modeli już nie jest niczym ciekawym.

Do zagadnienia podejdziemy w dwóch krokach. Na początku stworzymy generator płaskiego terenu, a dopiero później dodamy generację nierówności terenu. Pierwszy krok umożliwia zrozumienie podstaw generacji terenu, które potem można dowolnie rozszerzać.

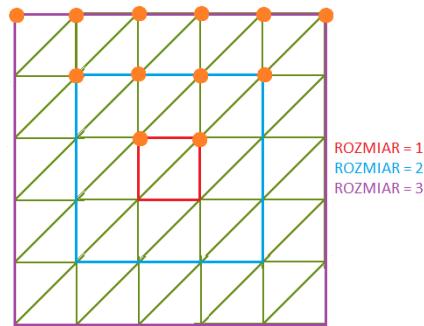
Etap 1:

Zacznijmy od napisania funkcji, która zajmie się generowaniem terenu o podanym rozmiarze.

```
GLuint generateTerrain(unsigned short n, unsigned short cell_size, int &elements, int &cells)
{
    int vertices_count = 2 + (n - 1) * 2;
    // ...
```

}

Przyjmuje ona cztery parametry. W pierwszym określamy rozmiar terenu, który jest następnie przeliczany na liczbę wierzchołków występującą w jednym wierszu (lub kolumnie) terenu (rys. 35.2).



Rys. 35.2. Różne rozmiary terenu i odpowiadająca im liczba wierzchołków w wierszu.

Jeśli chcemy wygenerować tylko jeden kwadrat (pole) terenu, to w jednym wierszu będą 2 wierzchołki. Przy rozmiarze terenu równym 2, wierzchołków będzie 4, a przy rozmiarze terenu równym 3, będzie ich 6. Liczba tych wierzchołków obliczana jest za pomocą wzoru na ciąg arytmetyczny i wynik przypisywany jest do zmiennej `vertices_count`.

Drugi parametr służy do określenia rozmiaru pojedynczego kwadratu (pola) terenu. Trzeci i czwarty parametr posłużą do zwrócenia danych, które będą potrzebne dalej do narysowania terenu.

Dalej w funkcji definiujemy kontenery na dane.

```
std::vector<float> positions;
std::vector<float> normal_vectors;
std::vector<float> texture_coords;

std::vector<int> indices;
```

Ostatni kontener będzie przechowywał indeksy wierzchołków. Niedługo stanie się jasne po co on nam jest.

Skoro wiemy ile wierzchołków powinno być w jednym wierszu (oraz kolumnie, gdyż jest to teren kwadratowy), możemy napisać dwie pętle, które „przejdą” po wszystkich wierzchołkach i je wygenerują.

```
for (int w = 0; w < vertices_count; w++)
{
    for (int k = 0; k < vertices_count; k++)
    {
        float vertex_x = k * cell_size;
        float vertex_y = 0.0;
        float vertex_z = w * cell_size;

        positions.push_back(vertex_x);
        positions.push_back(vertex_y);
        positions.push_back(vertex_z);

        float vector_x = 0.0;
        float vector_y = 1.0;
        float vector_z = 0.0;
```

```

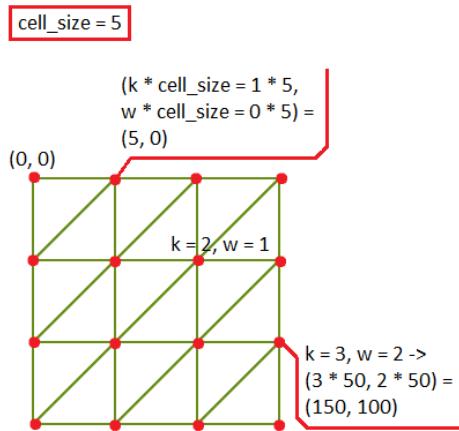
    normal_vectors.push_back(vector_x);
    normal_vectors.push_back(vector_y);
    normal_vectors.push_back(vector_z);

    float s = (1.0 / (vertices_count - 1)) * k;
    float t = (1.0 / (vertices_count - 1)) * w;

    texture_coords.push_back(s);
    texture_coords.push_back(t);
}
}

```

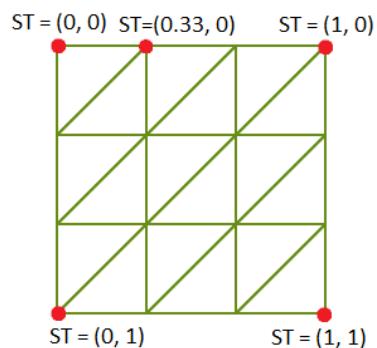
Myślę, że zrozumienie tego algorytmu nie powinno być problemem. Współrzędne wierzchołka X oraz Z obliczamy jako iloczyn indeksu wierzchołka w wierszu (lub kolumnie) oraz rozmiaru pojedynczego pola terenu (rys. 35.3). Współrzędnej Y na razie nie obliczamy, gdyż generujemy płaski teren.



Rys. 35.3. Obliczenie współrzędnych wierzchołka terenu.

Skoro mamy płaski teren, to wektor normalny dla każdego wierzchołka jest taki sam i jest równy $[0, 1, 0]$, czyli jest to wektor skierowany pionowo do góry wzduł osi Y.

Współrzędne tekstury rozciągamy tak jak na rys. 35.4. Wierzchołki znajdujące się w narożnikach terenu otrzymują skrajne wartości współrzędnych, czyli 0 lub 1. Z kolei wierzchołki znajdujące się pomiędzy wierzchołkami skrajnymi, otrzymują wartości pośrednie.



Rys. 35.4. Przypisane współrzędne tekstury do terenu.

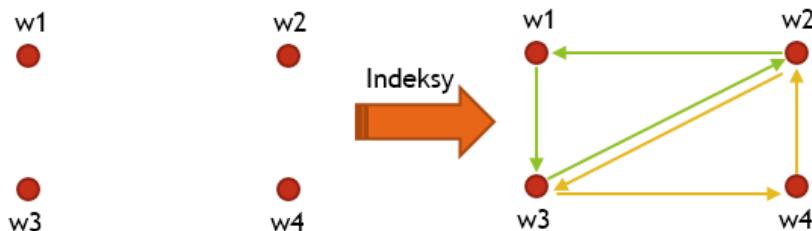
Zauważysz może, że generujemy dużo mniej wierzchołków niż powinniśmy, aby potem poprawnie połączyć je w trójkąty (czyli po 3 wierzchołki na trójkąt). Spójrz jeszcze raz na

rys. 35.3. Mamy tutaj 18 trójkątów, a zatem powinniśmy wygenerować 54 wierzchołki, po trzy wierzchołki na każdy trójkąt. Wierzchołki wewnętrzne terenu są współdzielone przez 6 trójkątów. Oznacza to, że musielibyśmy wygenerować niektóre wierzchołki po 6 razy. Istna strata pamięci.

Można to zrobić o wiele łatwiej i bardziej mniej „zasobozernie”. Wystarczy, że zdefiniujemy tylko 16 wierzchołków (czerwone kropki na rys. 35.3), a potem określmy jak OpenGL ma je połączyć podczas rysowania. Do tego celu służą indeksy wierzchołków.

Tablica indeksów wierzchołków jest to zwykła tablica liczb typu `int`, w której określa się jakby kolejność rysowania wierzchołków. Posiadamy oddzielnny bufor wierzchołków oraz oddzielnny bufor indeksów. Kolejna wartość pobrana z bufora indeksów określa, który wierzchołek należy teraz narysować. Jeśli jest to trzeci narysowany z kolei wierzchołek oraz jako prymitywy rysowane są trójkąty (`GL_TRIANGLES`), to rysowany jest nowy trójkąt i proces zaczyna się od początku - pobierany jest kolejny wierzchołek na podstawie kolejnej wartości z bufora indeksów (rys. 35.5).

$$\begin{aligned} W &= \{w_1, w_2, w_3, w_4\} \\ I &= \{1, 3, 2, 2, 3, 4\} \end{aligned}$$



Rys. 35.5. Ilustracja działania indeksów wierzchołków.

Poniższa pętla ma za zadanie przejść jeszcze raz po wszystkich wygenerowanych wierzchołkach i stworzyć odpowiednią tablicę indeksów (rys. 35.6).

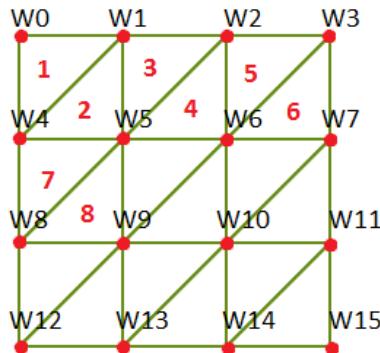
```
for (int w = 0; w < vertices_count - 1; w++)
{
    for (int k = 0; k < vertices_count - 1; k++)
    {
        int index_1 = w * vertices_count + k;
        int index_2 = index_1 + 1;
        int index_3 = (w + 1) * vertices_count + k;
        int index_4 = index_3 + 1;

        indices.push_back(index_1);
        indices.push_back(index_3);
        indices.push_back(index_2);

        indices.push_back(index_2);
        indices.push_back(index_3);
        indices.push_back(index_4);
    }
}
```

I tak, trójkąt nr 1 tworzony jest przez wierzchołki W0, W4 oraz W1. Trójkąt nr 2 tworzony jest przez wierzchołki W1, W4 oraz W5. A zatem tablica indeksów po wpisaniu tych dwóch

trójkątów będzie wyglądała następująco $\{0, 4, 1, 1, 4, 5\}$. Tak samo powstają indeksy dla pozostałych trójkątów.



Rys. 35.6. Indeksowanie wierzchołków.

To gdzie tu oszczędność, skoro i tak musimy stworzyć drugą tablicę? Jeśli mielibyśmy 54 wierzchołki, to mielibyśmy $54 \cdot 3 = 162$ zmiennych typu float. Każda zmienna typu float zajmuje 4 bajty, a zatem jest to $162 \cdot 4 = 648$ bajtów. Korzystając z tablicy indeksów mamy tylko 16 wierzchołków, czyli jest to 192 bajty. Dodatkowo na każdy trójkąt przypadają trzy indeksy (zmienne int, rozmiar int to 2 bajty), czyli jest to $18 \cdot 3 \cdot 2 = 108$ bajtów. W sumie uzyskujemy $192 + 108 = 300$ bajtów. To jest ok. 54% mniej danych! Różnica diametralna.

Na koniec zostaje już tylko nam wrzucić wszystkie dane do obiektu VAO. Postępujemy tak jak zawsze, z małą różnicą. Musimy przecież dodać bufor odpowiedzialny za indeksy wierzchołków. Tworzymy dla niego oddzielnny obiekt VBO, a następnie bindujemy go. Nie bindujemy jak zawsze do obiektu GL_ARRAY_BUFFER, a do obiektu GL_ELEMENT_ARRAY_BUFFER. Dla tablicy indeksów nie używamy funkcji glVertexAttribPointer().

```

GLuint position_vbo = 0;
 glGenBuffers(1, &position_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
 glBufferData(GL_ARRAY_BUFFER, positions.size() * sizeof(GLfloat), positions.data(),
              GL_STATIC_DRAW);

GLuint normal_vector_vbo = 0;
 glGenBuffers(1, &normal_vector_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, normal_vector_vbo);
 glBufferData(GL_ARRAY_BUFFER, normal_vectors.size() * sizeof(GLfloat), normal_vectors.data(),
              GL_STATIC_DRAW);

GLuint texture_coords_vbo = 0;
 glGenBuffers(1, &texture_coords_vbo);
 glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo);
 glBufferData(GL_ARRAY_BUFFER, texture_coords.size() * sizeof(GLfloat), texture_coords.data(),
              GL_STATIC_DRAW);

GLuint indices_vbo = 0;
 glGenBuffers(1, &indices_vbo);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices_vbo);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(int), indices.data(),
              GL_STATIC_DRAW);

GLuint handle = 0;
 glGenVertexArrays(1, &handle);
 glBindVertexArray(handle);

```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices_vbo);

glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, normal_vector_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, 0);

glBindVertexArray(0);
```

Funkcja powinna zwracać uchwyt do obiektu VAO oraz poprzez parametr `elements` liczbę indeksów wierzchołków. Dodatkowo zwracamy także liczbę kwadratów (pół) w jednym wierszu (zmienna `cells`). Przykładowo, na rys. 35.6, są 4 wierzchołki w jednym wierszu, a zatem $4 - 1 = 3$ pola terenu.

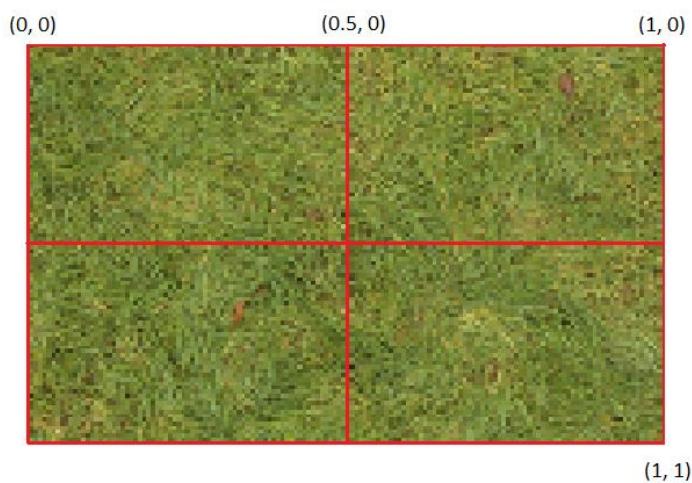
```
elements = indices.size();
cells = vertices_count - 1;
return handle;
```

Nasza funkcja generująca teren jest już gotowa, możemy jej użyć.

```
int elements = 0;
int cells = 0;
GLuint terrain handle = generateTerrain(3, 5, elements, cells);
// Load texture
unsigned int grass_tex = loadTexture("Grass.jpg");
```

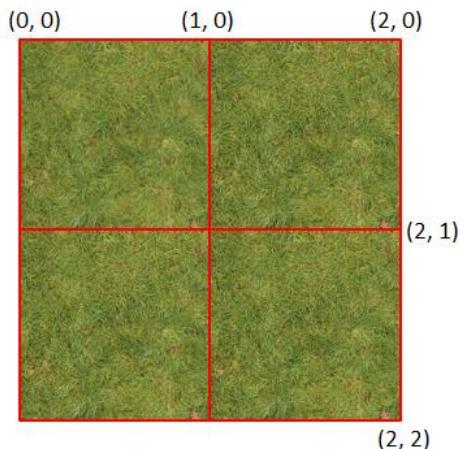
Teksturę możemy załadować w dowolny sposób, tak jak robimy to zawsze. Konieczne może okazać się tylko wygenerowanie mipmapy oraz ustawienie jakiegoś filtrowania, żeby tekstura lepiej wyglądała.

Renderowanie terenu wymaga także oddzielnych shaderów. Ponadto, musimy przesyłać do programu shadera liczbę typu `int` określającą ile pół (kwadratów) jest w jednym wierszu terenu (zmienna `cells`). Wartość ta wykorzystywana jest w celu przeskalowania współrzędnych tekstuury. Domyślnie ustawiliśmy, że współrzędne tekstuury mogą zmieniać się od 0 do 1. W takiej sytuacji, jeśli nasz teren będzie ogromny, tekstura rozciągnie się na cały teren tworząc brzydkie efekt (piksela) (rys. 35.7).



Rys. 35.7. Piksele widoczne na rozciagniętej na duży obszar tekstuurze.

Jeśli przemnożymy współrzędne tekstury przez pewną wartość, np. 2, uzyskamy wtedy dwie sąsiadujące obok siebie takie same tekstury. Jeśli przemnożymy przez 3, to uzyskamy trzy tekstury obok siebie, itd. (rys. 35.8). W naszym przypadku istnieje konieczność przemnożenia współrzędnych tekstury przez liczbę pól terenu. W ten sposób każde pole terenu będzie teksturowane osobno.



Rys. 35.8. Przemnożenie współrzędnych tekstury mnozuje teksturę.

Vertex shader dla terenu wygląda następująco:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal_vector;
layout(location = 2) in vec2 vt;

uniform int cells;
uniform mat4 view_matrix;
uniform mat4 perspective_matrix;

out vec2 texture_coordinates;
out vec3 vertex_to_camera;
out vec3 normal_to_camera;

void main()
{
    normal_to_camera = vec3(view_matrix * vec4(normal_vector, 0.0));
    vertex_to_camera = vec3(view_matrix * vec4(position, 1.0));

    texture_coordinates = cells * vt;
    gl_Position = perspective_matrix * view_matrix * vec4(position, 1.0);
}
```

Zmienna `texture_coordinates`, o czym była mowa wyżej, jest obliczana poprzez przemnożenie współrzędnych tekstury przez liczbę pól terenu.

W fragment shader mamy dowolność, zależy wszystko od tego jak chcemy kolorować teren oraz czy chcemy uwzględniać oświetlenie. Na przykład może on wyglądać następująco:

```
#version 330
in vec2 texture_coordinates;
in vec3 vertex_to_camera;
in vec3 normal_to_camera;

uniform mat4 view_matrix;
uniform sampler2D basic_texture;

out vec4 frag_colour;
```

```

vec3 light direction = vec3(0.0, -1.0, 0.0);
vec3 ambient color = vec3(0.2, 0.2, 0.2);
vec3 diffuse color = vec3(0.8, 0.8, 0.8);

vec3 object_ambient_factor = vec3(1.0, 1.0, 1.0);
vec3 object diffuse factor = vec3(0.6, 0.6, 0.6);

void main()
{
    // Ambient
    vec3 ambient_intense = ambient_color * object_ambient_factor;

    // Diffuse
    vec3 light direction CAMERA = (view matrix * normalize(vec4(light direction, 0.0))).xyz;
    vec3 direction to light = -light direction CAMERA;
    float dot_product = dot(direction_to_light, normal_to_camera);
    dot_product = max(dot_product, 0.0);
    vec3 diffuse intense = diffuse color * object diffuse factor * dot product;

    vec4 texel = texture(basic texture, texture coordinates);
    frag_colour = vec4(ambient_intense + diffuse_intense, 1.0) * texel;
}

```

Teraz wystarczy, że w kodzie programu aktywujemy program shadera odpowiedzialny za rysowanie terenu i ustawimy odpowiednie zmienne uniform.

```

GLuint terrain_shader = 0;
createShaderProgram(terrain shader, "vertex shader terrain.glsl",
                    "fragment shader terrain.glsl");

GLint view_uniform_terrain = findUniform(terrain shader, "view_matrix");
GLint perspective_uniform_terrain = findUniform(terrain_shader, "perspective_matrix");
GLint texture_slot_terrain = findUniform(terrain_shader, "basic_texture");
GLint cells_terrain = findUniform(terrain shader, "cells");

// ...
activateShaderProgram(terrain_shader);
setUniform(view_uniform_terrain, view_matrix);
setUniform(perspective_uniform_terrain, perspective);
setUniform(texture_slot_terrain, 0);
setUniform(cells_terrain, cells);

```

Wszystko jest już gotowe i możemy narysować nasz wygenerowany teren. Z tego względu, że korzystamy z bufora indeksów, musimy skorzystać z funkcji `glDrawElements()` zamiast funkcji `glDrawArrays()`. Przyjmuje ona cztery parametry:

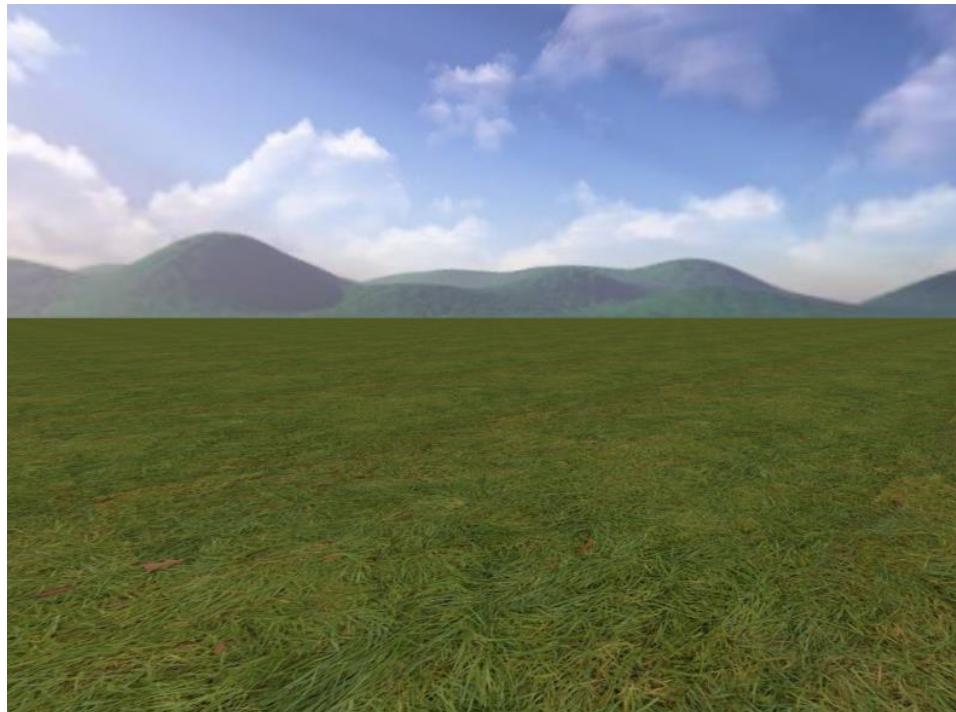
- typ rysowanych prymitywów,
- liczba indeksów wierzchołków,
- typ danych w buforze,
- adres położenia bufora indeksów; u nas niewykorzystywane, gdyż indeksy umieściliśmy w oddzielnym obiekcie VBO.

```

glBindVertexArray(terrain handle);
 glEnableVertexAttribArray(0);
 glEnableVertexAttribArray(1);
 glEnableVertexAttribArray(2);
 glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, grass_tex);
 glDrawElements(GL_TRIANGLES, elements, GL_UNSIGNED INT, 0);
 glDisableVertexAttribArray(0);
 glDisableVertexAttribArray(1);
 glDisableVertexAttribArray(2);
 glBindVertexArray(0);

```

W wyniku otrzymamy rezultat jak na rys. 35.9.

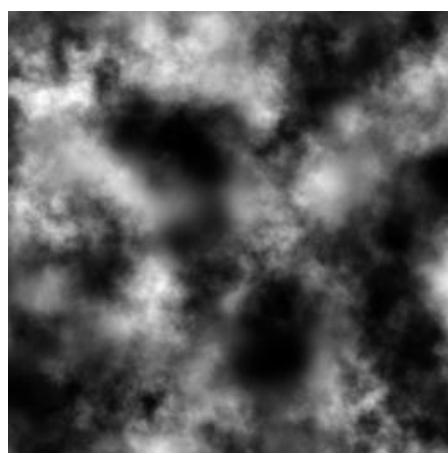


Rys. 35.9. Wygenerowany płaski teren.

Etap 2:

Gdy już wszystko działa poprawnie możemy teraz dodać nierówności do naszego generatora terenu.

W tym celu wykorzystamy, tzw. mapę wysokości (**heightmap**) (rys. 35.10). Jest to obraz w skali szarości, który służy do opisywania przemieszczania (wysokości), np. wierzchołka. 8-bitowa mapa wysokości może reprezentować 256 poziomów wysokości, a 24-bitowa mapa wysokości dostarcza informacji o aż 256^3 poziomach wysokości.



Rys. 35.10. Przykładowa mapa wysokości.

Im dany piksel jest ciemniejszy, tym jest położony niżej. Piksel całkowicie biały oznacza najwyższy poziom wysokości, a piksel czarny poziom najniższy.

Zacznijmy od zmodyfikowania deklaracji funkcji generującej. Nowa deklaracja tej funkcji jest następująca:

```
GLuint generateTerrain(std::string height_map, unsigned short cell_size, int &elements, int &cells)
{
```

```
}
```

Zmianie uległ pierwszy parametr. Tym razem funkcja musi przyjmować ścieżkę do ładowanej mapy wysokości. Nie potrzebujemy już także parametru określającego rozmiar terenu, gdyż liczba generowanych wierzchołków w jednym wierszu będzie teraz równa szerokości mapy wysokości (liczba pikseli w jednym wierszu lub kolumnie mapy wysokości).

Musimy dalej załadować plik z mapą wysokości do programu.

```
Texture height_map_tex;
loadTexture(height_map, height_map_tex);
```

Potrzebna będzie także funkcja, która pobierze kolor konkretnego piksela z mapy wysokości oraz obliczy na tej podstawie wysokość wierzchołka. Chciałbym zaznaczyć, że wysokość wierzchołka można obliczyć na dowolny sposób w zależności od naszych potrzeb, bądź upodobań. Niektóre metody generują bardziej strome zbocza, a niektóre bardziej łagodne. Ja przedstawiam metodę, która najprawdopodobniej będzie stosowana najczęściej.

```
float calculateHeight(Texture *texture, int x, int y)
{
    const float MAX_HEIGHT = 30.0f;

    RGBQUAD colour;
    FreeImage_GetPixelColor(texture->image_ptr, x, y, &colour);

    float height = colour.rgbRed * colour.rgbGreen * colour.rgbBlue;

    if (height > 256 * 256 * 256 / 2)
        height = height;
    else
        height = -height;

    height /= 256 * 256 * 256 / 2;
    height *= MAX_HEIGHT;

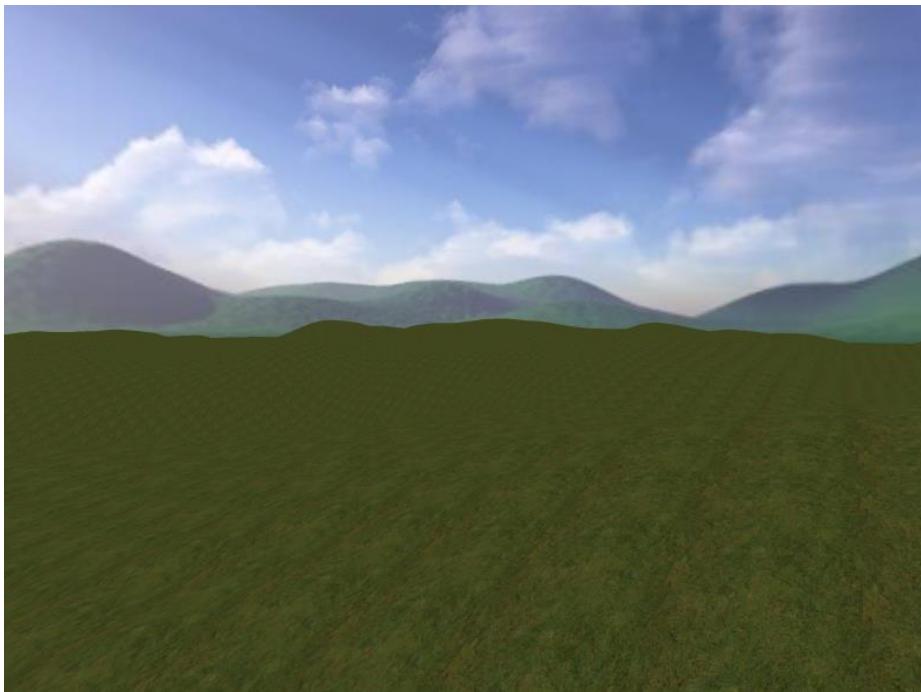
    return height;
}
```

Jest to funkcja, która przyjmuje na wejściu mapę wysokości oraz współrzędne piksela. W jej wnętrzu posługujemy się funkcją `FreeImage_GetPixelColor()`, która pobiera kolor piksela z pliku graficznego. Na podstawie pobranej wartości przeprowadzane są obliczenia. W pierwszej kolejności przemnażamy przez siebie trzy składowe koloru RGB (jest to obraz w skali szarości, więc wszystkie będą sobie równe). Dzięki temu zwiększamy zakres możliwych poziomów wysokości, czyli tak jakby zwiększamy rozdzielczość. Następnie sprawdzamy, czy obliczona wysokość jest większa od połowy zakresu. Jeśli nie, to oznacza to, że wysokość powinna być ujemna. Kolejna operacja przeskala wysokość do zakresu od -1 do +1. Na samym końcu przemnażamy uzyskaną wysokość przez współczynnik określający maksymalną wysokość (wzmocnienie).

Teraz wystarczy, że jako współrzędną Y wierzchołka przypiszemy obliczoną właśnie wysokość.

```
float vertex_x = k * cell_size;
float vertex_y = calculateHeight(&height map tex, k, w);
float vertex_z = w * cell_size;
```

Reszta pozostaje już bez zmian. Jeśli teraz uruchomimy nasz program, zauważymy teren z wygenerowanymi nierównościami (rys. 35.11). Niestety, teren nie jest poprawnie oświetlony, nie zmieniliśmy przecież wektorów normalnych. Nadal są skierowane pionowo do góry.



Rys. 35.11. Teren z nierównościami bez skorygowanych wektorów normalnych.

Jest kilka metod korekcji wektorów normalnych. Ja używam następującej (chyba najbardziej popularnej):

$$\begin{aligned} hL &= \text{height}(x - 1, z) \\ hR &= \text{height}(x + 1, z) \\ hD &= \text{height}(x, z - 1) \\ hU &= \text{height}(x, z + 1) \end{aligned}$$

$$\begin{aligned} N &= [hL - hR, 2, hD - hU] \\ N &= \text{normalize}(N) \end{aligned}$$

Metoda ta oparta jest o obliczenie wysokości sąsiednich pikseli. Na tej podstawie możliwe jest wyznaczenie nowego wektora normalnego, który będzie skierowany poprawnie. Możemy zatem napisać funkcję, która obliczy wektory normalne.

```
glm::vec3 calculateNormal(Texture *texture, int x, int z)
{
    float hL = calculateHeight(texture, x - 1, z);
    float hR = calculateHeight(texture, x + 1, z);
    float hD = calculateHeight(texture, x, z - 1);
    float hU = calculateHeight(texture, x, z + 1);

    glm::vec3 new_normal_vector = glm::vec3(hL - hR, 2.0, hD - hU);
```

```
    glm::normalize(new_normal_vector);
    return new_normal_vector;
}
```

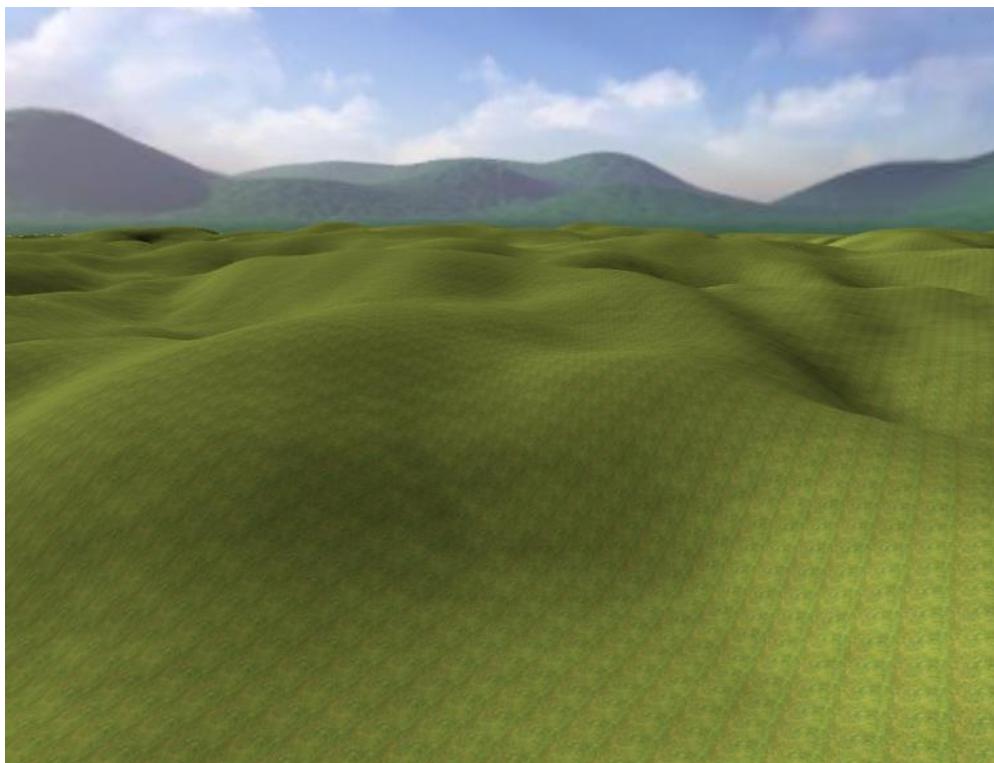
W funkcji tej wykorzystujemy naszą funkcję do obliczenia wysokości i obliczamy wysokości sąsiadujących pikseli. Następnie wykonujemy obliczenia według wzoru i na końcu zwracamy nowy wektor normalny.

Ostatnią zmianą w funkcji generującej teren jest zatem ustawienie nowego wektora normalnego dla wierzchołka.

```
auto normal = calculateNormal(&height_map_tex, k, w);

float vector_x = normal.x;
float vector_y = normal.y;
float vector_z = normal.z;
```

Nie musimy już wykonywać więcej modyfikacji. Jeśli teraz uruchomimy nasz program, cieniowanie powinno już działać poprawnie (rys. 35.12). Możemy pomanewrować wartością zmiennej MAX_HEIGHT w celu osiągnięcia mniejszych bądź większych nierówności.



Rys. 35.12. Wygenerowany teren z poprawnym oświetleniem.

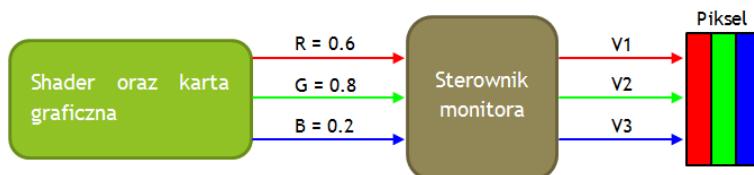
Kod źródłowy z tego rozdziału jest w katalogu: [25_Generowanie_terenu](#)

36. Korekcja gamma

Z reguły zależy nam na tym, aby renderowany obraz odzwierciedlał jak najlepiej rzeczywistość. Jeśli jednak na obrazie powstają nienaturalne kolory, albo jest on prześwietlony, nasze oko od razu wychwytuje różnice. Istnieje pewna technika, nazywana **korekcją gamma**, która służy do poprawy jakości obrazu zerowym kosztem. Niestety jest często zapominana i niedoceniana. Czas się jej przyjrzeć bliżej.

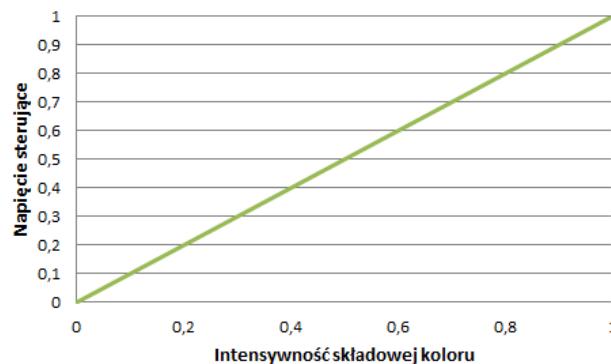
Pracując z komputerem obraz wyświetlamy oczywiście na monitorze. Nie ważne czy jest to monitor CRT, czy LCD, zasada działania jest taka sama. Światło emitowane z monitora dociera do naszego oka. Każdy piksel obrazu może wyświetlać trzy składowe koloru: czerwony, zielony i niebieski (RGB). Intensywności poszczególnych składowych koloru mieszają się ze sobą i w efekcie do naszego oka dociera światło o pełnej (no prawie) paletie barw.

Zwróćmy uwagę na to, jak definiowaliśmy kolory do tej pory. Na przykład w programie shadera zmiennej `frag_colour` przypisywaliśmy trzy wartości w przedziale od 0 do 1, które stanowiły intensywności poszczególnych składowych koloru. Wartości te były następnie wysyłane do monitora poprzez kartę graficzną. Każda składowa koloru piksela w monitorze jest sterowana oddzielnie. Po otrzymaniu trzech wartości informujących o wymaganym poziomie intensywności, monitor przelicza je na odpowiedni poziom napięcia, który służy do wysterowania intensywności świecenia poszczególnej składowej koloru piksela (rys. 36.1).



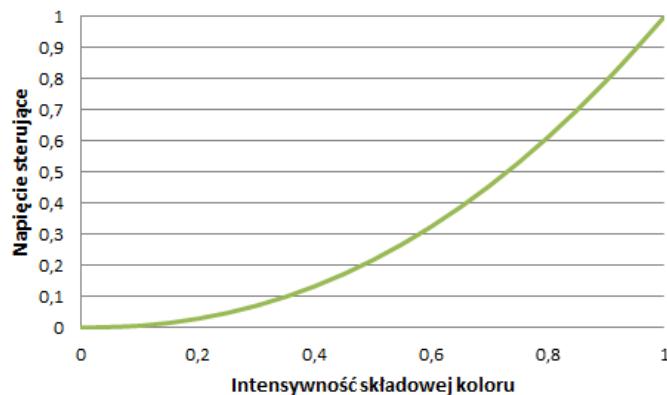
Rys. 36.1. Symboliczny schemat wysyłania koloru do monitora.

W idealnym świecie najlepszą zależnością napięcia sterującego intensywnością świecenia piksela od podanej intensywności koloru byłaby zależność liniowa (rys. 36.2).



Rys. 36.2. Idealna zależność napięcia sterującego pikselem od intensywności składowej koloru.

Niestety, nie ma lekko, nie żyjemy w świecie idealnym. Rzeczywista zależność napięcia sterującego od intensywności składowej koloru wygląda bardziej tak jak na rys. 36.3.



Rys. 36.3. Rzeczywista zależność napięcia sterującego pikselem od intensywności składowej koloru.

Powyższa krzywa jest opisana równaniem $f(x) = x^{2,2}$, gdzie wykładnik potęgi równy 2.2 jest wartością **gamma**. W nowych monitorach wartość gamma mieści się najczęściej w przedziale od 2.0 do 2.4, ale standardowo jest to wartość równa 2.2. Możemy zatem powiedzieć, że **wartość gamma służy do opisywania charakterystyki monitora**.

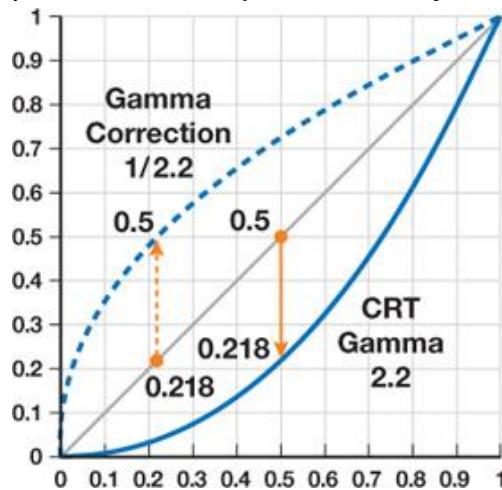
Jeśli przyjrzymy się krzywej na rys. 36.3 zauważymy pewien problem. Przy małych wartościach intensywności (do ok. 0.5), następuje bardzo mała zmiana napięcia sterującego. W związku z tym, dla tych intensywności (kolorów) będziemy uzyskiwać „ciemniej” świecące piksele (przeczernienie) (rys. 36.4).



Rys. 36.4. Obraz jest przeczerniony z lewej strony, kolory są tam ciemniejsze. Z prawej strony kolory są prawidłowe.

Dochodzimy teraz do sedna sprawy. A gdyby tak wprowadzić jakąś kompensację tej nielinowej charakterystyki monitora? I to jest właśnie **korekcja gamma**. Krótko mówiąc, polega ona na wprowadzeniu charakterystyki o **współczynniku gamma równym $1/\gamma$** (np. $1/2,2$) tak, aby w sumie po nałożeniu na charakterystykę monitora uzyskać charakterystykę liniową (rys. 36.5).

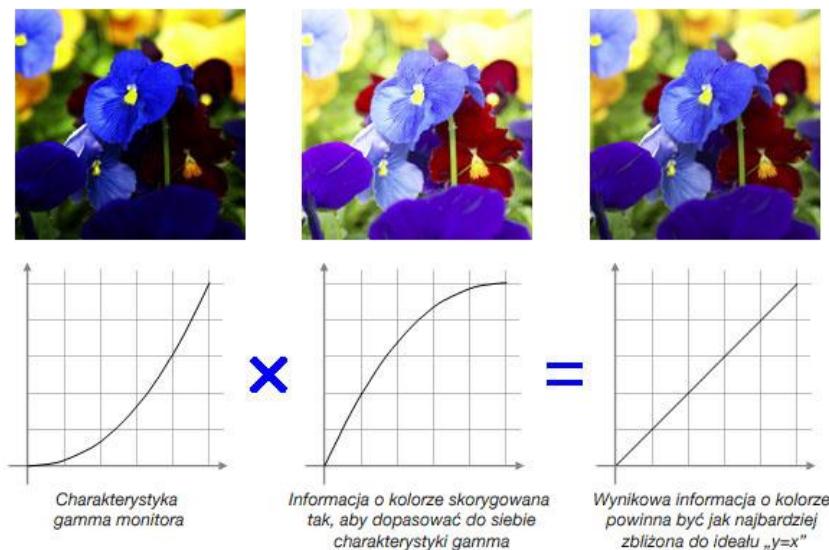
Wykres na rys. 36.5 jest standardowym wykresem opisu charakterystyki monitora. Na wykresie tym zaznaczono także, że piksel o intensywności 50% emitemuje mniej niż $\frac{1}{4}$ światła (bo 0.218) w porównaniu z pikselem o intensywności równej 100%.



Rys. 36.5. Przykładowy wykres charakterystyki monitora.

Krzywa oznaczona linią przerywaną na rys. 36.5 jest **krzywą korekcyjną**. Jak wspomniałem wyżej, współczynnik tej krzywej wynosi $1/\gamma$, czyli $1/2.2=0.454$.

Jeśli teraz będziemy coś rysować (renderować) albo w jakiś inny sposób pracować z grafiką komputerową, musimy wprowadzać korektę kolorów według krzywej korekcyjnej (przerywana linia na rys. 36.5). Wyświetlając obraz na monitorze zostanie automatycznie uwzględniona charakterystyka monitora, a w efekcie otrzymamy charakterystykę liniową (rys. 36.6).



Rys. 36.6. Korekcja gamma obrazu (źródło obrazka: [LINK](#)).

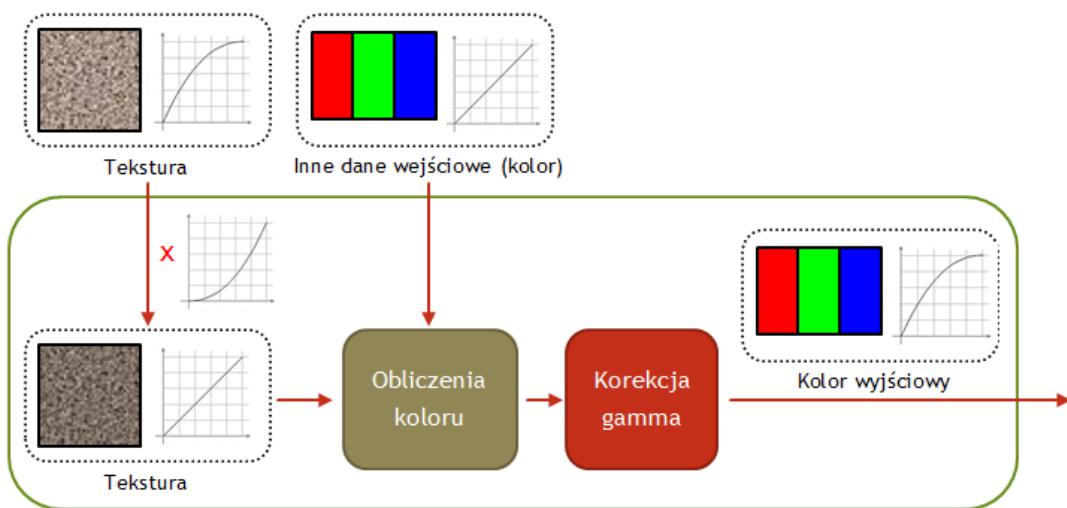
Należy jeszcze podkreślić, że kolor czarny (0.0, 0.0, 0.0) oraz biały (1.0, 1.0, 1.0) nigdy nie ulegają korekcji. To widać także na krzywej z rys. 36.5. Tylko kolory pośrednie ulegają korekcji.

Korekcję gamma na swoim komputerze można wprowadzić za pomocą różnych narzędzi systemu operacyjnego bądź zewnętrznych programów. Niestety, prawie nikt tego nie robi. Z tego założenia wychodzą też twórcy programów graficznych. Wszystkie pliki PNG, JPEG, itp., są już po korekcji gamma, a więc intensywności składowych koloru są wyliczane wg krzywej $1/\gamma$. Wyświetlając taki skorygowany obrazek na monitorze (nieskalibrowanym) otrzymujemy poprawne kolory.

Zastanówmy się teraz, w jaki sposób obliczamy kolory podczas renderowania. Mamy na przykład jeden wektor trójwymiarowy, który reprezentuje jakiś kolor, np. kolor powierzchni. Mamy także drugi wektor, który reprezentuje kolor oświetlenia. Teraz te dwa wektory możemy do siebie dodawać albo przemnażać. Ale to wszystko są operacje liniowe. Jeśli wewnątrz shadera dodamy do siebie wartość 0.1 oraz 0.3, to oczekujemy, że otrzymamy 0.4. I taką też wartość dostaniemy. Wynika z tego to, że **shadery działają na danych liniowych, a nie na nieliniowych**.

Skoro pliki graficzne (PNG, JPG, itp.), czyli tekstury, które używaliśmy do tej pory są nieliniowe, a shader pracuje na danych liniowych, to w wyniku otrzymywaliśmy źle obliczone kolory.

Poprawny potok obliczeń danych (kolorów) w shaderze powinien być jak na rys. 36.7. Do danych wejściowych shadera należy tekstuра oraz jakiś kolor (przekazywany np. przez zmienną uniform). Tekstura jest wyrażona w przestrzeni nieliniowej, a kolor w liniowej (po prostu jakieś trzy składowe bez żadnej korekcji). Po przyjęciu tekstuury w programie shadera oraz pobraniu z niej kolejnego teksta, musimy otrzymany kolor zlinearyzować. Gdy wszystkie dane wejściowe są już w przestrzeni liniowej można na nich wykonywać obliczenia, np. oświetlenia. Na sam koniec na kolor wyjściowy należy nałożyć korekcję gamma.



Rys. 36.7. Schematycznie przedstawiony potok danych wraz z korekcją gamma wewnątrz shadera.

Pokażę teraz jak zaimplementować korekcję gamma w OpenGL. Jest to dziecinne proste.

Jeśli odczytujemy tekstuwę w fragment shader musimy pamiętać o tym, aby zlinearyzować odczytany kolor z tekstuły.

```
vec4 diffuse_texel = texture(diffuse_texture, texture_st);
diffuse_texel.rgb = pow(diffuse_texel.rgb, vec3(2.2, 2.2, 2.2));
```

Teraz możemy już wykonywać dowolne obliczenia, np. dodawać do siebie, przemnażać przez jakąś wartość. Obliczenia zostaną wykonane poprawnie.

Po obliczeniu całkowitego koloru fragmentu musimy wprowadzić jeszcze dla niego korekcję gamma:

```
frag_colour.rgb = pow(final_colour.rgb, vec3(1.0 / 2.2, 1.0 / 2.2, 1.0 / 2.2));
```

W dwóch krótkich krokach poprawiliśmy jakość obrazu. Kolory nie będą już przeczernione. Paleta barw będzie szersza oraz kolory będą żywsi. Rys. 36.8 przedstawia scenę bez korekcji gamma, a na rys. 36.9 wprowadzona już została korekcja. Ewentualnie drugi obrazek wygląda dużo lepiej.



Rys. 36.8. Scena bez korekcji gamma.



Rys. 36.9. Scena po korekcji gamma.

Podsumowując, za pomocą korekcji gamma korygujemy przestrzeń barw. Sprawiamy, że kolory stają się bardziej naturalne. Pamiętać należy jednak o kilku rzeczach:

- shadery są liniowe, pracują na danych liniowych,
- kolory wpisywane przez nas ręcznie w kodzie, np. [0.3, 0.2, 0.6], są już liniowe, nie należy ich korygować,
- tekstury są nieliniowe, trzeba je pierw zlinearyzować, aby dalej na nich pracować,
- należy uważać, aby nie wykonać korekcji gamma kilkukrotnie, gdyż otrzymamy wtedy dziwne efekty.

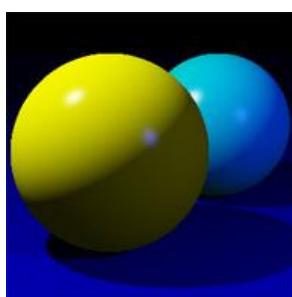
Korekcja gamma może przysporzyć dużo problemów z dobrym jej zrozumieniem. Mam nadzieję, że ten uproszczony w dużym stopniu opis pozwoli zrozumieć podstawy tej techniki.

37. Odbicie oraz refrakcja

Odbicie oraz refrakcja są jednymi z podstawowych zjawisk światła, z którymi mamy do czynienia. Jeśli powierzchnia obiektu jest błyszcząca, to powstaje odbicie światła. Jeśli natomiast powierzchnia jest przezroczysta, np. szkło lub woda, wtedy zachodzi refrakcja. Zaimplementowanie tych dwóch zjawisk w znacznym stopniu może wizualnie urozmaicić scenę.

Odbicie

W oświetleniu Phonga jeśli chcemy sprawić, aby jakaś powierzchnia wyglądała na gładką, to ustawiamy jej wysoki współczynnik odbicia światła (specular). Światło o jakimś kolorze pada na powierzchnię pod pewnym kątem i odbija się od niej. Wynikiem są jasne odblaski na powierzchni o takim samym kolorze jak światło padające (rys. 37.1).



Rys. 37.1. Odblaski światła (specular).

W świecie rzeczywistym jeśli jakiś obiekt jest gładki, to nie tylko powstają na nim odblaski światła, ale również zachowuje się w pewnym stopniu jak lustro (rys. 37.2), np. karoseria samochodu, chromowana kulka, itp.



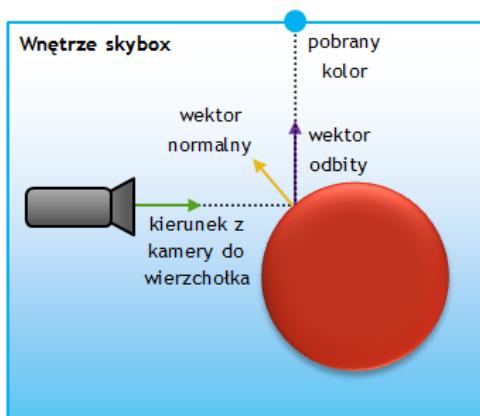
Rys. 37.2. Lustrzane odbicia w niektórych obiektach.

Aby przedstawić samą ideę odbicia, zaimplementujemy odbicie w obiekcie tylko obiektu skybox. Zaimplementowanie odbicia wszystkich otaczających obiektów nie jest trywialne. Należy postugiwać się wtedy między innymi techniką, która opiera się na renderowaniu do tekstury. Temat ten zostanie na pewno poruszony w przyszłości, ale na razie uproszczamy zagadnienie.

To, co ulega odbiciu w obiekcie możemy nazwać **mapą otoczenia (environment map)**. Nasza mapa otoczenia będzie statyczna, będzie zawierała tylko tekstury z skybox. Żadne inne obiekty na scenie nie będą uwzględniane.

A co w przypadku, gdy dany obiekt znajduje się w jakimś pomieszczeniu? Wtedy odbijanie skybox'a w obiekcie nie ma sensu. To racja. W takim przypadku można tymczasowo tekstury skybox'a „podmienić” innymi teksturami, które reprezentują to otoczenie. Chodzi tylko o to, żeby **mapa otoczenia była statyczna**.

Możemy już przejść do implementowania odbicia. Spójrzmy jeszcze tylko na rys. 37.3, który pokazuje w jaki sposób będzie obliczany kolor danej powierzchni odbijającej światło. Wektor wychodzący z kamery w kierunku wierzchołka jest odbijany od powierzchni zgodnie z wektorem normalnym tego wierzchołka, a następnie ten wektor odbity jest wykorzystywany do pobrania próbki koloru z skybox. Ostatecznie konkretny teksel jest kolorowany pobranym kolorem.



Rys. 37.3. Pobieranie koloru z mapy otoczenia.

W kodzie aplikacji jedyne, co musimy zrobić to wysyłać do programu shadera teksturę z mapą otoczenia, czyli na przykład skybox. Oczywiście tekstura mapy otoczenia jest wysyłana na innym slocie niż tekstura modelu.

```
GLint texture_slot = findUniform(shader_program, "basic_texture");
GLint env_map = findUniform(shader_program, "env_map");
// ...
glBindVertexArray(mesh_handle);
// Diffuse texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, diffuse_texture);
// Environment map
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_CUBE_MAP, skybox_texture);
```

Kod vertex shader jest następujący:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal_vector;
layout(location = 2) in vec2 vt;

uniform mat4 view_matrix;
uniform mat4 perspective_matrix;
uniform mat4 model_matrix;

out vec2 texture_coordinates;
out vec3 vertex_to_camera;
out vec3 normal_to_camera;

void main()
{
```

```

normal_to_camera = vec3(view_matrix * model_matrix * vec4(normal_vector, 0.0));
vertex_to_camera = vec3(view_matrix * model_matrix * vec4(position, 1.0));

texture_coordinates = vt;
gl_Position = perspective_matrix * view_matrix * model_matrix * vec4(position, 1.0);
}

```

Nic szczególnego nie jest tu robione. Przeliczamy tylko położenie wierzchołka oraz wektor normalny do układu współrzędnych kamery.

W fragment shader zajmujemy się obliczeniami odbicia.

```

vec3 camera_vector = normalize(vertex_to_camera);
vec3 normal_vector = normalize(normal_to_camera);

vec3 reflected_vector = reflect(camera_vector, normal_vector);
reflected_vector = vec3(inverse(view_matrix) * vec4(reflected_vector, 0.0));
vec4 reflected_colour = texture(env_map, reflected_vector);

```

Powyższy kod pozwoli pobrać próbkę koloru z tekstury mapy otoczenia. W pierwszej kolejności potrzebujemy dwóch wektorów: wektora z kamery do wierzchołka oraz wektora normalnego wierzchołka. Obydwa wektory uzyskujemy w łatwy sposób (już je przecież mamy) i normalizujemy je. Następnie korzystamy z znanej już funkcji `reflect()`, która oblicza wektor odbity. Obliczony wektor odbity wyrażony jest w układzie współrzędnym kamery. Jak już też wiemy, kolor z skybox jest próbkowany za pomocą wektora kierunku. Należy zatem przeliczyć wyliczony wektor odbity na współrzędne globalne tak, aby potem można było go wykorzystać jako wektor kierunku do spróbkowania skybox'a (mapy otoczenia). Na sam koniec możemy już pobrać kolor z skybox za pomocą funkcji `texture()` oraz właśnie wyliczonego wektora kierunku.

Teraz uzyskany kolor możemy dowolnie wykorzystać. Możemy przypisać go do aktualnego fragmentu, aby uzyskać efekt jak na rys. 37.4.

```
frag_colour = reflected_colour;
```



Rys. 37.4. Model samochodu pokolorowany tylko kolorem odbitym z mapy otoczenia.

Możemy także uwzględnić teksturę (rys. 37.5):

```

vec4 texel = texture(basic_texture, texture_coordinates);
frag_colour = mix(reflected_colour, texel, 0.6);

```



Rys. 37.5. Model samochodu pokolorowany kolorem odbitym oraz teksturową.

Model wygląda już o wiele bardziej realistycznie. Oczywiście należałoby jeszcze podczas wczytywania modelu wczytywać także materiały powiązane z poszczególnymi elementami modelu. Dzięki temu można byłoby wykorzystać zdefiniowane współczynniki materiału, do określenia czy dany element ma odbijać otoczenie, czy ma być matowy (np. opony). Można potraktować to jako ćwiczenie do wykonania samodzielnie.

Moglibyśmy także dodać oświetlenie (rys. 37.6):

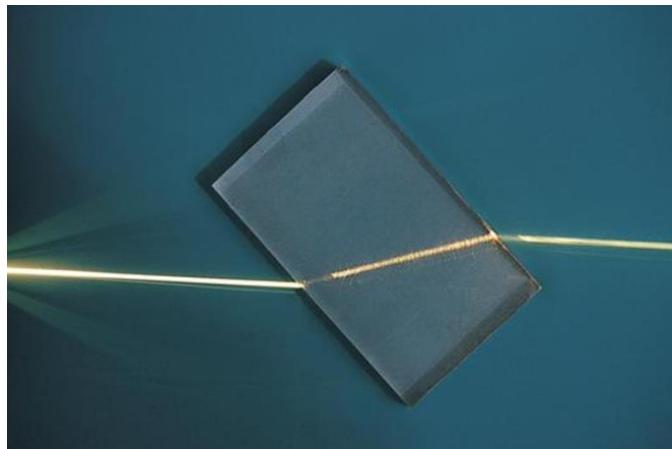
```
frag_colour = vec4(ambient_intense + diffuse_intense + specular_intense, 1.0) *  
mix(reflected_colour, texel, 0.6);
```



Rys. 37.6. Model po dodaniu oświetlenia Phonga.

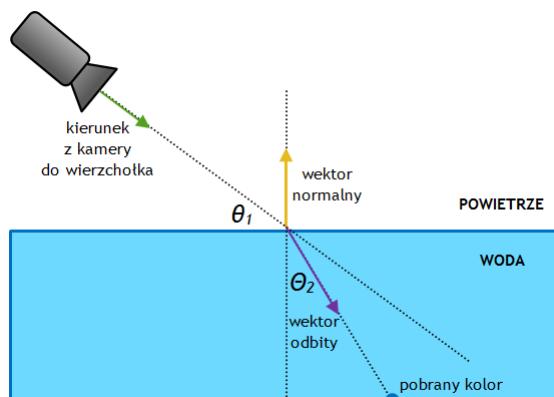
Refrakcja

Refrakcja zachodzi w przezroczystych obiektach. Zjawisko to polega na tym, że światło ulega załamaniu przy przechodzeniu do ośrodka o innej gęstości, np. z powietrza do wody. Refrakcja jest inaczej nazywana po prostu załamaniem światła (rys. 37.7).



Rys. 37.7. Zjawisko refrakcji przy przechodzeniu przez szklany blok.

W OpenGL refrakcję implementujemy podobnie jak odbicie. Zamiast jednak odbijać wektor wychodzący z kamery w kierunku wierzchołka od wektora normalnego, załamujemy go na granicy ośrodków (rys. 37.8).



Rys. 37.8. Pobieranie koloru podłożu w refrakcji.

Na powyższym rysunku (rys. 37.8) refrakcja zachodzi tylko raz. Światło przechodząc z powietrza do wody załamuje się i trafia w dno zbiornika. Jeśli w powietrzu umieścilibyśmy szklaną płytę, refrakcja zaszłaby wtedy dwa razy. Pierwszy raz z powietrza do płyty, a potem z płyty do powietrza (rys. 37.7).

„Stopień ugięcia” padającej wiązki światła zależy od współczynnika załamania światła tych ośrodków, przez które światło przechodzi. Zależność ta jest opisana [prawem Snelliusa](#) i wygląda ona następująco:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} = n_{21}$$

gdzie:

θ_1 - kąt padania,

θ_2 - kąt załamania,

n_1 - współczynnik załamania światła ośrodka pierwszego,

n_2 - współczynnik załamania światła ośrodka drugiego,

n_{21} - wzajemny współczynnik załamania światła ośrodka drugiego względem pierwszego.

Zaimplementowanie refrakcji na tafli wody i pobieranie koloru z dna zbiornika również wymaga zaawansowanej techniki, jaką jest renderowanie do tekstury. To jest tak samo jak z odbiciem, musimy mieć pewną mapę otoczenia, która zostanie spróbkowania w celu pobrania koloru. Na razie takiej mapy otoczenia nie mamy, więc zaimplementujemy bardzo prostą refrakcję w szkle (szzybie). Szyba jest na tyle cienka, że można pominąć podwójną refrakcję. Za mapę otoczenia posłuży nam skybox jak w przypadku odbicia.

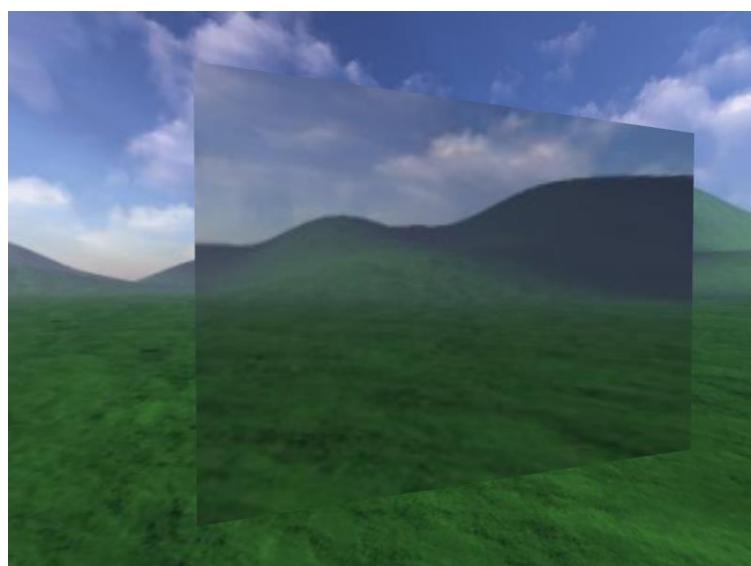
Kod fragment shader będzie następujący:

```
vec3 camera_vector = normalize(vertex_to_camera);
vec3 normal_vector = normalize(normal_to_camera);

float n_ratio = 1.0 / 1.49;
vec3 refracted_vector = refract(camera_vector, normal_vector, n_ratio);
refracted_vector = vec3(inverse(view_matrix) * vec4(refracted_vector, 0.0));
vec4 refracted_colour = texture(env_map, refracted_vector);
```

W pierwszej kolejności definiujemy współczynnik refrakcji. Współczynnik załamania światła dla szkła to ok. 1.49, więc współczynnik refrakcji obliczamy jako iloraz współczynnika załamania dla powietrza (1.0) oraz szkła. Dalej korzystamy z funkcji `refract()`, która oblicza odpowiedni wektor po ugięciu na podstawie wektora wejściowego, wektora normalnego oraz współczynnika refrakcji. Następnie przeliczamy uzyskany wektor do współrzędnych globalnych, aby na końcu uzyskać kolor próbki z tekstuury otoczenia (skybox).

W wyniku otrzymamy rezultat jak na rys. 37.9. Zaimplementowaliśmy bardzo prosty przezroczysty obiekt.



Rys. 37.9. Refrakcja na przykładzie prostej płaszczyzny.

Połączenie tych dwóch zjawisk świetlnych, tzn. odbicia oraz refrakcji, służy przede wszystkim do uzyskania realnie wyglądającej wody. W jednym z kolejnych rozdziałów przyjrzymy się tym dwóm zjawiskom jeszcze bliżej w bardziej ciekawym zastosowaniu.

Tips & Tricks:

- Jeśli odbity obraz jest do góry nogami, należy sprawdzić przede wszystkim wektor kierunku pobierający kolor z mapy otoczenia. Może się okazać, że należy odwrócić znak współrzędnej Y oraz Z (najczęściej). Należy również upewnić się, że znak wektora normalnego oraz wektora wychodzącego z kamery są poprawne. Normalizacja wektorów jest również wymagana.
- Odbicie wygląda dużo lepiej jeśli model posiada, tzw. gładkie wektory normalne (smooth normals). Jeśli takich nie posiada, możemy zaimportować go za pomocą programu Blender, zmienić Flat normals na Smooth normals, a następnie ponownie go eksportować.
- Jeśli refrakcja nie wygląda zbyt dobrze, można użyć tekstur skybox o większej rozdzielczości.

Kod źródłowy z tego rozdziału jest w katalogu: [26_Odbicie_refrakcja](#)

38. Renderowanie tekstu na podstawie bitmap

Wyświetlanie tekstu podczas renderowania grafiki trójwymiarowej jest najczęściej czymś pożądanym. Podczas tworzenia gry komputerowej na pewno zajdzie potrzeba wyświetlenia jakiegoś tekstu na ekranie. Mogą to być na przykład komunikaty skierowane do gracza oraz przerożne liczniki. OpenGL nie posiada jednak zaimplementowanej możliwości renderowania tekstu. Musimy sami sobie zaimplementować taką możliwość od podstaw i tym właśnie zajmiemy się w tym rozdziale.

Mogliśmy wyróżnić tak jakby „dwa rodzaje tekstu”, z którym będziemy mieli do czynienia. Pierwszym jest **tekst stały**, który nie zmienia się poprzez cały czas jego wyświetlania. Są to na przykład elementy interfejsu użytkownika. Taki tekst wyświetlić na ekranie jest bardzo prosto. Wystarczy uruchomić program graficzny, utworzyć w nim jakiś napis na przezroczystym tle oraz zapisać jako plik graficzny. Następnie utworzyć z kilku wierzchołków prostokątną powierzchnię, na którą zostanie naniesiona utworzona tekstura z tekstem (napisem) (rys. 38.1). Na samym końcu renderowania rysujemy taki prostokąt (bo wtedy nie zostanie już przykryty przez inne obiekty), włączamy przezroczystość i napis mamy gotowy.



Rys. 38.1. Najprostsza metoda renderowania tekstu - wyświetlenie tekstu na powierzchni utworzonej przez kilka wierzchołków.

Drugim rodzajem tekstu jest oczywiście **tekst zmienny** i to on przysparza najwięcej problemów. Do tego rodzaju tekstu zaliczamy na przykład zmienne komunikaty (cele misji w grze), liczniki czasu lub amunicji albo aktualna liczba punktów. Nie możemy przecież na bieżąco generować nowych tekstur, aby wyświetlić je na ekranie. Problem ten trzeba rozwiązać inaczej.

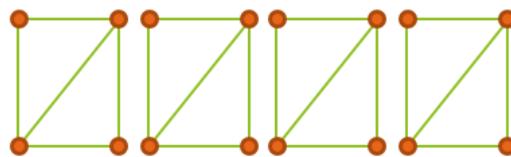
Pisząc coś na komputerze używana jest zawsze jakaś czcionka. To ona definiuje wygląd poszczególnych liter. Jak wspomniałem, OpenGL nie posiada możliwości wyświetlania (renderowania) tekstu, a co za tym idzie, używania czcionek. W zamian możemy skorzystać z tekstuury. Wystarczy, że stworzymy tekstuwę, która jest podzielona **na kwadraty o znanym rozmiarze**. W każdym kwadracie umieszczamy (lub rysujemy) kolejne litery oraz znaki, z których będziemy następnie chcieli skorzystać (rys. 38.2). Uzyskujemy w ten sposób czcionkę zapisaną w formacie graficznym. Taki plik graficzny z zapisanym wyglądem liter (czcionką) nazywamy zazwyczaj **font atlas**.

Pomysł wyświetlania zmiennych tekstów opiera się na pomyśle wyświetlania stałego tekstu opisanego powyżej. Różnica jest jedna. Dla tekstu stałego tworzymy z wierzchołków **jedną powierzchnię dla całego tekstu**, a dla tekstu zmiennego tworzymy **jedną odrębną powierzchnię dla każdej wyświetlanej litery** (rys. 38.3). Rozmiar generowanej

powierzchni dla pojedynczej litery ma oczywiście wpływ na ostateczną wielkość litery na ekranie.

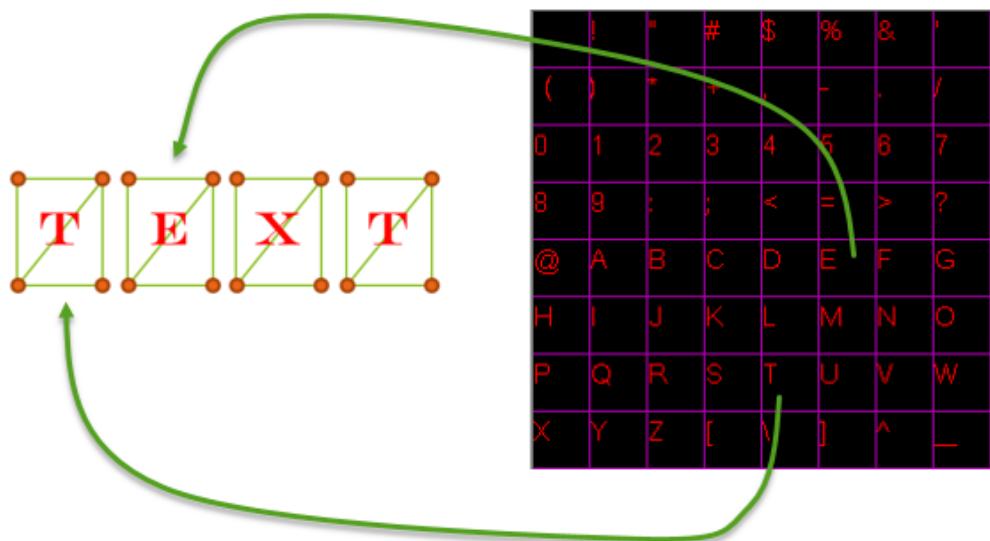
	l	"	#	\$	%	&	'
()	*	+	,	-	.	/
0	1	2	3	4	5	6	7
8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	[\]	^	_

Rys. 38.2. Przykładowy plik graficzny z zestawem liter oraz znaków (font atlas).



Rys. 38.3. Dla każdej litery tworzony jest osobny zestaw wierzchołków.

Posiadając teksturę z zapisanymi literami (font atlas) oraz informację o rozmiarze pojedynczego kwadratu, w którym znajduje się litera (znak), jesteśmy w stanie obliczyć współrzędne tekstury dla każdej z liter (znaków). Możemy nazwać to mapowaniem współrzędnych tekstury. Odpowiednio zmapowane współrzędne tekstury są następnie rysowane na poszczególnych przygotowanych wcześniej powierzchniach (rys. 38.4).



Rys. 38.4. Mapowanie współrzędnych tekstury font atlas na poszczególne powierzchnie.

Ta technika renderowania tekstów ma swoje wady i zalety. Największą jej zaletą jest to, że **możemy narysować własne artystyczne litery oraz znaki**. Font atlas nie musi przecież

zawierać nudnych liter napisanych czcionką Arial, itp. Wystarczy uruchomić program graficzny typu Gimp, podzielić płótno na kwadraty o stałym rozmiarze, a następnie w każdym z nich narysować swoją własną interpretację poszczególnego znaku. Wadą tej metody renderowania tekstu jest to, że używamy pliku graficznego, który nie jest wektorowy. Skalowanie liter nie będzie po prostu ładnie wyglądało. Jeśli w font atlas używamy liter, które mają X pikseli, powinniśmy renderować je tak, aby miały na ekranie także X pikseli. Druga wada związana jest z tym, że litery umieszczamy w kwadratach o stałym rozmiarze. Na przykład, litera W jest dość szeroka i zajmie prawie cały kwadrat na szerokość, ale już litera I jest wąska i zajmie tylko niewielką szerokość kwadratu. W wyniku otrzymamy nierówne odstępy pomiędzy literami i może to wyglądać dziwnie. Można oczywiście dla każdej litery wprowadzić także indywidualny współczynnik korekcji (przesunięcia w którąś stronę), ale to już wymaga od nas dodatkowej pracy. W związku z tymi wadami powstały lepsze metody renderowania tekstu, ale w tym rozdziale skupiamy się tylko na tej podstawowej.

Renderowanie tekstu, tak samo jak każdego innego obiektu, wymaga programu shadera. Kod vertex shader oraz fragment shader dla renderowania tekstu nie jest trudny, nie wymaga on większego tłumaczenia. Vertex shader przyjmuje tylko współrzędne wierzchołków oraz współrzędne tekstury. Oblicza współrzędne wierzchołków na ekranie, a współrzędne tekstury przekazuje do fragment shader. Fragment shader pobiera próbki koloru z tekstury na podstawie podanych współrzędnych i koloruje nią aktualny fragment ekranu.

Vertex shader:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;

out vec2 texture_coordinates;

void main()
{
    texture_coordinates = vt;
    gl_Position = vec4(position, 1.0);
}
```

Fragment shader:

```
#version 330
in vec2 texture_coordinates;

out vec4 frag colour;

uniform sampler2D font_texture;

void main()
{
    vec4 texel = texture(font_texture, texture_coordinates);
    frag colour = texel;
}
```

Pozostały kod, który musimy napisać najlepiej będzie zorganizować w klasę, aby lepiej powiązać dane z funkcjami. Będzie to klasa, która umożliwi wyświetlenie tekstu w dowolnym miejscu na ekranie korzystając z dostarczonego pliku graficznego z zestawem znaków (font atlas).

Zacznijmy od definicji tej klasy wraz z konstruktorem i destruktorem oraz zmiennymi składowymi.

```
class FontAtlasRenderer
{
public:
    FontAtlasRenderer(std::string font_file_name, int columns, int rows)
    {
        glGenVertexArrays(1, &handle );
        glGenBuffers(1, &vertices_vbo_);
        glGenBuffers(1, &texture_coords_vbo_);

        Texture font texture;
        loadTexture(font file name, font texture);
        loadTexture2D(font texture handle , font texture);

        atlas_height_ = font_texture.height;
        atlas_width_ = font_texture.width;

        atlas_rows_ = rows;
        atlas_columns_ = columns;

        character_width_ = atlas_width_ / atlas_columns_;
        character height_ = atlas height_ / atlas rows ;
    }

    ~FontAtlasRenderer()
    {
        glDeleteBuffers(1, &vertices_vbo_);
        glDeleteBuffers(1, &texture coords vbo );

        glDeleteVertexArrays(1, &handle_);
    }

protected:
    GLuint font texture handle {0};
    GLuint handle {0};
    GLuint texture_coords_vbo_{0};
    GLuint vertices_vbo_{0};
    int spacing horizontal {3};
    int spacing vertical {5};
    std::map<char, int> character offsets ;
    std::string characters ;
    unsigned int atlas_columns_{0};
    unsigned int atlas_rows_{0};
    unsigned int atlas width {0};
    unsigned int atlas height {0};
    unsigned int character height {0};
    unsigned int character_width_{0};
    unsigned int viewport_height_{320};
    unsigned int viewport width {240};

};
```

Zacznijmy od konstruktora. Przyjmuje on trzy parametry. Pierwszym jest ścieżka do pliku graficznego z zdefiniowaną czcionką (font atlas). Drugi oraz trzeci parametr służą do określenia ile kolumn oraz wierszy znaków występuje w podanym pliku graficznym. Te dwie wartości umożliwiają później poprawne obliczenie współrzędnych tekstury dla danego znaku. Wewnątrz konstruktora generowane są odpowiednie obiekty, czyli jeden obiekt VAO oraz dwa obiekty VBO. Jeden będzie przechowywał współrzędne wierzchołków, a drugi współrzędne tekstury. Dalej ładowana jest tekstura, czyli plik graficzny będący naszym font atlas. Uchwyty do tej tekstury zostają zapisane w zmiennej `font_texture_handle_`. Na podstawie informacji o wymiarach pliku graficznego w pikselach oraz o liczbie kolumn i wierszy jesteśmy w stanie obliczyć wymiar (szerokość oraz wysokość) pojedynczego znaku w pikselach. Wartości te trafiają do zmiennych `character_width_` oraz `character_height_`.

Destruktor jest bardzo prosty. Jego zadaniem jest tylko usunięcie niepotrzebnych już obiektów.

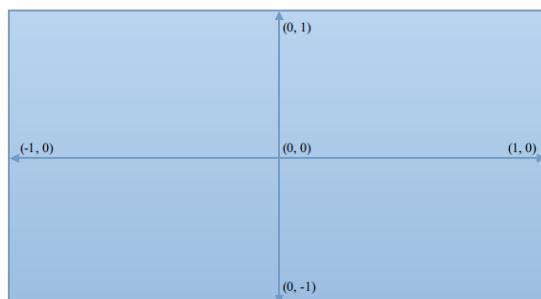
Będziemy potrzebowali także kilku funkcji pomocniczych.

```
void setViewportSize(int width, int height)
{
    viewport_width_ = std::abs(width);
    viewport_height_ = std::abs(height);
}

float calculateScreenCoordX(float x)
{
    return (x / static_cast<float>(viewport_width_) * 2 - 1);
}

float calculateScreenCoordY(float y)
{
    return (1 - y / static_cast<float>(viewport_height_) * 2);
}
```

Funkcja `setViewportSize()` służy do ustawiania rozmiaru okna w pikselach, w jakim odbywa się renderowanie. Rozmiar powierzchni renderowania jest potrzebny w celu przeliczenia współrzędnych ekranu z pikseli na przedział od -1.0 do 1.0, czyli na współrzędne wykorzystywane podczas rysowania. Dla przypomnienia proponuję spojrzeć na rys. 38.5. Funkcja `calculateScreenCoordX()` przelicza współrzędne X, a funkcja `calculateScreenCoordY()` współrzędne Y.



Rys. 38.5. Układ współrzędnych ekranu znajduje się w punkcie centralnym.

Możemy także dodać dwie funkcje, które posłużą do ustawiania odstępu poziomego oraz pionowego między kolejnymi znakami. Odstęp podawany jest w pikselach.

```
void setHorizontalSpacing(int spacing)
{
    spacing_horizontal_ = spacing;
}

void setVerticalSpacing(int spacing)
{
    spacing_vertical_ = spacing;
}
```

Niektóre znaki, np. litera y, powinny być czasami (zależy od czcionki) ułożone nieco niżej w porównaniu z innymi znakami. Można zatem dodać kolejną funkcję, która będzie umożliwiała wprowadzenie **korekcji położenia pionowego** poszczególnych znaków. Tutaj wartość przesunięcia także podawana jest w pikselach. W pierwszym parametrze podawany jest znak, dla którego chcemy dodać korekcję.

```
void setCharacterOffset(char character, int offset)
```

```
{
    character offsets [character] = offset;
}
```

Nasza klasa powinna także wiedzieć jakie znaki są dostępne w podanym font atlas. Jeśli spojrzysz jeszcze raz na rys. 38.2, zauważysz, że występujące znaki w tym font atlas ułożone są zgodnie z rosnącym kodem ASCII. Ale nie zawsze tak musi być. Czasem różnego rodzaju znaki specjalne, takie jak @ lub ~, nie są nam potrzebne. Jeśli potrzebujemy tylko małych i dużych liter oraz cyfr, to wtedy tworzymy font atlas zawierający tylko takie znaki. Skoro niezachowana jest ciągłość kodów ASCII program musi wiedzieć, gdzie ma szukać danego znaku w pliku graficznym. Aby dostarczyć klasie takiej wiedzy, należy przekazać jej ciąg znaków, który zawiera wszystkie znaki zdefiniowane w font atlas w takiej kolejności, w jakiej tam występują. Na przykład dla font atlas z rys. 38.2, byłby to ciąg znaków: „!\"#\$%&\'()*+,.-/0123 ...” itd. Zauważ, że na pierwszym miejscu jest znak pusty, czyli spacja. Funkcja, która umożliwia ustawienie dostępnego zestawu znaków wygląda następująco:

```
void setAvailableCharacters(std::string characters)
{
    characters = characters;
}
```

Wszystkie funkcje pomocnicze są już gotowe. Przyszedł czas na główną funkcję, którą nazwałem `renderText()`:

```
void renderText(std::string text, int x, int y, int scale)
{
    ...
}
```

W pierwszym parametrze przyjmuje ona oczywiście tekst, który ma zostać wyrenderowany. Drugi i trzeci parametr służą do określenia współrzędnych X oraz Y ekranu w pikselach, gdzie ma pojawić się górny lewy róg pierwszej litery. Ostatnim parametrem jest skala. Wyżej nie polecałem skalowania takich znaków pochodzących z tekstury, ale mimo to zdecydowałem się zaimplementować skalowanie w celach demonstracyjnych.

Wewnątrz funkcji, na samym jej początku, umieszczałyśmy definicje zmiennych pomocniczych. Wektor `vertices_buffer` będzie przechowywał współrzędne wierzchołków, a wektor `texture_coords_buffer` przechowywał będzie współrzędne tekstury. Zmienna `character_index` określa ile znaków w danej linii zostało już wygenerowanych, a zmienna `lines_index` aktualną liczbę linii tekstu.

```
std::vector<GLfloat> vertices_buffer;
std::vector<GLfloat> texture_coords_buffer;

int character_index{0};
int lines_index{0};
```

Dalej potrzebujemy pętli, która będzie przechodziła po wszystkich znakach podanego przez parametr tekstu (zmienna `text`):

```
for (const auto &character : text)
{
    if (character == '\n')
    {
        lines_index++;
    }
}
```

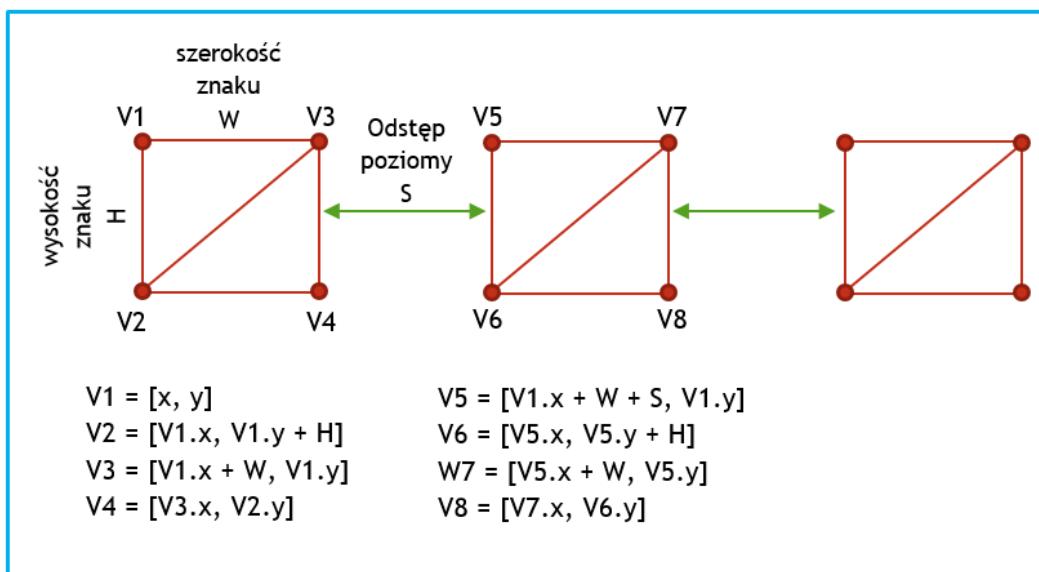
```

        character_index = 0;
        continue;
    }
    ...
}

```

Wewnątrz pętli sprawdzamy na początku, czy aktualnie przetwarzany znak, nie jest znakiem nowej linii. Jeśli tak, inkrementujemy licznik linii, zerujemy liczbę znaków w linii oraz przechodzimy do kolejnego znaku (`continue`).

Każda litera będzie tworzona przez cztery wierzchołki, na których będzie „umieszczana” tekstura. W pętli tej musimy je zatem wygenerować. Zasada obliczania kolejnych współrzędnych wierzchołków jest bardzo prosta i prezentuje ją rys. 38.6.



Rys. 38.6. Obliczanie współrzędnych wierzchołków dla znaków.

Programowo wygląda to następująco:

```

std::vector<glm::vec3> vertices(4);

vertices[0] = glm::vec3(
    x + character_index * (character_width_ + spacing_horizontal_) * scale,
    y + lines_index * (character_height_ + spacing_vertical_) * scale +
    static_cast<float>(character_offsets_[character]),
    0.0);

vertices[1] = glm::vec3(
    vertices[0].x,
    vertices[0].y + character_height_ * scale,
    0.0);

vertices[2] = glm::vec3(
    vertices[0].x + character_width_ * scale,
    vertices[0].y,
    0.0);

vertices[3] = glm::vec3(
    vertices[2].x,
    vertices[1].y,
    0.0);

for (auto &vertex : vertices)
{
    vertex.x = calculateScreenCoordX(vertex.x);
}

```

```

        vertex.y = calculateScreenCoordY(vertex.y);
    }

    for (auto &index : {0, 1, 2, 2, 1, 3})
    {
        vertices_buffer.push_back(vertices[index].x);
        vertices_buffer.push_back(vertices[index].y);
        vertices_buffer.push_back(vertices[index].z);
    }
}

```

Na początku tworzona jest czteroelementowa tablica (wektor) na wierzchołki. Następnie obliczane są współrzędne poszczególnych wierzchołków aktualnego znaku. W przypadku obliczania współrzędnej X uwzględniana jest wartość zmiennej `character_index`, a w przypadku współrzędnej Y `lines_index`. Dla współrzędnej Y dodatkowo uwzględniane jest przesunięcie (offset) dla aktualnie przetwarzanego znaku. Domyslnie wynosi ono zero.

Powyżej obliczone współrzędne wyrażone są w pikselach. Musimy przeliczyć je na współrzędne ekranu. Do tego celu przydadzą nam się funkcje pomocnicze `calculateScreenCoordX()` oraz `calculateScreenCoordY()`, które były już opisane wcześniej.

Utworzyliśmy cztery wierzchołki, ale teraz musimy utworzyć z nich dwa trójkąty, tak aby powstał prostokąt. Do tego celu służy ostatnia pętla, która pobiera kolejny indeks wierzchołka z listy oraz dodaje odpowiedni wierzchołek do bufora wierzchołków. Oczywiście nie musimy tak robić. Możemy wykorzystać tablicę indeksów, aby potem poinformować OpenGL podczas renderowania, które wierzchołki należy połączyć ze sobą. Nie chciałem już jednak wprowadzać kolejnej komplikacji. Nie dublujemy dużej ilości danych, dlatego możemy to pominąć.

Po obliczeniu współrzędnych wierzchołków dla aktualnego znaku należy obliczyć współrzędne tekstury dla tego znaku.

```

std::vector<glm::vec2> coords(4);
auto character_pos = characters_.find(character);
if (character_pos != std::string::npos)
{
    int row = character_pos / atlas_columns;
    int col = character_pos - row * atlas_columns;

    coords[0] = glm::vec2{
        col * character_width,
        (atlas_rows - row) * character_height};

    coords[1] = glm::vec2{
        coords[0].x,
        coords[0].y - character_height};

    coords[2] = glm::vec2{
        coords[0].x + character_height_,
        coords[0].y};

    coords[3] = glm::vec2{
        coords[2].x,
        coords[1].y};

    for (auto &coord : coords)
    {
        coord.x /= static_cast<float>(atlas_width);
        coord.y /= static_cast<float>(atlas_height);
    }
}

for (auto &index : {0, 1, 2, 2, 1, 3})

```

```

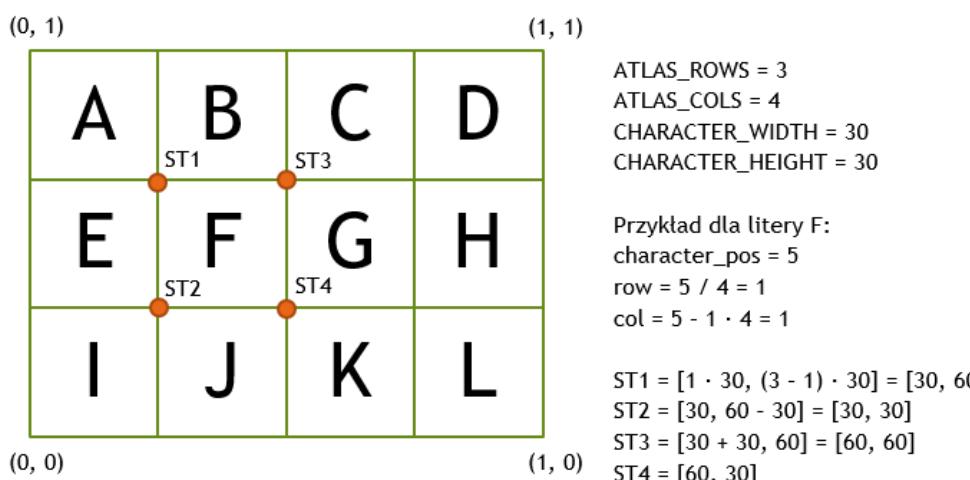
    {
        texture coords buffer.push back(coords[index].x);
        texture coords buffer.push back(coords[index].y);
    }

character_index++;

```

Pierw rezerujemy miejsce na cztery wektory dwuwymiarowe, w których będą przechowywane współrzędne tekstury.

Jak pamiętamy, w zmiennej `characters_` przechowywany jest dostępny zbiór znaków, które mogą zostać wyrenderowane (bo są zdefiniowane w font atlas). Musimy zatem przeszukać ten zbiór, aby sprawdzić, czy aktualny znak może zostać wyrenderowany. Jeśli aktualny znak zostanie znaleziony, obliczane jest jego położenie na font atlas, tzn. rząd i kolumna, w której się znajduje. Posiadając tą informację oraz informację o szerokości i wysokości litery w pikselach na font atlas, jesteśmy w stanie obliczyć współrzędne tekstury (w pikselach), które zawierają ten znak. Przypomnę, że układ współrzędnych tekstury zaczepiony jest w jej dolnym lewym narożniku. Między innymi z tego względu należy obliczyć różnicę (`atlas_rows_ - row`) (rys. 38.7).



Rys. 38.7. Przykład obliczania współrzędnych tekstury dla znaku (wartości w pikselach).

Występujące dalej dwie pętle mają zadanie podobne jak w przypadku współrzędnych wierzchołków. Pierwsza przelicza współrzędne z pikseli na przedział od 0.0 do 1.0 (współrzędne tekstury), a druga dodaje współrzędne tekstury w odpowiedniej kolejności do bufora.

Na zakończenie jednego przebiegu pętli, gdzie przetwarzany jest pojedynczy znak, należy jeszcze inkrementować wartość `character_index`, aby kolejne wierzchołki były obliczane poprawnie.

Wszystkie dane już są przygotowane. Można je wpisać do obiektu VAO.

```

glBindVertexArray(handle_);

glBindBuffer(GL_ARRAY_BUFFER, vertices vbo );
glBufferData(GL_ARRAY_BUFFER, vertices buffer.size() * sizeof(GLfloat),
             vertices_buffer.data(), GL_DYNAMIC_DRAW);

```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);  
glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo );  
glBufferData(GL_ARRAY_BUFFER, texture_coords_buffer.size() * sizeof(GLfloat),  
             texture_coords_buffer.data(), GL_DYNAMIC_DRAW);  
  
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, NULL);
```

Powyższy kod nie wymaga wyjaśnień. Jedynym ciekawszym elementem tutaj jest użycie `GL_DYNAMIC_DRAW` zamiast `GL_STATIC_DRAW`, które działały tak samo (ja nie zauważylem różnicy). Za pomocą `GL_DYNAMIC_DRAW` informujemy OpenGL, że dane będą się często zmieniać. W takim wypadku każde użycie funkcji `glBufferData()` powoduje usuwanie nieużywanej pamięci powodując (teoretycznie) wzrost wydajności. Chciałbym zaznaczyć, że jest to jedynie sugestia dla sterownika karty graficznej. Jak on to zinterpretuje i co z tym zrobi, nie do końca wiadomo.

Obiekty z danymi są już gotowe. Można przystąpić do wyrenderowania tekstu. Aby tekst nie został przykryty przez inne obiekty, które byłyby renderowane później, renderujemy go na samym końcu.

```
 glEnable(GL_BLEND);  
 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
  
 glBindTexture(GL_TEXTURE_2D, font.texture_handle );  
 glDisable(GL_DEPTH_TEST);  
 glEnableVertexAttribArray(0);  
 glEnableVertexAttribArray(1);  
 glDrawArrays(GL_TRIANGLES, 0, vertices_buffer.size() / 3);  
 glDisableVertexAttribArray(0);  
 glDisableVertexAttribArray(1);  
 glBindVertexArray(0);  
 glEnable(GL_DEPTH_TEST);
```

Jeśli nasz font atlas jest umieszczony na przezroczystym tle, możemy włączyć obsługę przezroczystości za pomocą funkcji `glEnable()`. Możemy to także zrobić wewnątrz fragment shader. Wystarczy sprawdzać wartość kanału alfa i odrzucać fragmenty za pomocą instrukcji `discard`, jak było to pokazane w poprzednich rozdziałach.

Dobrym pomysłem jest także wyłączenie testowania głębi na czas renderowania tekstu. Jesteśmy wtedy pewni, że nasz napis nie zostanie przypadkowo przykryty przez jakiś inny obiekt na scenie. Oczywiście na koniec musimy ponownie włączyć testowanie głębi.

Nasza klasa zajmująca się renderowaniem tekstu jest już gotowa. Pokażę teraz jak z niej skorzystać.

```
FontAtlasRenderer font_renderer("font_atlas.png", 16, 6);  
font_renderer.setAvailableCharacters(" !\"#$%&'\")*+,-.0123456789;:<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ"  
                                    "UVWXYZ[\\"\\]^ \\'abcdefghijklmnopqrstuvwxyz{|}~");  
font_renderer.setHorizontalSpacing(-15);  
font_renderer.setCharacterOffset('p', 2);
```

Oczywiście potrzebujemy obiektu naszej klasy, więc na samym początku go tworzymy. Jako parametry konstruktora podajemy nazwę pliku z font atlas oraz liczbę kolumn oraz wierszy. Następnie ustawiamy zestaw dostępnych znaków. Możemy też skonfigurować odstępy między znakami oraz indywidualne współczynniki korekcji dla poszczególnych znaków.

Wewnątrz głównej pętli zajmującej się renderowaniem możemy skorzystać z naszego obiektu klasy w celu wyrenderowania jakiś napisów. Należy jeszcze pamiętać o ustawieniu rozmiaru okna za pomocą funkcji `setViewportSize()`, aby współrzędne były przeliczane poprawnie.

```
font renderer.setViewportSize(window width, window height);
font renderer.renderText("Hello 3D OpenGL World!", 50, 100, 1);
font_renderer.renderText("FPS: " + std::to_string(fps), 100, 300, 1);
font_renderer.renderText("Multiline text\nHello\nWorld", 300, 400, 1);
```

Ostatecznie otrzymamy rezultat taki jak na rys. 38.8.

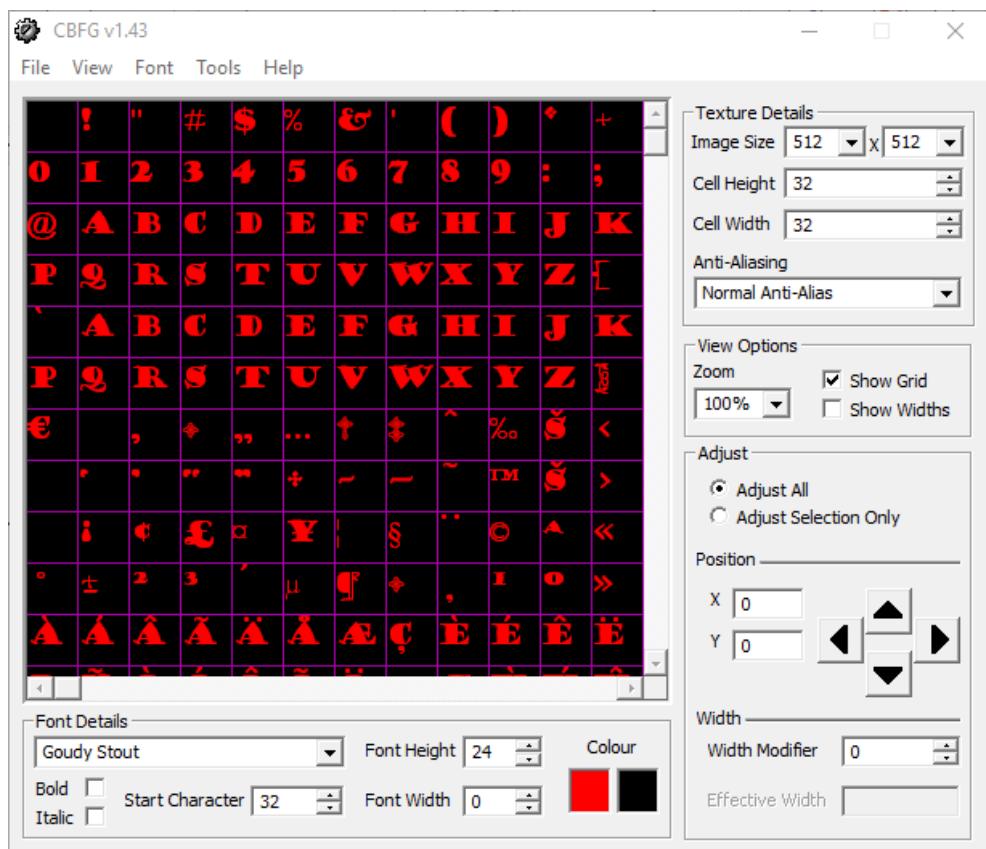


Rys. 38.8. Końcowy rezultat renderowania tekstu.

Tips & Tricks:

- Font atlas jest teksturą tak jak każda inna tekstura. Dla tej tekstury można także włączyć filtrowanie, aby w pewnym stopniu poprawić wygląd renderowanego tekstu.
- Internet jest dobrym źródłem gotowych plików graficznych z font atlas. Możemy jednak także stworzyć swoje. Jeśli zależy nam na artystycznych czcionkach, narysujmy je sami w Gimp'ie. Do generowania font atlas'ów na podstawie dostępnych w systemie czcionek służy narzędzie **Bitmap Font Generator**, które można pobrać ze strony <http://www.codehead.co.uk/cbfg/>. Po uruchomieniu programu (rys. 38.9) wybieramy czcionkę, jej rozmiar oraz kolor. Następnie wybieramy początkowy kod ASCII oraz rozmiar generowanej bitmapy. Im bitmapa jest większa, tym pomieści więcej znaków. Możemy także korygować indywidualne litery, przesuwać je w dowolne strony wewnątrz kwadratów. Gdy jesteśmy już

zadowoleni z tego jak font atlas wygląda, wchodzimy w menu **File -> Export -> Bitmap** i zapisujemy font atlas w postaci pliku graficznego. Wyeksportowany plik nie ma niestety przezroczystego tła. Musimy usunąć je ręcznie w dowolnym programie graficznym. Ja polecam tutaj także program Gimp. Wystarczy, że załadujemy do niego nasz font atlas, a następnie wejdziemy w menu **Kolory -> Zmiana koloru na alfę** i w opcjach narzędzia ustawimy kolor tła, np. czarny. Jeśli chcemy, możemy także dodatkowo usunąć w Gimp'ie niepotrzebne nam znaki z font atlas. Teraz już wystarczy zapisać wyretuszowany plik i font atlas jest gotowy do użycia.



Rys. 38.9. Narzędzie Bitmap Font Generator.

Kod źródłowy z tego rozdziału jest w katalogu: [27_Tekst_1](#)

39. Renderowanie tekstu z FreeType

Największą wadą renderowania tekstu na podstawie bitmap jest to, że poszczególne litery nie są w pełni skalowalne (tekstura nie jest przecież wektorowa). Oczywiście można je skalować, ale tekst złożony z takich liter, nie będzie wyglądał ładnie. Najlepiej renderować jest litery o takim samym rozmiarze w pikselach, w jakim są one zdefiniowane w pliku graficznym font atlas. Jeśli stwierdzimy, że chcielibyśmy jednak większe litery, bo nasz napis jest nieczytelny, musimy wygenerować nowy font atlas. Potem może się jeszcze okazać, że w innym miejscu potrzebujemy czcionki o takim samym kroju, ale jeszcze o większych literach i innym kolorze. Trzeba będzie zatem wygenerować kolejny font atlas. I tak dalej, i tak dalej. Możemy w dużym stopniu uprościć sobie sprawę za pomocą biblioteki **FreeType**, która jest tematem niniejszego tekstu.

Biblioteka **FreeType** jest darmową biblioteką odpowiedzialną za generowanie bitmap z poszczególnymi znakami żądanej czcionki. To znaczy, wybieramy czcionkę (np. Arial lub Times New Roman) oraz jej krój (np. kursywa), ustawiamy wielkość czcionki i już wszystko mamy przygotowane do dalszej pracy. Na podstawie tych parametrów, biblioteka jest w stanie wygenerować bitmapę z dowolnym znakiem (na jedną bitmapę przypada tylko jeden znak, np. litera 'W'). Ważną zaletą tej biblioteki jest to, że potrafi ona generować bitmapy z znakami nie tylko w kodzie ASCII, ale także i Unicode. Dzięki temu, możemy używać polskich znaków z ogonkami (ą, ę, itp.) lub na przykład cyrylicę. Ponadto, możemy sterować rozmiarem generowanych bitmap z znakiem. Font atlas staje się już zbędny. Każdy znak będzie generowany na bieżąco przez CPU wtedy, gdy będzie potrzebny. FreeType jest ogólnie zaawansowaną biblioteką. Posiada bardzo dużo opcji generowania bitmap z znakami graficznymi. Na razie zajmiemy się tylko tymi podstawowymi funkcjami, czyli najbardziej potrzebnymi.

Jako, że jest to biblioteka zewnętrzna, musimy zainstalować ją w systemie.

1. Pobieramy kod źródłowy biblioteki ze [strony](#) (w chwili pisania najnowsza wersja biblioteki to 2.6.2) oraz rozpakowujemy go do jakiegoś katalogu, np. `freetype`.
2. Wewnątrz katalogu biblioteki, zakładamy katalog o nazwie, np. `build`, oraz przechodzimy do niego poprzez wiersz poleceń.

```
>> cd freetype  
>> mkdir build  
>> cd build
```

3. Zarówno dla systemu Windows oraz Linux procedura jest na razie taka sama. Wewnątrz katalogu `bulid` wydajemy następujące polecenie (jeśli nie mamy `cmake`, musimy go zainstalować ze [strony](#)):

```
>> cmake ..
```

Uwaga: użytkownicy systemu Windows nie powinni mieć raczej problemów z użyciem tego polecenia. Jeśli jednak wystąpią jakieś problemy, polecam użyć narzędzia graficznego **Cmake GUI** lub na przykład **QtCreator**, który odczytuje pliki **CMakeLists.txt**.

4. Gdy proces się zakończy, użytkownicy systemu Linux mogą po prostu wydać polecenia:

```
>> make  
>> sudo make install
```

Natomiast użytkownicy systemu Windows w katalogu **build** znajdą najprawdopodobniej pliki posiadanego IDE niezbędne do zbudowania biblioteki, np. plik projektu **freetype.sln** dla **Visual Studio**.

5. Należy jeszcze pamiętać o ustawieniu w naszym IDE ścieżki do plików nagłówkowych FreeType (jest to katalog **include** wewnątrz katalogu FreeType) oraz ścieżki do pliku z zbudowaną biblioteką (dla Windows jest to ścieżka do pliku **freetype.lib** w katalogu **build**). Niestety jest za dużo kombinacji konfiguracji systemu oraz różnych IDE, więc niemożliwe jest opisanie ich wszystkich. Liczę na doświadczenie czytelnika w kwestii instalacji tej biblioteki.

Po skonfigurowaniu środowiska możemy przystąpić do pisania klasy, która będzie zajmowała się renderowaniem tekstu z wykorzystaniem biblioteki FreeType. W pierwszej kolejności należy dodać plik nagłówkowy biblioteki FreeType:

```
#include <ft2build.h>  
#include FT_FREETYPE_H
```

Plik nagłówkowy **ft2build.h** zawiera zdefiniowane różnego rodzaju makra, które są wykorzystywane w celu dołączenia odpowiednich plików nagłówkowych. Jednym z nich jest użyte tutaj makro **FT_FREETYPE_H**. Dołącza ono między innymi pozostałe wymagane pliki nagłówkowe oraz „pilnuje”, aby nie zaszedł konflikt z przestarzałą wersją FreeType.

Klasa, którą będziemy pisać, będzie nazywać się **FreeTypeFontRenderer**. Jej na razie puste wnętrze będziemy stopniowo uzupełniać.

```
class FreeTypeFontRenderer  
{  
public:  
    FreeTypeFontRenderer(std::string font_name, unsigned int size)  
    {  
        // ...  
    }  
  
protected:  
};
```

Biblioteka FreeType potrzebuje do działania dwóch obiektów, które należy zdefiniować:

```
protected:  
    FT_Face font_face_;  
    FT_Library ft_library_;
```

Obiekt typu **FT_Library** jest to jakby uchwyt do instancji biblioteki FreeType, a obiekt **FT_Face** służy do przechowywania wybranej czcionki oraz jej kroku. Na podstawie tego obiektu będą generowane bitmapy z znakami, o które poprosimy.

Konstruktor klasy przyjmuje dwa parametry. Pierwszym jest ścieżka do pliku z czcionką, np. /usr/share/fonts/truetype/msttcorefonts/arial.ttf na Linux, a drugim parametrem jest oczywiście żądaný rozmiar czcionki.

Pierwszą czynnością, jaką musimy zrobić przed użyciem jakichkolwiek funkcji z biblioteki FreeType jest jej inicjalizacja. Do tego celu używamy funkcji `FT_Init_FreeType()`. Jako parametr przyjmuje ona uchwyt do instancji biblioteki, który ma zainicjalizować. W przypadku błędu, funkcja zwraca kod błędu różny od zera.

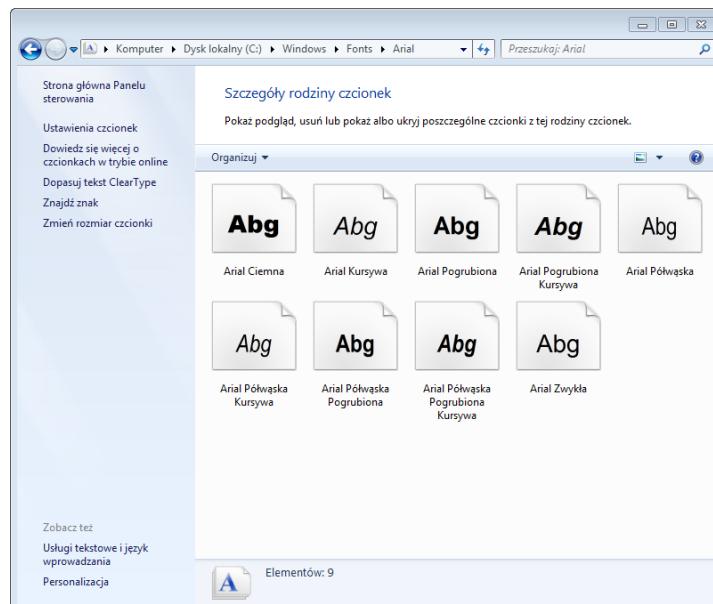
```
if (FT_Init_FreeType(&ft_library))
{
    std::cout << "FreeType initialization error." << std::endl;
    return;
}
```

Kolejnym krokiem jest załadowanie podanej czcionki.

```
if (FT_New_Face(ft_library, font_name.c_str(), 0, &font_face))
{
    std::cout << "Font loading error. Does font exist?" << std::endl;
    return;
}
```

Za ładowanie czcionki odpowiedzialna jest funkcja `FT_New_Face()`. Każda czcionka może mieć kilka krojów (rys. 39.1), np. pogrubienie, kursywa. Żądaný numer kroju podajemy w trzecim parametrze tej funkcji. Kruk o numerze 0 ma każda czcionka. Liczbę dostępnych krojów czcionki można sprawdzić następująco (oczywiście wtedy, gdy czcionka będzie już załadowana):

```
std::cout << font_face->num_faces << std::endl;
```



Rys. 39.1. Lista krojów czcionki Arial w systemie Windows.

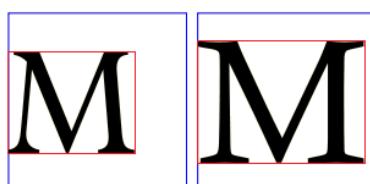
W pierwszym parametrze funkcji `FT_New_Face()` podajemy uchwyt do biblioteki FreeType, a w drugim ścieżkę do czcionki, która ma zostać załadowana. Ostatnim parametrem jest obiekt czcionki, gdzie podana czcionka zostanie załadowana.

Gdy biblioteka jest już zainicjalizowana oraz czcionka jest załadowana, należy ustawić żądanego rozmiar czcionki. Można do tego celu wykorzystać funkcję `FT_Set_Pixel_Sizes()`. Jest jeszcze inna funkcja do tego celu, ale ta jest prostsza w użyciu. Muszę zaznaczyć, że nie jest to rozmiar litery lub znaku wyrażony w pikselach. To, co tutaj ustawiamy, **jest to jakby rozmiar prostokąta, do którego wpisywana jest litera**, tzw. EM. Jest to jednostka stosowana w typografii. Nazwa tej jednostki wzięła się od wielkiej litery M, która jest najszersza i na jej podstawie tworzone były przez typografów formatki do projektowania liter danej czcionki (rys. 39.2).



Rys. 39.2. Dwie różne czcionki wpisane do em o takim samym rozmiarze.

Jeszcze jest jedna ważna kwestia do wyjaśnienia, która może być myląca. Biblioteka FreeType służy do generowania bitmapy z żądanym znakiem. Rozmiar wygenerowanej bitmapy nie będzie taki sam (najprawdopodobniej), jak rozmiar ustawiony przez funkcję `FT_Set_Pixel_Sizes()`. Funkcja ta ustawia tylko **maksymalny rozmiar litery**, a więc **maksymalny rozmiar bitmapy**. Na rys. 39.3 powinno być to dobrze widoczne.



Rys. 39.3. Dwie różne czcionki. Niebieska linia wyznacza maksymalny rozmiar litery, a czerwona rozmiar wygenerowanej bitmapy. Ostateczny rozmiar bitmapy zależy więc od użytej czcionki.

Tak jeszcze jako ciekawostka, jeśli mamy czcionkę o rozmiarze 12 punktów, to 1 em wynosi wtedy 12 punktów. Jeśli czcionka ma 16 punktów, to 1 em jest równe 16 punktów, itd. Możemy na przykład powiedzieć, że spacja jest równa 1.5 em, to wtedy dla czcionki o rozmiarze 12 punktów, spacja będzie miała 18 punktów.

Użycie funkcji `FT_Set_Pixel_Sizes()` jest następujące:

```
FT_Set_Pixel_Sizes(font_face_, 0, size);
```

Pierwszym parametrem jest obiekt czcionki, który chcemy skonfigurować. W drugim oraz trzecim parametrze podajemy kolejno szerokość oraz wysokość czcionki. Jeśli w drugim parametrze wpiszemy 0, oznacza to, że szerokość ma być taka sama jak wysokość.

Do dalszej rozbudowy klasy będziemy potrzebowali dodatkowych zmiennych składowych.

```
protected:
    GLuint font_texture_handle_{0};
    GLuint handle_{0};
    GLuint texture_coords_vbo_{0};
    GLuint vertices_vbo_{0};
    unsigned int viewport_height_{800};
    unsigned int viewport_width_{600};
```

Zmienna `font_texture_handle_` będzie uchwytem do tekstury z bitmapą kolejnej renderowanej litery (znaku). Potrzebujemy też kilku zmiennych na obiekty VAO oraz VBO i są to zmienne `handle_`, `vertices_vbo_` oraz `texture_coords_vbo_`. Aby możliwe było prawidłowe przeliczanie pikseli na współrzędne ekranu, potrzebny jest także jego rozmiar. Przechowywany jest on w dwóch zmiennych `viewport_height_` oraz `viewport_width_`.

Dalej wewnątrz konstruktora musimy wygenerować nową teksturę OpenGL oraz włączyć jej filtrowanie. Tekstura ta będzie przechowywała generowane przez FreeType na bieżąco bitmapy z znakami.

```
glGenTextures(1, &font_texture_handle_);

 glBindTexture(GL_TEXTURE_2D, font_texture_handle_);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 glBindTexture(GL_TEXTURE_2D, 0);
```

Ostatnią rzeczą, którą należy wykonać jeszcze w konstruktorze jest stworzenie obiektów VBO oraz VAO.

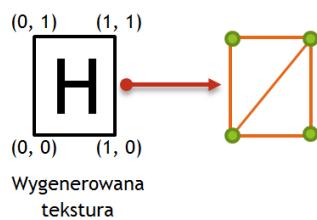
```
glGenVertexArrays(1, &handle );

 glGenBuffers(1, &vertices_vbo_);
 glGenBuffers(1, &texture_coords_vbo_);

 float texture_coords[] = {
    0, 0,
    0, 1,
    1, 0,
    1, 0,
    0, 1,
    1, 1
};

 glBindVertexArray(handle);
 glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindBuffer(GL_ARRAY_BUFFER, texture_coords_vbo );
 glBufferData(GL_ARRAY_BUFFER, sizeof(texture_coords), texture_coords, GL_STATIC_DRAW);
 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindVertexArray(0);
```

Na początku generujemy obiekt VAO oraz dwa obiekty VBO. Jeden będzie przechowywał współrzędne wierzchołków, a drugi współrzędne tekstury. Współrzędne tekstury będą takie same dla każdego renderowanego znaku. Tym razem nie mamy font atlas, który musimy odpowiednio „przeszukać” oraz znaleźć odpowiednie współrzędne tekstury. Mamy tylko jedną teksturę, która będzie renderowana na prostokątnym obszarze utworzonym przez wierzchołki (rys. 39.4).



Rys. 39.4. Jedna pojedyncza tekstuura trafia na jedną powierzchnię.

Jak widać, do obiektu VBO niewpisywane są na początku żadne dane. Ustawiany jest tylko wskaźnik na atrybut. Natomiast obiekt VBO przechowujący współrzędne tekstury jest od razu inicjalizowany danymi wraz z parametrem `GL_STATIC_DRAW`, czyli dane te, nie będą już zmieniane.

Głównym elementem tej klasy będzie funkcja renderująca podany tekst.

```
void renderText(std::wstring text, int x, int y)
{
    // ...
}
```

Elementem rzucającym się od razu w oczy jest tutaj to, że zamiast klasycznego typu `string`, przyjmuje ona parametr typu `wstring`. Typ ten obsługuje po prostu znaki zapisane w kodzie Unicode. Pozostałe dwa parametry określają współrzędne X oraz Y renderowanego tekstu na ekranie.

Wewnątrz funkcji pierw trzeba przeprowadzić kilka przygotowań. Przede wszystkim należy wybrać slot, do którego będzie wpisywana tekstura oraz należy ją zbindować. Zaraz będziemy do niej wpisywać dane. Musimy także przygotować bufor na generowane wierzchołki. Jest to `vertices_buffer`. Bindujemy także obiekt VAO. Ogólnie, bindujemy wszystko, co się da, przed wejściem do pętli, aby potem w kółko nie bindować tych samych obiektów.

```
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, font.texture.handle );
 glBindVertexArray(handle_);

int cursor_pos_x = x;
int cursor_pos_y = y;

std::vector<GLfloat> vertices_buffer;
```

Teraz przyszedł czas na najważniejszy moment. Musimy napisać pętlę, która będzie przechodziła po wszystkich znakach podanego tekstu, generowała odpowiednią bitmapę oraz następnie renderowała ją na ekranie.

```
for (const auto &character : text)
{
    // ...
}
```

Pierwszymi instrukcjami w pętli będzie generowanie bitmapy za pomocą biblioteki FreeType oraz następnie wpisanie jej do obiektu tekstury.

```
if (FT_Load_Char(font_face, character, FT_LOAD_RENDER))
    continue;

glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, font_face->glyph->bitmap.width,
            font_face->glyph->bitmap.rows, 0, GL_RED, GL_UNSIGNED_BYTE,
            font_face->glyph->bitmap.buffer);
```

Funkcja `FT_Load_Char()` wraz z parametrem `FT_LOAD_RENDER` zajmuje się właściwie wczytaniem znaku podanego w drugim parametrze, a następnie wygenerowaniu bitmapy z podanym znakiem. Jeśli proces ten się nie uda, pomijamy ten znak za pomocą instrukcji

continue. Wygenerowana bitmapa trafia do slotu (schowka) wewnątrz obiektu `FT_Face`. Slot ten potrafi przechowywać tylko jedną bitmapę w danej chwili, a więc powtórne wywołanie tej funkcji nadpisałoby wcześniejszą wygenerowaną bitmapę.

Za pomocą funkcji `glTexImage2D()` możemy wpisać uzyskaną bitmapę do obiektu tekstury. Dostęp do wygenerowanej tekstury mamy za pomocą obiektu `FT_Face` oraz wskaźnika o nazwie `glyph`. Zawartość bitmapy dostępna jest pod wskaźnikiem `bitmap.buffer`.

Trzeba nadmienić, że generowana bitmapa posiada tylko jeden kanał (czerwony). Dlatego też, w parametrach funkcji `glTexImage2D()` muszą zostać użyte wartości `GL_RED`. Bitmapą jest więc czarno-biała, gdzie tło jest czarne, a sam znak jest biały.

Teksturę mamy już gotową oraz załadowaną. Teraz czas na wygenerowanie wierzchołków aktualnego znaku.

```
std::vector<glm::vec3> vertices(4);

vertices[0] = glm::vec3{
    cursor_pos_x + font_face->glyph->bitmap_left,
    cursor_pos_y - font_face->glyph->bitmap_top,
    0.0};

vertices[1] = glm::vec3{
    vertices[0].x,
    vertices[0].y + font_face->glyph->bitmap.rows,
    0.0};

vertices[2] = glm::vec3{
    vertices[0].x + font_face->glyph->bitmap.width,
    vertices[0].y,
    0.0};

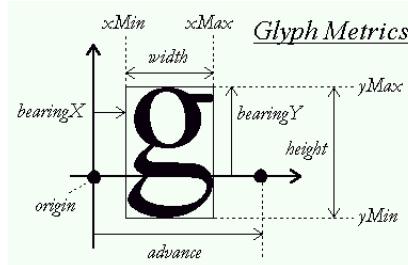
vertices[3] = glm::vec3{
    vertices[2].x,
    vertices[1].y,
    0.0};

for (auto &vertex : vertices)
{
    vertex.x = calculateScreenCoordX(vertex.x);
    vertex.y = calculateScreenCoordY(vertex.y);
}

for (auto &index : {0, 1, 2, 2, 1, 3})
{
    vertices_buffer.push_back(vertices[index].x);
    vertices_buffer.push_back(vertices[index].y);
    vertices_buffer.push_back(vertices[index].z);
}
```

Na początku rezerwujemy pamięć na cztery wierzchołki, które będą tworzyć jeden znak, a następnie je generujemy. Przewagą FreeType nad renderowaniem tekstu na podstawie font atlas jest to, że mamy dostęp do rozmiarów znaku oraz jego prawidłowego ustawienia. Proponuję spojrzeć na rys. 39.5. Nas w tej chwili interesują dwie wartości, jest to `bearingX` oraz `bearingY`. Informują nas one o tym, jak dana litera (znak) powinny być ustawione na wspólnej linii wraz z innymi znakami. Posiadając te dane jesteśmy w stanie poprawnie obliczyć ułożenie wierzchołków na ekranie. Po obliczeniu należy je przeliczyć na współrzędne ekranu. Służą do tego funkcje opisane w poprzednim rozdziale o

renderowaniu tekstu na podstawie bitmap. Na koniec dodajemy obliczone wierzchołki do bufora wierzchołków w odpowiedniej kolejności.



Rys. 39.5. Metryka znaków.

Gdy wszystkie wierzchołki są już obliczone oraz wpisane do bufora, można uzupełnić nimi obiekt VBO, a następnie wszystko wyrenderować.

```
glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo_);
glBufferData(GL_ARRAY_BUFFER, vertices buffer.size() * sizeof(GLfloat),
             vertices buffer.data(), GL_DYNAMIC_DRAW);
enableDepthTesting(false);
glDrawArrays(GL_TRIANGLES, 0, 6);
enableDepthTesting(true);
```

Oczywiście w pierwszej kolejności musimy aktywować obiekt VBO poprzez funkcję `glBindBuffer()`. Potem wpisujemy do niego obliczone wierzchołki za pomocą funkcji `glBufferData()` wraz z parametrem `GL_DYNAMIC_DRAW`, sugerującym, że dane będą wpisywane dynamicznie. Na czas renderowania tekstu najlepiej jest wyłączyć testowanie głębi oraz pamiętać o tym, żeby renderować go na końcu, aby nie został przykryty przez inne obiekty. W tym przypadku renderujemy 6 wierzchołków jednego prostokąta, a więc też taki parametr przekazujemy do funkcji `glDrawArrays()`.

Po wyrenderowaniu kolejnego znaku tekstu musimy odpowiednio przesunąć kursor. Na rys. 39.5 można zauważyć odległość nazwaną `advance`. To właśnie jej wartość interesuje nas najbardziej. Informuje ona po prostu o tym, jak należy przesunąć kursor po napisaniu danego znaku.

```
cursor_pos_x += font_face_->glyph->advance.x >> 6;
```

A czemu taka dziwna składnia z przesuwaniem bitowym? Można to zapisać inaczej:

```
cursor_pos_x += font_face_->glyph->advance.x / 64;
```

Chodzi po prostu o to, że wartość `advance` wyrażona jest nie w pikselach, a w 1/64 części piksela, czyli na przykład jest równa 64 razy 12 pikseli. Jeśli podzielimy ją przez 64 uzyskamy liczbę pikseli.

Na koniec czyścimy bufor na wierzchołki:

```
vertices_buffer.clear();
```

Jeśli chcielibyśmy jeszcze obsługiwać znak nowej linii, należałoby dodać na początku pętli takie instrukcje:

```

if (character == '\n')
{
    cursor pos x = x;
    cursor pos y += font face ->glyph->bitmap.rows + font face ->glyph->bitmap top;
    continue;
}

```

Klasa jest gotowa. Teraz możemy napisać shadery odpowiedzialne za renderowanie tekstu. Vertex shader nie wymaga żadnych wyjaśnień, jest bardzo prosty.

```

#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 vt;

out vec2 texture coordinates;

void main()
{
    texture_coordinates = vt;
    gl_Position = vec4(position, 1.0);
}

```

Fragment shader w porównaniu z renderowaniem tekstu na podstawie bitmap uległ małej zmianie.

```

#version 330

in vec2 texture_coordinates;
out vec4 frag_colour;

uniform sampler2D font texture;
uniform vec3 colour;

void main()
{
    float texel R = texture(font texture, texture coordinates).r;
    frag colour = vec4(1.0, 1.0, 1.0, texel R) * vec4(colour, 1.0);
}

```

Jak wspominałem nieco wyżej, generowana tekstura ma tylko jeden kanał. Jest nim kanał koloru czerwonego R. Nie ma sensu więc pobierać wszystkich składowych koloru teksta, skoro będą one równe 0 (teksel ma w tym przypadku tylko dwie możliwe kombinacje [0, 0, 0, 1] lub [1, 0, 0, 1]). Wystarczy, że pobierzemy sam kanał R z teksta.

Przypomnę jeszcze, że na generowanej bitmapie jest wygenerowany biały znak na czarnym tle. Możemy ten fakt wykorzystać do ustawienia kanału alfa. Jeśli wartość jest równa 0 (czarne tło), to alfa jest równa 0 (całkowicie przezroczyste). A jeśli natomiast wartość jest równa 1 (biały znak), to alfa w tym przypadku wynosi 1 (całkowicie widoczne).

Kolejną sztuczką, którą można tutaj zastosować jest to, że kolorujemy teksel na biało, a następnie przemnażamy przez nowy kolor, który będziemy podawać poprzez zmienną uniform. Dzięki temu możemy dowolnie ustalać kolor renderowanego tekstu.

Do pełni szczęścia pozostało jeszcze kilka kroków. Musimy przede wszystkim utworzyć obiekt naszej klasy.

```

FreeTypeFontRenderer ft font renderer("/usr/share/fonts/truetype/msttcorefonts/arial.ttf",
32);

```

W pierwszym parametrze podałem lokalizację czcionki Arial w systemie Linux. W systemie Windows czcionki znajdują się w katalogu C:\Windows\Fonts.

Domyślnie tekstury w OpenGL używają 4 bajtów na piksel (jeden bajt na jeden kanał). Nasza generowana tekstura jest jedno-kanałowa. Aby była ona prawidłowo renderowana należy wyłączyć domyślnie 4-bajtowe wyrównanie tekstur. Możemy zrobić to gdziekolwiek przed główną pętlą renderującą. Służy do tego celu funkcja `glPixelStorei()` wraz z parametrem `GL_UNPACK_ALIGNMENT`. Od tej pory wszystkie tekstury będą wyrównane do jednego bajta.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

W fragment shader ustawiamy kanał alfa, a więc nie możemy zapomnieć o włączeniu przezroczystości w OpenGL.

```
 glEnable(GL_BLEND);
 glEnable(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Teraz już możemy wyrenderować dowolny tekst. Pamiętać należy o aktywowaniu shadera odpowiedzialnego za renderowanie tekstu oraz za ustawienie koloru w zmiennej `uniform`.

```
activateShaderProgram(font_shader);
setUniform(texture_slot_font, 0);
setUniform(colour_font, glm::vec3(1.0, 1.0, 0.0));
ft font renderer.renderText(L"Hello World!\nNew Line", 100, 50);
setUniform(colour_font, glm::vec3(0.6, 0.2, 0.8));
ft_font_renderer.renderText(L"Zażółć gęślą jaźń ...", 200, 200);
setUniform(colour_font, glm::vec3(1.0, 0.1, 0.9));
ft font renderer.renderText(L"\x410\x411\x412\x413\x414", 300, 400);
setUniform(colour_font, glm::vec3(0.0, 1.0, 0.9));
ft_font_renderer.renderText(L"\x3B2\x436\x2122\x263A", 50, 500);
```

Zwracam uwagę na to, że korzystamy z `wstring`, a nie `string`, dlatego przed ciągiem znaków musimy wstawić przyrostek L.

Ciekawą rzeczą jest także to, w jaki sposób możemy renderować dowolne znaki poprzez podanie ich kodu Unicode. Wystarczy, że wewnątrz cudzysłowu umieścimy kod znaku poprzedzony ciągiem \x umożliwiającym wprowadzenie wartości szesnastkowej. Kod znaku Unicode możemy sprawdzić na przykład w systemie Windows w Tablicy znaków (rys. 36.6).



Rys. 39.6. Tablica znaków systemu Windows.

Rezultat działania tego systemu renderowania czcionek jest na rys. 36.7.



Rys. 39.7. Wynik działania programu.

W celu bliższego poznania biblioteki FreeType polecam zapoznać się z [tutorialem](#) umieszczonym na stronie biblioteki. Pozwoli on bardziej zrozumieć niektóre zagadnienia związane z renderowaniem czcionek.

Kod źródłowy z tego rozdziału jest w katalogu: **28_Tekst_2**

40. Interfejsy użytkownika - GUI

Do graficznego interfejsu użytkownika (**GUI** - **G**raphical **U**sers **I**nterface) można zaliczyć elementy (kontrolki) wyjściowe oraz wejściowe. Oczywiście, mogą także istnieć elementy, które należą do obu tych grup. Elementy wyjściowe służą do wyświetlania różnych informacji użytkownikowi aplikacji, np. prosty komunikat tekstowy, pasek postępu oraz ikonki symbolizujące pewne stany aplikacji. Natomiast elementami wejściowymi są te wszystkie elementy interfejsu użytkownika, które wymagają na użytkowniku wykonania pewnych akcji. Umożliwiają one przede wszystkim wprowadzenie do programu jakiś danych. Przykładem takich elementów mogą być przyciski oraz pola edycyjne.

Zaprojektowanie oraz oprogramowanie dobrze działającego GUI jest trudniejsze niż się wydaje. Musimy przede wszystkim pomyśleć jak mają wyglądać poszczególne elementy oraz jaką mają mieć funkcję. Już samo narysowanie kontrolek na ekranie może przynieść dużo problemów. Narysowanie prostych elementów GUI jest raczej łatwe, ale jeśli chcielibyśmy rysować bardziej skomplikowane elementy, to wtedy musimy na to poświęcić już nieco więcej czasu. Ponadto, konieczna jest także obsługa zdarzeń z klawiatury oraz myszy. Najpierw należy przechwycić zdarzenie, następnie rozpoznać w jakich warunkach zostało ono wywołane, a na końcu wywołać odpowiednią funkcję obsługi zdarzenia. Mimo wszystko, warto poświęcić czas na zaprojektowanie dobrego GUI. Źle zaprojektowane GUI może po prostu zniechęcać użytkownika naszej aplikacji do korzystania z niej.

GUI z reguły tworzone jest pod konkretną aplikację (np. jakąś grę komputerową). Oprócz podstawowych kontrolek jak przyciski, czy paski postępu, może zajść potrzeba stworzenia całkiem nowych, np. przewijany przycisk, na którym wyświetlany jest pasek postępu. W związku z tym, nie jestem w stanie przedstawić wszystkich możliwych rozwiązań. Rozdział ten został podzielony na kilka podrozdziałów, w których opisane są różne podstawowe kontrolki GUI. Jeśli stanie się dla Ciebie jasne, jak tworzy się podstawowe GUI, będziesz w stanie sam zaprojektować oraz zaprogramować własne kontrolki.

Chciałbym jeszcze zaznaczyć, że przedstawione przeze mnie tutaj metody tworzenia kontrolek są tylko przykładowe. Wszystko można napisać w całkowicie inny sposób i o wiele lepiej (!), ale chciałem przedstawić sposób w miarę najprostszy bez żadnych technik zaawansowanego programowania.

40.1. Pasek postępu

Pasek postępu jest jednym z podstawowych elementów wyjściowych, tzn. informuje użytkownika o pewnym stanie (wartości). Działanie takiego elementu jest bardzo proste. Wraz ze zmianą wartości reprezentowanej przez pasek postępu zmienia się długość lub stopień wypełnienia pewnego pod-elementu paska postępu. Symbolicznie można porównać taki pasek do szklanki z wodą. Poziom wody w szklance reprezentuje pewną wartość (stopień wypełnienia szklanki) od 0 do 100%.

Nasz pasek postępu będzie zbudowany z **trzech pod-elementów**: tła, właściwego paska postępu oraz obramowania. Tło oraz pasek postępu będą dwukolorowe (gradient),

obramowanie natomiast będzie jednokolorowe. Aby nie komplikować zbytnio rysowania, tekstury nie będą wykorzystywane. Posiadając już takie wstępne założenia co do projektu, możemy napisać shadery, które zajmą się narysowaniem paska postępu na ekranie.

Vertex shader:

```
#version 330
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 colour;

out vec3 vertex_colour;

void main()
{
    vertex colour = colour;
    gl_Position = vec4(position, 1.0);
}
```

Fragment shader:

```
#version 330
in vec3 vertex_colour;

out vec4 frag colour;

uniform vec3 border_colour;
uniform int alpha_factor;
uniform bool rendering_border;

void main()
{
    if (rendering_border)
        frag colour = vec4(border_colour, alpha_factor / 100.0);
    else
        frag colour = vec4(vertex colour, alpha factor / 100.0);
}
```

Myślę, że większe wytłumaczenie jest zbędne. Przypomnę tylko, że renderując GUI oraz tekst z **reguły nie uwzględniamy macierzy kamery** (widoku oraz projekcji).

Fragment shader przyjmuje trzy zmienne typu `uniform`:

- `border_colour` - kolor obramowania elementu,
- `alpha_factor` - nasz pasek postępu będzie mógł być częściowo przezroczysty oraz stopień widoczności (w procentach) będzie mógł być sterowany z wnętrza programu,
- `rendering_border` - zmienna typu `bool` - jeśli jest równa `true`, oznacza to, że aktualnie renderujemy jednokolorowe obramowanie, a nie dwukolorowe tło lub właściwy pasek postępu.

Każdy element GUI rysowany na ekranie ma swoje położenie, szerokość, wierzchołki tworzące element, itp. W związku z tym dobrym pomysłem jest napisanie klasy bazowej elementu GUI, na podstawie której będą następnie tworzone kolejne nowe elementy GUI. Zaoszczędzimy sobie tym trochę pracy oraz czasu.

Nasza klasa bazowa nazywa się po prostu `GUIElement` i wygląda następująco:

```
class GUIElement
{
public:
    GUIElement(int x, int y, int width, int height)
    {
```

```

        if (x < 0 || y < 0)
            throw std::string("Specified invalid GUI element's position.");

        if (height <= 0 || width <= 0)
            throw std::string("Specified invalid GUI element's size.");

        x_ = x;
        y_ = y;

        height_ = height;
        width_ = width;

        glGenVertexArrays(1, &handle);

        glGenBuffers(1, &colours_vbo);
        glGenBuffers(1, &vertices_vbo);
        glGenBuffers(1, &indices_vbo_);

        glBindVertexArray(handle);
        glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
        glBindBuffer(GL_ARRAY_BUFFER, colours_vbo_);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
        glEnableVertexAttribArray(0);
        glEnableVertexAttribArray(1);
        glBindVertexArray(0);

        border_vertices_.resize(4);
        border_vertices_[0] = glm::vec3(x_, y_, 0.0f);
        border_vertices_[1] = glm::vec3(border_vertices_[0].x, border_vertices_[0].y +
                                         height,
                                         0.0f);
        border_vertices_[2] = glm::vec3(border_vertices_[0].x + width_, border_vertices_[0].y,
                                         0.0f);
        border_vertices_[3] = glm::vec3(border_vertices_[2].x, border_vertices_[1].y, 0.0f);

        for (auto &vertex : border_vertices)
        {
            vertex.x = calculateScreenCoordX(vertex.x);
            vertex.y = calculateScreenCoordY(vertex.y);
        }
    }

    virtual ~GUIElement()
    {
        glDeleteBuffers(1, &colours_vbo_);
        glDeleteBuffers(1, &vertices_vbo_);
        glDeleteBuffers(1, &indices_vbo_);

        glDeleteVertexArrays(1, &handle_);
    }

    void setBackgroundColours(const glm::vec3 &colour1, const glm::vec3 &colour2)
    {
        background_colour_1_ = colour1;
        background_colour_2_ = colour2;
    }

    void setBorderColour(const glm::vec3 &colour)
    {
        border_colour_ = colour;
    }

    static void setViewportSize(int width, int height)
    {
        if (width > 0 && height > 0)
        {
            viewport_height_ = height;
            viewport_width_ = width;
        }
    }

    virtual void render() = 0;

protected:
    float calculateScreenCoordX(float x)
    {

```

```

        return (x / static_cast<float>(viewport_width_) * 2 - 1);
    }

    float calculateScreenCoordY(float y)
    {
        return (1 - y / static_cast<float>(viewport_height_) * 2);
    }

protected:
    static int viewport_height;
    static int viewport_width;

    glm::vec3 background_colour_1 {0.5f, 0.5f, 0.5f};
    glm::vec3 background_colour_2 {0.8f, 0.8f, 0.8f};
    glm::vec3 border_colour {0.0f, 0.0f, 0.0f};

    GLuint colours_vbo {0};
    GLuint gui_shader_{0};
    GLuint handle_{0};
    GLuint indices_vbo {0};
    GLuint vertices_vbo {0};
    int height_{0};
    int width_{0};
    int x_{0};
    int y_{0};
    std::vector<glm::vec3> border_vertices;
    const std::vector<GLuint> border_indices {0, 1, 3, 2};
    const std::vector<GLuint> fill_indices {0, 1, 2, 2, 1, 3};
};


```

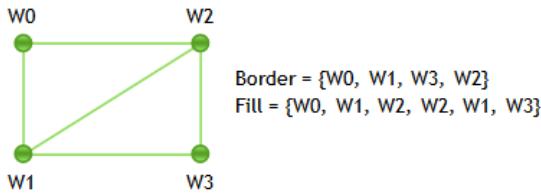
Element GUI musi posiadać swoje położenie oraz rozmiar (szerokość i wysokość) na ekranie. Dane te będą przechowywane w zmiennych `x_`, `y_`, `width_` oraz `height_`.

Według założeń początkowych, tło elementu będzie dwukolorowe, a obramowanie jednokolorowe. Do przechowywania kolorów tła wykorzystane zostaną zmienne `background_colour_1_` oraz `background_colour_2_`, a do przechowywania koloru obramowania zmienna `border_colour_`.

Nasze elementy GUI będą prostokątne, czyli obramowanie będzie zbudowane z czterech wierzchołków. Wierzchołki obramowania będą przechowywane w kontenerze `border_vertices_`. Dodatkowo potrzebna jest także jedna zmienna na obiekt VAO (zmienna `handle_`) oraz trzy zmienne na obiekty VBO. Pierwszy obiekt VBO przeznaczony będzie na współrzędne wierzchołków (zmienna `vertices_vbo_`), drugi na kolory wierzchołków (`colours_vbo_`) oraz trzeci na indeksy wierzchołków (informujące o tym, jak mają zostać połączone wierzchołki podczas renderowania) (zmienna `indices_vbo_`).

Do renderowania elementu GUI potrzebny jest także program shadera. Uchwyty do programu shadera dla konkretnego elementu będzie przechowywany w zmiennej `gui_shader_`.

Jak wspomniałem wyżej, element GUI (jego ramka) będzie zbudowany z czterech wierzchołków. W zależności od tego jak wyrenderujemy te wierzchołki, możemy narysować obramowanie lub kolorowe wypełnienie elementu GUI (tło). Do tego celu można przygotować sobie dwie tablice z odpowiednimi indeksami wierzchołków. Kontener (tablica) `border_indices_` przechowuje indeksy wierzchołków służące do wyrenderowania tylko obramowania, a kontener `fill_indices_` przechowuje indeksy wierzchołków do wyrenderowania wypełnienia (tła) elementu GUI zbudowanego z dwóch połączonych ze sobą trójkątów (rys. 40.1).



Rys. 40.1. Sposób ułożenia wierzchołków wraz z indeksami.

Aby możliwe było prawidłowe przeliczenie współrzędnych wyrażonych w pikselach na współrzędne ekranu potrzebujemy znać rozmiar ekranu (okna). Będzie on przechowywany w dwóch zmiennych statycznych (`viewport_height_` oraz `viewport_width_`), czyli wspólnych dla wszystkich obiektów klasy. Do ustawienia tych zmiennych służy statyczna funkcja składowa `setViewportSize()`.

Konstruktor klasy `GUIElement` sprawdza na początku, czy podane dane (współrzędne oraz rozmiar) są prawidłowe. W przeciwnym razie zgłasza wyjątek typu `std::string`. Następnie generowane są obiekty VBO oraz obiekt VAO. Na razie nie wpisujemy do nich żadnych danych. Będzie to robione dynamiczne podczas renderowania. Kolejnym krokiem jest obliczenie współrzędnych czterech wierzchołków elementu GUI oraz przeliczenie tych współrzędnych z pikseli na współrzędne ekranu (od -1.0 do +1.0). Poza konstruktorem jest także destruktor, który zajmuje się usuwaniem niepotrzebnych już obiektów VBO oraz VAO.

Konieczne jest zdefiniowanie także kilku innych dodatkowych funkcji, które m.in. pozwolą skonfigurować kolor tła oraz obramowania. Są to funkcje `setBackgroundColours()` oraz `setBorderColour()`.

Ważnym składnikiem klasy `GUIElement` jest czysto wirtualna funkcja `render()`. Każda klasa pochodna po tej klasie będzie musiała zdefiniować własną wersję tej funkcji, która będzie rysowała obiekt tej klasy na ekranie. Zaletą takiego rozwiązania jest to, że będziemy mogli mieć na przykład kontener różnych elementów GUI i rysować je wszystkie za pomocą wskaźnika do klasy bazowej.

Tym sposobem omówiona została klasa bazowa `GUIElement`. Na jej podstawie napiszemy teraz klasę `GUIProgressBar`.

```
class GUIProgressBar : public GUIElement
{
public:
    GUIProgressBar(int x, int y, int width, int height) : GUIElement{x, y, width, height}
    {
        createShaderProgram(gui_shader_, "gui_progressbar_vs.gls1",
                            "gui_progressbar_fs.gls1");

        alpha_uniform_ = glGetUniformLocation(gui_shader_, "alpha_factor");
        border colour uniform_ = glGetUniformLocation(gui shader , "border colour");
        rendering border uniform_ = glGetUniformLocation(gui shader , "rendering border");
    }

    void render()
    {
        // ...
    }
}
```

```

void setAlpha(int value)
{
    alpha_ = value;

    if (alpha_ > 100)
        alpha_ = 100;
    if (alpha_ < 0)
        alpha_ = 0;
}

void setBarColours(const glm::vec3 &colour1, const glm::vec3 &colour2)
{
    bar colour 1 = colour1;
    bar colour 2 = colour2;
}

void setValue(int value)
{
    value_ = value;

    if (value_ > 100)
        value_ = 100;
    if (value_ < 0)
        value_ = 0;
}

protected:
    GLint alpha_uniform_{0};
    GLint border_colour_uniform_{0};
    GLint rendering_border_uniform {0};
    glm::vec3 bar colour 1 {0.8f, 0.0f, 0.0f};
    glm::vec3 bar colour 2 {0.6f, 0.0f, 0.0f};
    int alpha_{100};
    int value_{0};
    const std::vector<GLuint> bar_indices{4, 5, 6, 6, 5, 7};

};

```

Konstruktor tej klasy musi przede wszystkim wywołać konstruktor klasy bazowej `GUIElement`. Przekazuje mu takie parametry jak położenie elementu GUI na ekranie oraz jego szerokość i wysokość. Wewnątrz konstruktora klasy `GUIProgressBar` nie dzieje się już dużo. Jedyne co robimy, to ładowamy program shadera dla paska postępu oraz znajdujemy lokalizacje interesujących nas zmiennych typu `uniform`.

Klasa `GUIProgressBar` definiuje także kilka dodatkowych funkcji:

- `setAlpha()` - ustawia przezroczystość tego elementu GUI,
- `setBarColours()` - umożliwia skonfigurowanie kolorów (gradient) paska postępu,
- `setValue()` - ustawia wartość paska postępu w zakresie od 0 do 100% wypełnienia.

Najciekawszym i jednocześnie najważniejszym elementem klasy jest funkcja `render()`. Zajmowała się będzie ona narysowaniem paska postępu na ekranie.

```

void render()
{
    int progress end x = x_ + value_ / 100.0 * width;

    std::vector<glm::vec3> progress_bar_vertices(4);
    progress_bar_vertices[0] = glm::vec3(x_, y_, 0.0f);
    progress_bar_vertices[1] = glm::vec3(progress_bar_vertices[0].x,
                                         progress_bar_vertices[0].y + height_,
                                         0.0f);
    progress_bar_vertices[2] = glm::vec3(progress_end_x, progress_bar_vertices[0].y, 0.0f);
    progress_bar_vertices[3] = glm::vec3(progress_bar_vertices[2].x,
                                         progress_bar_vertices[1].y,
                                         0.0f);
}

```

```

for (auto &vertex : progress_bar_vertices)
{
    vertex.x = calculateScreenCoordX(vertex.x);
    vertex.y = calculateScreenCoordY(vertex.y);
}

std::vector<GLfloat> vertices buffer;

for (auto &vertex : border vertices )
{
    vertices_buffer.push_back(vertex.x);
    vertices_buffer.push_back(vertex.y);
    vertices_buffer.push_back(vertex.z);
}

for (auto &vertex : progress bar vertices)
{
    vertices_buffer.push_back(vertex.x);
    vertices_buffer.push_back(vertex.y);
    vertices_buffer.push_back(vertex.z);
}

std::vector<glm::vec3> colours(8);
colours[0] = background_colour_1_;
colours[1] = background colour 2 ;
colours[2] = background colour 1 ;
colours[3] = background colour 2 ;
colours[4] = bar_colour_1_;
colours[5] = bar_colour_1_;
colours[6] = bar colour 2 ;
colours[7] = bar colour 2 ;

std::vector<GLfloat> colours_buffer;
for (auto &colour : colours)
{
    colours_buffer.push_back(colour.r);
    colours_buffer.push_back(colour.g);
    colours_buffer.push_back(colour.b);
}

glBindVertexArray(handle_);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo );
glBufferData(GL_ARRAY_BUFFER, vertices_buffer.size() * sizeof(GLfloat),
            vertices_buffer.data(), GL_DYNAMIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, colours_vbo_);
glBufferData(GL_ARRAY_BUFFER, colours_buffer.size() * sizeof(GLfloat),
            colours_buffer.data(), GL_DYNAMIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices_vbo_);

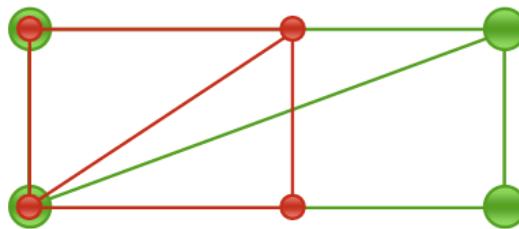
glUseProgram(gui shader );
glUniformi(alpha uniform , alpha );

glDisable(GL_DEPTH_TEST);
// Narysuje tlo elementu GUI
glUniformi(rendering_border_uniform , 0); // Rysujemy elementy dwukolorowe
glBufferData(GL ELEMENT ARRAY BUFFER, fill_indices .size() * sizeof(GLuint),
             fill_indices .data(), GL_DYNAMIC_DRAW);
glDrawElements(GL_TRIANGLES, fill_indices_.size(), GL_UNSIGNED_INT, 0);
// Narysuje pasek postepu
glBufferData(GL ELEMENT ARRAY BUFFER, bar indices.size() * sizeof(GLuint),
             bar indices.data(), GL_DYNAMIC_DRAW);
glDrawElements(GL_TRIANGLES, bar indices.size(), GL UNSIGNED INT, 0);
// Narysuje ramke dookola elementu GUI
glUniformi(rendering_border_uniform_, 1); // Rysujemy elementy jednokolorowe
glUniform3fv(border_colour_uniform_, 1, glm::value_ptr(border_colour_));
glBufferData(GL ELEMENT ARRAY BUFFER, border indices .size() * sizeof(GLuint),
             border indices .data(), GL_DYNAMIC DRAW);
glDrawElements(GL LINE LOOP, border indices .size(), GL UNSIGNED INT, 0);

 glEnable(GL_DEPTH_TEST);
 glBindVertexArray(0);
}

```

Pasek postępu zbudowany jest z dwóch podelementów: obramowania wraz z tłem (te wierzchołki mamy zdefiniowane w klasie bazowej) oraz właściwego paska postępu zbudowanego także z czterech wierzchołków (no i obramowanie jest trzecim podelementem, ale ma wszystkie wierzchołki wspólne z tłem, dlatego do tych rozważań je pomijam) (rys. 40.2). Dwa wierzchołki właściwego paska postępu pokrywają się z dwoma wierzchołkami obramowania (dwa skrajne lewe wierzchołki). Możemy zdefiniować je raz i potem wykorzystać je dla obu tych podelementów. Jednak w celu uproszczenia kodu dla podelementu paska postępu definiuje je powtórnie. A poza tym, dzięki temu można przesunąć podelement paska postępu w inne dowolne miejsce, np. bardziej w prawo (żeby nie zaczynał się od samej krawędzi elementu GUI).



Rys. 40.2. Poglądowe ułożenie wierzchołków paska postępu. Wierzchołki zielone to obramowanie (tło), a czerwone to właściwy pasek postępu.

Na samym początku funkcji `render()` musimy obliczyć wierzchołki właściwego paska postępu. Znając szerokość i położenie X na ekranie elementu GUI oraz wartość paska postępu, jesteśmy w stanie obliczyć gdzie (położenie w pikselach) powinien kończyć się właściwy pasek postępu (wskaazywana przez niego wartość). W tym celu przemnażamy szerokość elementu GUI przez procentową wartość paska postępu oraz dodajemy współrzedną X położenia. Otrzymana wartość wędruje to zmiennej `progress_end_x`.

Następnym krokiem jest obliczenie wierzchołków właściwego paska postępu, przeliczenie ich na współrzędne ekranu oraz wpisanie wszystkich wierzchołków do bufora `vertices_buffer`. Wpisujemy zarówno wierzchołki obramowania jak i wierzchołki paska postępu.

Musimy także stworzyć bufor, w którym będą znajdowały się kolory wierzchołków. W tym celu tworzymy tablicę `colours`, do której wpisujemy w pierwszej kolejności kolory wierzchołków tła, a następnie kolory wierzchołków właściwego paska postępu. Na koniec wszystko wpisywane jest do bufora `colours_buffer`.

Gdy już wszystkie dane są gotowe można zacząć rysowanie. Aktywujemy obiekt VAO oraz wpisujemy do obiektów VBO współrzędne wierzchołków oraz ich kolory. Bindujemy także obiekt VBO, do którego będziemy wpisywać podczas rysowania indeksy wierzchołków. Dalej aktywujemy program shadera i wysyłamy do niego wartość przezroczystości paska postępu. Należy również pamiętać o wyłączeniu testowania głębi, aby potem nie było niemiłych dla oka niespodzianek.

W pierwszej kolejności rysujemy tło elementu GUI. Przed wywołaniem funkcji rysującej musimy wysłać do programu shadera wartość informującą shader o tym, że będziemy rysować powierzchnie dwukolorowe (`rendering_border_uniform_`). Do narysowania

wypełnienia elementu GUI wykorzystujemy indeksy z kontenera `fill_indices_`. W dalszej kolejności rysujemy właściwy pasek postępu, a na samym końcu obramowanie całego elementu GUI. Przed każdym rysowaniem należy pamiętać o zmianie w obiekcie VBO aktualnych indeksów wierzchołków, a przed rysowaniem obramowania o wysłaniu odpowiedniego stanu do programu shadera.

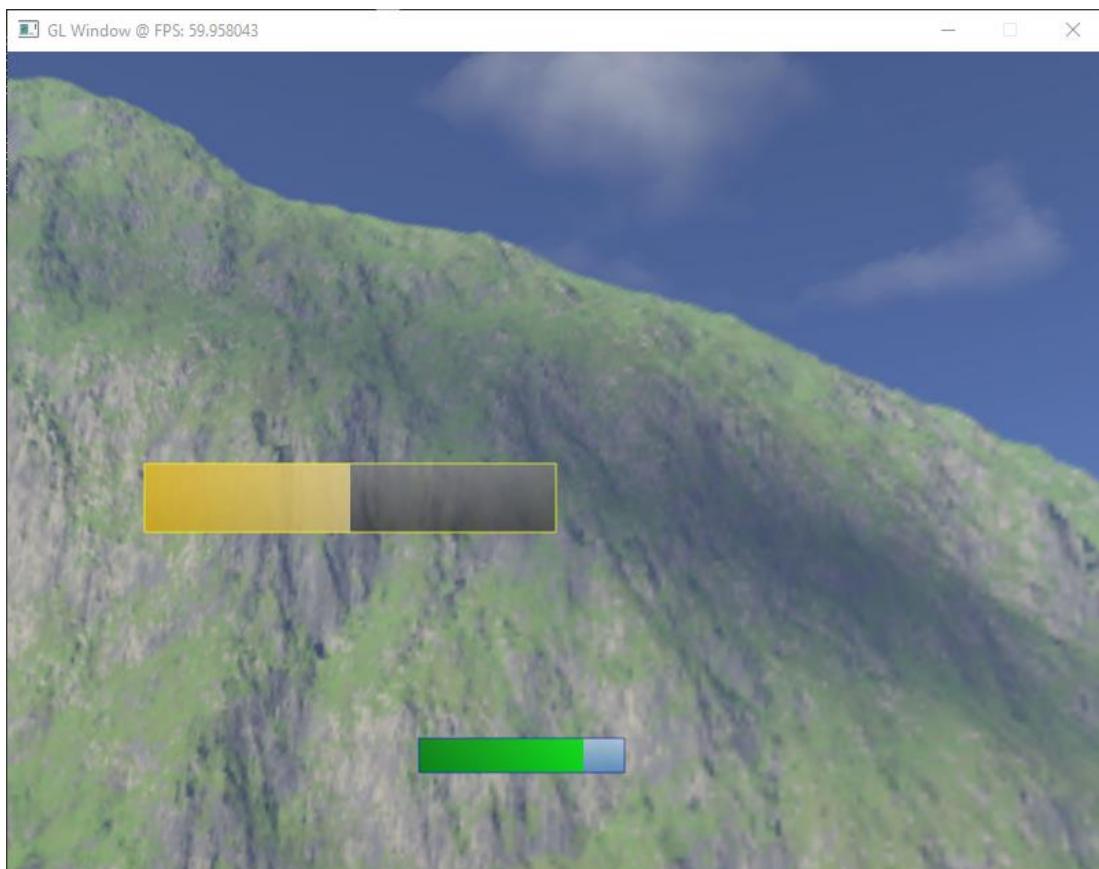
Klasa `GUIProgressBar` jest już gotowa. Przeszędł czas na utworzenie obiektów tej klasy i sprawdzenie jak to wszystko działa.

```
GUIElement::setViewportSize(window_width, window_height);

GUIProgressBar progress_bar(100, 300, 300, 50);
progress_bar.setAlpha(65);
progress_bar.setValue(0);
progress_bar.setBackgroundColours(glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(0.2f, 0.2f, 0.2f));
progress_bar.setBorderColour(glm::vec3(1.0f, 1.0f, 0.0f));
progress_bar.setBarColours(glm::vec3(1.0f, 0.80f, 0.06f), glm::vec3(1.0f, 0.90f, 0.55f));

GUIProgressBar progress_bar_2(300, 500, 150, 25);
progress_bar_2.setAlpha(85);
progress_bar_2.setValue(80);
progress_bar_2.setBackgroundColours(glm::vec3(0.68f, 0.79f, 0.88f), glm::vec3(0.32f, 0.55f,
0.75f));
progress_bar_2.setBorderColour(glm::vec3(0.0f, 0.0f, 0.65f));
progress_bar_2.setBarColours(glm::vec3(0.0f, 0.50f, 0.0f), glm::vec3(0.0f, 0.87f, 0.0f));
```

Ostatecznie otrzymamy rezultat jak na rys. 40.3. Wyrenderowane zostały dwa paski postępu według naszych założeń.



Rys. 40.3. Wyrenderowane paski postępu.

Aby przezroczystość działała prawidłowo nie możemy zapomnieć włączyć jej w kodzie programu:

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Kod źródłowy z tego rozdziału jest w katalogu: 40_GUI/ProgressBar

41. ???

Książka jest stale aktualizowana.
Po aktualną wersję zapraszam na stronę <http://kurs-opengl.pl/>