OpenGLFramework

v.1.05

Wersja dokumentu: 1.0

Autor: Roman "PlayeRom" Ludwicki
Kontakt: romek21@op.pl

Spis treści

1. Wstęp

OpenGLFramework jest to prosty szablon, na podstawie którego możemy budować aplikacje (w tym gry) oparte na bibliotece OpenGL. I takie też było jego główne założenie. OpenGLFramework zawiera więc podstawowe funkcje związane chyba z każdą grą. Całą resztę, tj. funkcjonalność, mechanizm gry należy już wykonać we własnym zakresie, dokładając do opisywanego tu szablonu własne klasy/moduły.
Na stan obecny OpenGLFramework zawiera:

- wczytywanie tekstur (nearest, linear oraz mipmap) z plików BMP oraz TGA,
- multiteksturing,
- obsługa dźwięku za pomocą biblioteki FMOD oraz OpenAL,
- wyświetlanie tekstu – aż pięć możliwość,
- obliczanie i wyświetlanie FPS (ilość generowanych klatek na sekundę),
- kontrola prędkości animacji, dzięki czemu animacje wyświetlane będą z prędkością niezależną od FPS,
- prosty profiler do sprawdzania czasu wykonania się różnych funkcji,
- konsola do wyświetlania i wprowadzania komunikatów,
- wspieranie UNICODE,
- wspieranie wielojęzyczności,
- wspieranie oświetlania i materiałów,
- pełnoekranowy antyaliasing,
- logger,
- tryb 2D oraz 3D z możliwością przełączania się,
- wykorzystanie VertexArrays,
- wykorzystanie Vertex Buffer Object,
- własny format pliku *3DObj* dla modeli 3D,
- włączanie / wyłączanie synchronizacji pionowej,
- emboss bump mapping (mapowanie wybojów),
- prosty system cząsteczkowy,
- rysowanie billboard'ów (sprite'ów zawsze zwróconych przodem do widza),
- możliwość zmiany, np. rozdzielczości, głębi kolorów, próbkowania dla antyaliasingu, itp. bez utraty aktualnego stanu gry,
- możliwość tworzenia cieni metodą shadow volume.

2. Aby projekt się skompilował

OpenGLFramework wykorzystuje takie biblioteki jak:

- `STL` minimum `v.1.2.11`,
- `SDL_ttf` minimum `v.2.0.9`,
- `FMOD v.3.74` i/lub `OpenAL` w zespół z `alut v.1.1.0`,
- `OpenGL` minimum `v.1.5`.

Oznacza to, że aby skompilować projekt należy posiadać biblioteki `SDL`, `SDL_ttf`, `FMOD` i/lub `OpenAL` z `alut` – możemy pobrać je z internetu (pozostałe powinniśmy już mieć), linki znajdują się w dziale *31. Bibliografia*. Oczywiście musimy w Visual C++ wskazać ścieżki do folderów *include* oraz *lib* – dla każdej z biblioteki z osobna. Dla VC++ 2005 Express Edition należy z menu „Tools" wybrać „Options". Następnie przejść na zakładkę „Projects and Solutions" → „VC++ directiories". Do głównego katalogu projektu należy także skopiować pliki DLL i są to: *fmod.dll* (o ile zamierzamy korzystać z biblioteki `FMOD` a nie z `OpenAL`), *alut.dll*, *OpenAL32.dll*, *ogg.dll*, *vorbis.dll* i *vorbisfile.dll* (o ile zamierzamy korzystać z biblioteki `OpenAL` a nie `FMOD`), *SDL.dll* oraz *SDL_ttf.dll*.

3. Schemat projektów

Cały projekt frameworka składa się na szereg mniejszych projektów. Poniżej znajduje się ich lista wraz z zawartymi klasami:

OpenGLFramework:

- `OpenGLFramework` – główny projekt zawierający następujące klasy:
  - *OpenGLFramework.cpp*
  - `CConsole`
  - `CCursor`
  - `CFramework`
  - `CGameControl`
  - `CWindowData`
- `Draw` – projekt zawierający wszelkie klasy związane z rysowaniem:
  - `C3DObjManager`
  - `C3DObjFile`
  - `CBillboard`
  - `CBrightness`
  - `CEmbossBump`
  - `CFog`
  - `CParticles`
  - `CTextureLoader`
  - `CVector2`, `CVector3`, `CVector4`
  - `CVertexArray`
- `Extensions` – projekt zwierający rozszerzenia OpenGL
  - `CARBMultisample`
  - `CARBMultiTexturing`
  - `CARBVertexBufferObject`
  - `CWGLEXTSwapControl`
- `Fonts` – tutaj znajdują się wszelkie klasy odpowiedzialne za rysowanie tekstu
  - `CBitmapFont`
  - `CMyFont`
  - `COutlineFont`
  - `CSDLFont`
  - `CTextureFont`
- `Fps` – projekt z klasą odpowiedzialną za wyliczanie i wyświetlanie ilości FPS
  - `CFps`
- `Lighting` – projekt odpowiedzialny za ustawienie oświetlenia i materiałów
  - `CLighting`
- `Logger` – projekt z klasą odpowiedzialną za rejestrowanie komunikatów do pliku
  - `CLogger`
- `MatrixOperation` – projekt z klasą odpowiedzialną za ręczne przekształcenia 3D
  - `CMatrixOperiation`
- `MultiLanguage` – projekt odpowiedzialny za obsługę wielojęzyczności
  - `CMultiLanguage`
- `Sound` – projekt ze wszelkimi klasami związanymi z odgrywaniem dźwięku, muzyki, itp.
  - `IVolume`
  - `CVolumeOutMaster`
  - `CMasterVolume`
  - `COpenALManager`
  - `CSoundOpenAL`
  - `COpenALSources`
  - `CSoundFMOD`

- `SpeedControl` – projekt niezbędny do kontrolowania prędkości wszelkich animacji
  - `CSpeedControl`
  - `CProfiler`

Używane przedrostki przy nazwach zmiennych:

Przedrostek	*Znaczenie*
`m_`	Składowa klasy
`g_`	Zmienna globalna
`b`	Typ `BOOL` (`GLboolean`)
`i`	Typ `int` (`GLint`)
`ui`	Typ `unsigned int` (`GLuint`)
`f`	Typ `float` (`GLfloat`)
`s`, `S`	Struktura
`C`	Klasa
`a`	Tablica
`c`	Tablica znaków `char` lub `wchar_t`
`lp`	Wskaźnika na łańcuch znaków `char` lub `wchar_t`
`h`	Uchwyt
`ub`	Typ `GLubyte`
`p`	Wskaźnik na jakiś obiekt
`rc`	Typ `RECT`
`e`, `E`	Typ wyliczeniowy
`in_`	Przy referencyjnych parametrach funkcji, informuje że dany parametr służy do wprowadzenia danych
`out_`	Przy referencyjnych parametrach funkcji, informuje że dany parametr zwraca wynik działania funkcji
`in_out_`	Przy referencyjnych parametrach funkcji, informuje że dany parametr służy do wprowadzenia danych oraz do ich wyprowadzenia

4. Klasa framework'a – `CFramework`

Główną klasą framework'a, od której rozpoczyna się jego praca jest klasa `CFramework` projektu `OpenGLFramework`. Klasa ta udostępnia tylko jedną metodę publiczną `Run()`. Wewnątrz tej metody inicjalizowane jest okno aplikacji, inicjalizowany jest OpenGL oraz inne obiekty niezbędne do funkcjonowania frameworka. W metodzie `Run()` znajduje się także główna pętla programu, w której przetwarzane są komunikaty i rysowana jest cała scena.
Instancja klasy `CFramework` tworzona jest w pliku *OpenGLFramework.cpp* znajdujący się w projekcie `OpenGLFramework`. Plik *OpenGLFramework.cpp* posiada główną funkcję aplikacji `WinAPI`:

```
int WINAPI WinMain( HINSTANCE hInstance,
```

```
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int iCmdShow );
```

5. Klasa zarządzania grą – `CGameControl`

Jest to klasa odpowiedzialna za zarządzanie tworzoną grą. Jako, że jest klasą newralgiczną dla projektu i zawiera ważne dane programu, `CGameControl` jest singletonem. Klasa ta tworzy większość obiektów opisywanych tu klas oraz wskazane jest aby obiekty naszych nowo dodanych klas także były tutaj tworzone. Nowe obiekty tworzymy w metodzie `CGameControl::CreateObjects()`. Pamiętajmy także o ich zwalnianiu w metodzie `CGameControl::DeleteObjects()`. Jeżeli nasze obiekty mają coś zainicjalizować w trakcie uruchamiania się programu, to metody te należy wywołać w `CGameControl::Initialization()`. Ważna jest także metoda `CGameControl::RestartObjects()`, jest ona wywoływana gdy restartowane jest okno, np. pod wpływem zmiany rozdzielczości, próbkowania dla antyaliasingu, itp. W metodzie tej należy wywołać funkcję, np. `RestartObjects()` naszych obiektów, które mogą zostać utracone pod wpływem zmian parametrów okna, jak np. rozdzielczość. W ostateczności możemy tutaj zniszczyć nasz obiekt i utworzyć ponownie nowy.
Poza tym klasa `CGameControl` otrzymuje wszelkie zdarzenia klawiatury i myszy oraz obsługuje je bądź przekazuje innym klasom. W metodzie `CGameControl::Draw()`, natomiast, rysujemy cały nasz wirtualny świat. Ogólnie mówiąc jest to klasa, od której rozpoczynamy tworzenie naszej gry. Więcej szczegółów w rozdziale *20. Schemat Framework'a oraz dołączanie własnych klas*.

6. Klasa okna – `CWindowData`

Klasa `CWindowData` projektu `OpenGLFramework` odpowiada za tworzenie i usuwanie okna oraz co ważne przechowuje różne zmienne takie jak:

- rozdzielczość ekranu,
- głębia kolorów,
- kolor tła,
- nazwę okna i klasy okna,
- aktywność antyaliasingu, trybu pełnoekranowego, wyświetlanie kursora

Większość z tych zmiennych możemy modyfikować wedle własnego uznania – a robimy to w konstruktorze opisywanej tu klasy `CWindowData`.
Jako, że `CWindowData` jest klasą newralgiczną dla projektu i zawiera ważne dane programu, `CWindowData` jest singletonem. Patrz także rozdział *20. Schemat Framework'a oraz dołączanie własnych klas*.

7. Plik *define.h*

Plik *define.h*, zlokalizowany w projekcie `OpenGLFramework`, zawiera wszelkie zdefiniowane (bądź „zakomentowane”, tj. nieużywane) makra, które sterują pewną funkcjonalnością opisywanego tu frameworka. I tak w pliku tym znajdują się następujące zestawy makr:

- `UNICODE` – zdefiniowanie tego makra oznacza, że nasza aplikacja wykorzystywać będzie ten standard kodowania. Jeżeli nie zdefiniujemy tego makra to użyte zostanie ASCII. Jednak dla VC++ 2005 Express Edition makro to definiujemy we właściwościach projektu a nie w pliku *define.h*. Patrz także rozdział *15. Obsługa UNICODE*.

- _USE_CONSOLE_WITH_MY_FONT_ oraz _USE_CONSOLE_WITH_BITMAP_FONT_ – za pomocą tego zestawu makr możemy zmieniać rodzaj czcionki i sposób rysowania tekstu w konsoli. Jeżeli zdefiniowane jest pierwsze makro to w konsoli, do rysowania tekstu użyta zostanie klasa CMyFont (patrz rozdział *11.1. Wyświetlanie tekstu za pomocą CMyFont*). Jeżeli zdefiniowane będzie drugie makro to użyta zostanie klasa CBitmapFont (patrz rozdział *11.2. Wyświetlanie tekstu za pomocą CBitmapFont*). W przypadku niezadeklarowania żadnego z makr użyte zostanie rysowanie z CSDLFont (patrz rozdział *11.4. Wyświetlanie tekstu za pomocą CSDLFont*). W przypadku zadeklarowania obydwu makr, użyte zostanie CSDLFont.
- _QUESTION_FULLSCREEN_ – zdefiniowanie tego makra spowoduje, iż w trakcie uruchamiania naszej aplikacji, zostaniemy zapytani czy program ma się uruchomić w trybie pełnoekranowym czy też nie. Jeżeli makro jest niezdefiniowane (i tak jest domyślnie) – to program zawsze uruchomi się na pełnym ekranie.
- _USE_SYSTEM_RES_AND_BITSPERPIXEL_ – zdefiniowanie tego makra powoduje, że program przed uruchomieniem pobierze rozdzielczość ekranu oraz głębię kolorów z ustawień systemowych pulpitu. Oznacza to, że ustawienia powyższych parametrów w klasie CWindowData zostaną zignorowane. Aby stała się dokładnie na odwrót, makro to należy zakomentować.
- _USE_2D_ONLY_ – makro to należy zdefiniować, jeżeli nasza gra będzie wykorzystywać grafikę tylko w trybie 2D. Jeżeli gra ma choć w niewielkim stopniu wykorzystywać grafikę trójwymiarową to oczywiście makro to nie może być zdefiniowane i tak też jest domyślnie.
- _USE_SOUND_OPENAL_ – jeżeli makro to jest zdefiniowane to do odgrywania dźwięków i muzyki użyta zostanie biblioteka OpaneAL. Jeżeli zakomentujemy to makro to użyta zostanie biblioteka FMOD. Domyślnie makro _USE_SOUND_OPENAL_ jest zdefiniowane.
- _USE_PROFILER_ – wygodnie jest wywoływać funkcje profilera w opisywanym tu makrze. Dzięki temu gdy w wersji release będziemy chcieli pozbyć się profilera wystarczy zakomentować to makro. Patrz także rozdział *29. Profiler*.

Jeżeli w trakcie pisania własnego programu będziesz potrzebował nowych, własnych makr zawsze możesz dopisywać je do pliku *define.h*. Plik *define.h* jest automatycznie widziany w każdym pliku – za pomocą *stdafx.h*.

8. Mgła

Do ustawienia parametrów mgły dla OpenGL, służy klasa CFog projektu Draw. Obiekt tej klasy nie jest nigdzie stworzony, więc aby skorzystać z tej klasy najpierw musimy ją utworzyć, np. operatorem new. Konstruktor klasy CFog automatycznie przypisze nam biały kolor mgły. Oczywiście możemy to zmienić poprzez interfejs klasy, który przedstawia się następująco:

```
GLvoid CFog::SetColor( GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f );
```

Parametry:
r, g, b, a Składowe nowego koloru mgły.

Za pomocą poniższej funkcji ustawiamy typ mgły:

```
GLvoid CFog::SetFogMode( GLfloat fParam );
```

Parametry:
fParam Typ mgły, do wyboru GL_LINEAR, GL_EXP oraz GL_EXP2. Po więcej szczegółów, patrz

dokumentacja OpenGL.

Za pomocą poniższej funkcji ustawiamy początek mgły względem obserwatora:

```
GLvoid CFog::SetStart( GLfloat fStart );
```

Parametry:
`fStart` Wartość początku mgły względem obserwatora.

Za pomocą poniższej funkcji ustawiamy koniec mgły względem obserwatora:

```
GLvoid CFog::SetEnd( GLfloat fEnd );
```

Parametry:
`fEnd` Wartość końcowa mgły względem obserwatora.

Kolejna funkcja umożliwia nam ustalenie zakresu mgły d razu podając jej początek i koniec:

```
GLvoid CFog::SetStartEnd( GLfloat fStart, GLfloat fEnd );
```

Parametry:
`fStart` Wartość początku mgły względem obserwatora.
`fEnd` Wartość końcowa mgły względem obserwatora.

Za pomocą ostatniej funkcji ustalamy natężenie mgły:

```
GLvoid CFog::SetDensity( GLfloat fDensity );
```

Parametry:
`fDensity` Wartość natężenia mgły w zakresie od 0.0f do 1.0f, gdzie 0.0f to bardzo rzadka mgła – w sumie nieistniejąca, 1.0f to mgła bardzo gęsta.

Musimy także pamiętać o ręcznym włączeniu i wyłączeniu mgły wedle naszego uznania. Dokonamy tego za pomocą funkcji OpenGL: `glEnable( GL_FOG )` oraz `glDisable( GL_FOG )`.

9. Tekstury

Za wczytywanie tekstur odpowiada klasa `CTextureLoader` zawarta w projekcie `Draw`. Mamy możliwość wczytywania plików BMP, TGA oraz JEH z trzema możliwościami tworzenia tekstury, tj. nearest, linear oraz mipmap. Cała obsługa wczytywania tekstur jest już zapewniona. Naszym zadaniem jest tylko wczytać poszczególne pliki w metodzie `CGameControl::LoadAllTextures()`. Funkcja ta jest już wywoływana w `CGameControl::Initialization()`. Do wyboru mamy trzy funkcje wczytujące pliki BMP, TGA oraz JEH, a jedynym ich parametrem jest nazwa pliku, który chcemy uczynić teksturą:

Tworzy najlepiej wyglądającą teksturę mipmapową:
```
GLint CTextureLoader::LoadTextureMipmaps( LPCTSTR lpFilename );
```

Tworzy teksturę typu linear:
```
GLint CTextureLoader::LoadTexture( LPCTSTR lpFilename );
```

Tworzy najgorzej wyglądającą teksturę typu nearest:

```
GLint CTextureLoader::LoadTextureLowQuality( LPCTSTR lpFilename );
```

Parametry dla powyższych funkcji:

`lpFileName` Nazwa pliku obrazu, który chcemy uczynić teksturą.

Wartość zwracana dla powyższych funkcji:

Indeks nowej tekstury. W przypadku błędu zwracana jest wartość -1.

Teraz jeżeli w trakcie renderingu potrzebujemy danej tekstury należy wywołać poniższą funkcję (zamiast `glBindTexture( GL_TEXTURE_2D, uiTexture )`):

```
GLboolean CTextureLoader::SetTexture( GLint iIndex = 0 );
```

Parametry:

`iIndex` Indeks tekstury, którą chcemy uaktywnić. Indeksy liczone są od 0 w kolejności wczytywania tekstur.

Wartość zwracana:

`GL_TRUE` w przypadku powodzenia.

9.1. Multiteksturing

OpenGLFramework zawiera jakby dwa rodzaje multiteksturingu. Pierwszy rodzaj to takie pseudo multiteksturowanie, drugi to już te właściwe wspierane przez karty graficzne wyposażone co najmniej w dwie jednostki teksturujące.

1. Pseudo-multiteksturing

Pseudo-multiteksturing możemy wykorzystać przy sprite'ach z użyciem specjalnej maski do wskazania obszarów tekstury, które mają być niewidoczne. Dzięki temu w wydajny sposób (rysując obiekt tylko jeden raz) możemy uzyskać dowolne kształty sprite'a. Metoda ta polega na wczytaniu dwóch tekstur, gdzie pierwsza to tzw. maska – czyli tekstura w odcieniach szarości, gdzie czarny kolor oznacza całkowitą przezroczystość a biały całkowitą widoczność elementów drugiej tekstury. Druga tekstura to właściwy, w pełni kolorowy obraz. Następnie program spreparuje obydwie tekstury do jednej, którą możemy wykorzystać. Przykładowy kod zawarty we framework'u obrazuje wykorzystanie tej metody rysując gwiazdę. Aby wykorzystać tę metodę „multiteksturingu” należy po pierwsze przygotować dwie, opisane powyżej tekstury. Następnie owe tekstury musimy wczytać, robimy to w metodzie `CGameControl::LoadAllTexture()`, gdzie wywołujemy metody:

```
GLint CTextureLoader::LoadMultiTextureMaskMipmaps( LPCTSTR lpFileNameMask,
                                                   LPCTSTR lpFileNameRGB );

GLint CTextureLoader::LoadMultiTextureMask( LPCTSTR lpFileNameMask,
                                            LPCTSTR lpFileNameRGB );

GLint CTextureLoader::LoadMultiTextureMaskLowQuality( LPCTSTR lpFileNameMask,
                                                      LPCTSTR lpFileNameRGB );
```

Parametry:

`lpFileNameMask` Nazwa pliku z maską.

`lpFileNameRGB` Nazwa pliku z normalną, kolorową teksturą.

Wartość zwracana:
Indeks nowej tekstury. W przypadku błędu zwracana jest wartość -1.

Należy pamiętać, że pomimo podania dwóch plików, powyższe funkcje utworzą jedną teksturę czyli zostanie utworzony jeden indeks tekstury – pamiętajmy o tym przy wywoływaniu `CTextureLoader::SetMultiTextures()`. Poza tym, jak wynika z nazwy powyższych funkcji, wygenerowane zostaną kolejno tekstury: mipmapowa, linear oraz nearest.

Powyższe funkcje, tj. `CTextureLoader::LoadMultiTextureMask...()` mają swoją drugą odmianę, która to pozwoli nam w pewnych okolicznościach zaoszczędzić pamięć oraz miejsce na dysku. Otóż w owych odmiennych metodach, zamiast podawać drugą nazwę pliku z teksturą kolorową, należy podać sam kolor jako `COLORREF`. Metody te przydadzą nam się jeżeli druga, kolorowa tekstura będzie po prostu w jednolitym kolorze. Wtedy nie musimy tworzyć jej jako tekstury (i niepotrzebnie tracić pamięć) tylko wykorzystujemy poniższą funkcję, gdzie bezpośrednio podajemy jej kolor:

```
GLint CTextureLoader::LoadMultiTextureMaskMipmaps( LPCTSTR lpFileNameMask,
                                                   COLORREF crColor );

GLint CTextureLoader::LoadMultiTextureMask( LPCTSTR lpFileNameMask,
                                            COLORREF crColor );

GLint CTextureLoader::LoadMultiTextureMaskLowQuality( LPCTSTR lpFileNameMask,
                                                      COLORREF crColor );
```

Parametry:
`lpFileNameMask` Nazwa pliku z maską.

`crColor` Kolor jednolitej „tekstury”, podajemy w makrze `RGB(...)`, np. dla koloru czerwonego: `RGB(255, 0, 0)`.

Wartość zwracana:
Indeks nowej tekstury. W przypadku błędu zwracane jest -1.

Na koniec aby wykorzystać naszą pseudo multiteksturę, oczywiście przed narysowaniem poligonu musimy ją uaktywnić. Aby to uczynić przed rysowaniem danego obiektu należy wydołać poniższą metodę (taki odpowiednik `CTextureLoader::SetTexture()` dla naszej multitekstury):

```
GLboolean CTextureLoader::SetMultiTextures( GLint iIndex );
```

Parametry:
`iIndex` Indeks tekstury, którą chcemy uaktywnić. Indeksy liczone są od 0 w kolejności wczytywania tekstur.

Wartość zwracana:
`GL_TRUE` w przypadku powodzenia.

2. Właściwy multiteksturing

Pora przejść do omówienia drugiego rodzaju multiteksturowania, a jest to już ten właściwy, wspierany przez karty graficzne. Dla wykorzystania multiteksturingu stworzona została specjalna klasa `CARBMultiTexturing` (zawarta w projekcie `Extensions`) będąca singletonem. Multiteksturing możemy wykorzystywać zarówno przy rysowaniu za pomocą VertexArrays oraz metodą podstawową wykorzystując ręczne przypisywanie współrzędnych tekstur danym wierzchołkom.
Aby wykorzystać sprzętowy multiteksturing na początek należy wczytać dane tekstury. Wykonujemy to w normalny sposób opisany powyżej czyli wywołujemy np: `CTextureLoader::LoadTextureMipmaps()` dla każdego pliku. Należy jedynie pamiętać aby pliki z teksturami, które będą nakładane na ten sam obiekt były wczytywane jedna po drugiej – wówczas ich indeksy będą następowały po sobie co jest wymagane!
Po wczytaniu tekstur możemy je wykorzystać w procesie renderingu. Aby nałożyć nasze tekstury na ten sam obiekt musimy wywołać poniższą metodę (np. zamiast `CTextureLoader::SetTexture()`, która uaktywnia tylko jedną teksturę):

```
GLboolean CARBMultiTexturing::BindMultiTextures( GLint iFirstTex,
                                                 GLsizei iCount = 2 );
```

Parametry:

`iFirstTex` Indeks pierwszej tekstury. Indeksy liczone są od 0 w kolejności wczytywania tekstur.

`iCount` Ilość tekstur jaką chcemy nałożyć na dany obiekt. Minimalna i domyślna ilość to dwie tekstury. Wartość ta mówi także ile kolejnych tekstur, licząc od indeksu podanego w `iFirstTex`, należy wykorzystać – dlatego tekstury nakładane na ten sam obiekt należy wczytywać jedna po drugiej. Opisywany tu parametr mówi także, ile kolejnych jednostek teksturujących należy wykorzystać.

Wartość zwracana:
`GL_TRUE` w przypadku powodzenia uaktywnienia multiteksturowania.

Następnie możemy rozpocząć rysowanie danego obiektu, gdzie do przypisywania koordynatów położenia tekstury dla danego wierzchołka, należy wywołać poniższą metodę (odpowiednik `glTexCoord2f`):

```
GLboolean CARBMultiTexturing::MultiTexCoord2fARB( GLfloat s, GLfloat t );
```

Parametry:
`s, t` Koordynaty położenia tekstury.

Wartość zwracana:
`GL_TRUE` w przypadku powodzenia.

Po zakończeniu rysowania obiektu z mulitekstурami, należy powrócić do renderowania jedną jednostką teksturującą, czyli pozostałe należy wyłączyć. Dokonamy tego wywołując poniższą funkcję:

```
GLvoid CARBMultiTexturing::DisableMultiteksturing();
```

Przykład:

```
//na obiekt nakładane są dwie tekstury o indeksach 4 i 5
CARBMultiTexturing::GetInstance()->BindMultiTextures( 4, 2 );
glBegin( GL_QUADS );
      glNormal3f( 0.0f,  0.0f, +1.0f );
      CARBMultiTexturing::GetInstance()->MultiTexCoord2fARB( 0.0f, 0.0f );
      glVertex3f(-1.0f, -1.0f,  1.0f );
      CARBMultiTexturing::GetInstance()->MultiTexCoord2fARB( 1.0f, 0.0f );
      glVertex3f( 1.0f, -1.0f,  1.0f );
      CARBMultiTexturing::GetInstance()->MultiTexCoord2fARB( 1.0f, 1.0f );
      glVertex3f( 1.0f,  1.0f,  1.0f );
      CARBMultiTexturing::GetInstance()->MultiTexCoord2fARB( 0.0f, 1.0f );
      glVertex3f(-1.0f,  1.0f,  1.0f );
glEnd();
CARBMultiTexturing::GetInstance()->DisableMultiteksturing();
```

Trochę inaczej wygląda użycie multiteksturingu z wykorzystaniem tablic VertexArrays – patrz także rozdział *23. Wykorzystanie VertexArrays*. Jedyna różnica polega na tym, że na samym początku, przed rysowaniem tablicy i jeszcze przed wywołaniem `CARBMultiTexturing::BindMultiTextures()` należy wywołać poniższą funkcję klasy `CVertexArrays`:

```
GLvoid CVertexArrays::EnableTextureCoordMultiTex( GLsizei iCount = 2 );
```

Parametry:

`iCount` Ilość tekstur jaka zastanie nałożona na rysowany obiekt.

Przykład renderowania multitekstury z wykorzystaniem tablicy – dla metody rysującej `glDrawArrays()` – bez indeksowania:

```
//na obiekt nakładane są dwie tekstury o indeksach 4 i 5
GLint iTexNum = 2; // nakładamy dwie tekstury
CVertexArrays::GetInstance()->EnableTextureCoordMultiTex( iTexNum );
CARBMultiTexturing::GetInstance()->BindMultiTextures( 4, iTexNum );

glDrawArrays( GL_QUADS, 0, 4 );

CARBMultiTexturing::GetInstance()->DisableMultiteksturing();
```

Przykład renderowania multitekstury z wykorzystaniem tablicy – dla metody rysującej `glDrawElements()` – z indeksowaniem:

```
//na obiekt nakładane są dwie tekstury o indeksach 4 i 5
CVertexArrays *pVA = CVertexArrays::GetInstance();

GLint iTexNum = 2; // nakładamy dwie tekstury

pVA->EnableTextureCoordsMultiTexElem( iTexNum );
CARBMultiTexturing::GetInstance()->BindMultiTextures( 4, iTexNum );

pVA->LockArrays( 0, pVA->GetVerticesElemSize() );

glDrawElements( GL_TRIANGLES,
                pVA->GetSubIndicesSize( iIndex ),
                GL_UNSIGNED_SHORT,
                pVA->GetSubIndices( iIndex ));

pVA->UnlockArrays();
```

```
pVA->DisableClientState();
```

9.2. Tekstury Emboss Bump Mapping

Do wczytywania specjalnej tekstury dla uzyskania efektu mapowania wybojów, służy poniższa metoda:

```
GLint CTextureLoader::LoadEmbossBump( LPCTSTR lpFileName );
```

Parametry:

`lpFileName` Nazwa pliku, którą chcemy uczynić teksturą. Musi to być specjalna tekstura tzw. mapa wysokości, gdzie ciemne obszary oznaczają elementy niżej położone, a jaśniejsze wyżej.

Wartość zwracana:
Indeks nowej tekstury. W przypadku błędu zwracana jest wartość -1.

Należy pamiętać, że powyższa metoda z jednego pliku tworzy dwie tekstuty, a więc powstają dwa indeksy tekstur. W programie, przy wykorzytywaniu tekstury bump, używać będziemy tylko indeksu pierwszego! To właśnie ten pierwszy indeks jest zwracany przez tą funkcję.

10. Dźwięk i muzyka

Oczywiście żadna gra nie obędzie się bez dźwięku i muzyki dlatego też OpenGLFramework zawiera taką funkcjonalność. Do tego celu może być wykorzystywana jedna z dwóch bibliotek dźwiękowych: `OpenAL` oraz `FMOD`. Przy wyborze biblioteki musimy pamiętać, że `FMOD` jest biblioteką darmową tylko dla projektów niekomercyjnych. Aby użyć biblioteki `OpenAL`, to w pliku *define.h* musimy zdefiniować makro `_USE_SOUND_OPENAL_`. Jeżeli nie zdefiniujemy tego makra to domyślnie użyta zostanie biblioteka `FMOD`. Patrz także rozdział *7. Plik define.h*. Domyślnie OpenGLFramework używa biblioteki `OpenAL`.

10.1. FMOD

Obsługą biblioteki FMOD zajmuje się klasa `CSoundFMOD` zawarta w projekcie `Sound`. Aby wczytać pliki z muzyką/dźwiękiem należy dokonać odpowiednie wpisy do pliku *SoundFMOD.cpp*. Mianowicie wewnątrz funkcji `CSoundFMOD::LoadAllSounds()` należy wywołać odpowiednie metody zależnie od tego jaki rodzaj pliku chcemy wczytać. Dla małych plików dźwiękowych, np. formatu `WAV` wywołujemy:

```
GLboolean CSoundFMOD::LoadSample( const char* lpSample,
                                  GLboolean bLoopSound = GL_FALSE );
```

Dla większych plików, np. MP3 wywołujemy:

```
GLboolean CSoundFMOD::LoadStreamSample( const char* lpSample,
                                        GLboolean bLoopSound = GL_FALSE );
```

Parametry:

`lpSample` Nazwa pliku dźwiękowego wraz ze ścieżką.

`bLoopSound` Jeżeli ustawimy na `GL_TRUE` to wówczas dźwięk odrywany będzie w pętli aż do wyłączenia. Jeżeli ustawimy na `GL_FALSE` (wartość domyślna) to dźwięk zostanie odegrany tylko jeden raz.

Wartość zwracana:
`GL_TRUE` jeżeli plik zostanie wczytany prawidłowo.

Następnie aby odegrać wcześniej wczytany plik, należy w odpowiednim momencie wywołać metodę `Play...()`. I tak dla sampli będzie to:

```
GLvoid CSoundFMOD::PlaySample( GLint iIndex );
```

oraz dla większych plików:

```
GLvoid CSoundFMOD::PlayStreamSample( GLint iIndex );
```

Parametry:
`iIndex` Numer pliku dźwiękowego, licząc od zera. Kolejność plików ustalana jest wedle kolejności ich wczytywania przez metody `LoadSample()` i `LoadStreamSample()`.

Oczywiście możemy także zatrzymać odgrywanie jakiegoś dźwięku, poprzez metody:

```
GLvoid CSoundFMOD::StopSample( GLint iIndex );
GLvoid CSoundFMOD::StopStreamSample( GLint iIndex );
```

Parametry:
`iIndex` Numer pliku dźwiękowego, licząc od zera. Kolejność plików ustalana jest wedle kolejności ich wczytywania przez metody `LoadSample()` i `LoadStreamSample()`.

Przykład:

```
GLboolean CSoundFMOD::LoadAllSounds()
{
      ...
      LoadSample( "sounds/sample1.wav" );       // index 0
      LoadSample( "sounds/sample2.wav" );       // index 1
      LoadStreamSample( "sounds/music.mp3" );   // index 2
      ...
      return GL_TRUE;
}

      ...
      // gdzieś w kodzie odgrywamy sample2.wav:
      CGameControl::GetInstance()->GetSoundFMOD()->PlaySample( 1 );
      ...
      // gdzieś w kodzie odgrywamy music.mp3:
      CGameControl::GetInstance()->GetSoundFMOD()->PlayStreamSample( 2 );
```

10.2. OpenAL

Za obsługę biblioteki `OpenAL`, a konkretnie odgrywania pojedynczego pliku muzycznego, odpowiada klasa `CSoundOpenAL`. Natomiast tą klasę obsłużymy za pomocą singletonu:

COpenALManager. Obie te klasy zawarte są w projekcie Sound. Aby móc odegrać jakikolwiek dźwięk za pomocą tej biblioteki to musimy, najlepiej w metodzie COpenALManager::LoadAllSounds() wczytać dany plik. Uczynimy to za pomocą poniższej metody:

```
ALint COpenALManager::LoadSound( const char* lpFileName,
                                 ALboolean bLoop = AL_FALSE,
                                 ALboolean bStream = AL_FALSE );
```

Parametry:

lpFileName — Nazwa pliku muzycznego, który chcemy załadować. Metoda ta umożliwia wczytywanie i późniejsze odgrywanie jedynie plików WAV oraz OGG.

bLoop — Jeżeli ustawimy na AL_TRUE to plik muzyczny będzie nieustanie odgrywany w pętli.

bStream — Ustawiając na AL_TRUE wskazujemy, że chcemy aby plik OGG był wczytywany i odgrywany strumieniowo. Jest to niezbędne dla dużych plików muzycznych mających po kilka megabajtów. Dla plików WAV opcja ta jest nieistotna gdyż dla tych plików streaming zawsze będzie wyłączony.

Wartość zwracana:
Funkcja zwróci indeks dla danego pliku muzycznego.

Uwagi:
Jeżeli w naszym programie zaistnieje sytuacja, że chcemy aby ten sam dźwięk był słyszany jednocześnie po kilka razy, np. mamy dźwięk eksplozji i wiemy, że w grze może jednocześnie eksplodować kilka rzeczy na raz, to ten sam dźwięk musimy wczytać odpowiednio wiele razy. Jest to niezbędne gdyż, kilkakrotne wywołanie odegrania tego samego dźwięku w krótkich odstępach czasu, spowoduje zatrzymanie aktualnego odgrywania.

Przykłada wczytania pliku dźwiękowego i muzycznego:

```
ALboolean COpenALManager::LoadAllSounds()
{
      LoadSound( "sounds/dzwiek.wav" );                          // index 0
      LoadSound( "sounds/muzyka.ogg", AL_TRUE, AL_TRUE );        // index 1

      return AL_TRUE;
}
```

Po wczytaniu pliku możemy wywołać szereg poniższych metod, klasy COpenALManager:

```
ALboolean COpenALManager::Play( ALint iIndex );
ALboolean COpenALManager::Pause( ALint iIndex );
ALboolean COpenALManager::Stop( ALint iIndex );
```

Parametry:

iIndex — Indeks pliku dźwiękowego/muzycznego liczony od zera. Kolejność plików ustalana jest wedle kolejności ich wczytywania przez metodę COpenALManager::LoadSound().

Wartość zwracana:
AL_TRUE w przypadku podania odpowiedniego indeksu.

10.2.1. Dźwięk przestrzenny w OpenAL

Poza podstawowymi operacjami na plikach dźwiękowych, możemy dodatkowo ustalić z jakiego kierunku dochodzi dźwięk względem słuchacza. Da nam to efekt dźwięku przestrzennego. Tak więc na początek możemy ustalić pozycję słuchacza za pomocą poniższej metody:

```
ALvoid COpenALManager::SetListenerPosition( const CVector3 &in_cVec );
```

Parametry:

`in_cVec` Referencja na klasę `CVector3` zawierającej nowe współrzędne położenia słuchacza w przestrzeni 3D.

Domyślnie słuchać jest umiejscowiony w punkcie (0, 0, 0).

Poza zmianą położenia słuchacza, możemy także zmieniać położenia źródła dźwięku, za pomocą poniższych metod:

```
ALboolean COpenALManager::Move( ALint iIndex,
                                ALfloat fX, ALfloat fY, ALfloat fZ );
ALboolean COpenALManager::Move( ALint iIndex, const CVector3 &in_cVec );
```

Parametry:

`iIndex` Indeks pliku dźwiękowego/muzycznego liczony od zera. Kolejność plików ustalana jest wedle kolejności ich wczytywania przez metodę `COpenALManager::LoadSound()`.

`fX`, `fY`, `fZ` Nowe współrzędne źródła dźwięku w przestrzeni 3D.

`in_cVec` Referencja na klasę `CVector3` zawierającej nowe współrzędne położenia źródła dźwięku w przestrzeni 3D.

Wartość zwracana:

`AL_TRUE` w przypadku podania odpowiedniego indeksu.

10.2.2. Regulacja głośności w OpenAL

Klasa `COpenALManager` zawiera także interfejs dający możliwość zmiany głośności. Polega to na modyfikacji parametru „wzmocnienia” dla słuchacza, czyli zmiana głośności będzie dotyczyła wszystkich odgrywanych dźwięków i muzyki.

Możemy pobrać aktualnie ustawioną głośność za pomocą metod:

```
ALfloat COpenALManager::GetListenerGain();
ALint COpenALManager::GetPercentListenerGain();
```

Wartość zwracane:

Pierwsza metoda zwróci liczbę typu rzeczywistego z przedziału od 0.0f do 1.0f, gdzie 0.0f oznacza całkowite wyciszenia a 1.0f maksymalną głośność.
Druga metoda zwróci aktualnie ustawioną głośność w procentach, czyli będzie to liczba od 0 do 100, gdzie 0 oznacza całkowite wyciszenie a 100 maksymalną głośność.

Aby zmienić głośność możemy wywołać jedną z poniższych metod:

```
ALvoid COpenALManager::SetListenerGain( ALfloat fGain );
ALvoid COpenALManager::SetListenerGain( ALint iPecentGain );
```

Parametry:

`fGain` Dla pierwszej metody: aktualne „wzmocnienie" jako liczba z przedziału 0.0f – 1.0f, gdzie 0.0f oznacza całkowite wyciszenia a 1.0f maksymalną głośność.

`iPercentGain` Dla drugiej metody: aktualny procent głośności jako liczba z przedziału 0 – 100, gdzie 0 oznacza całkowite wyciszenie a 100 maksymalną głośność.

10.3. Regulacja głośności

Klasa `CMasterVolume` projektu `Sound` daje nam możliwość kontroli głównej głośności systemu. Sama klasa `CMasterVolume` wywołuje odpowiednie metody klasy `CVolumeOutMaster`, która to już jest autorstwa nijakiego Alexa Chmuta (zestaw klas został pobrany z witryny www.codeprojects.com). Obiekt klasy `CMasterVolume` już jest tworzony w `CGameControl`, więc pozostaje nam tylko z niego korzystać za pomocą poniższych metod. Pierwszy zestaw funkcji pozwala regulować głośnością procentowo (co jest bardzie wygodne). Natomiast drugi zestaw funkcji operuje na pewnej wartość, której domyślny zasięg zawiera się między 0 a 65535, gdzie 0 oznacza wyciszoną głośność a 65535, głośność maksymalną.

Aby ustawić głośność na zadany procent:

```
GLvoid CMasterVolume::SetPercentVolume( GLuint uiPercent = 50 );
```

Parametry:

`uiPercent` Procent na jaki ma być ustawiona głośność. Domyślna wartość wynosi 50%.

Aby pobrać aktualny procent głośności:

```
GLuint CMasterVolume::GetPercentVolume();
```

Wartość zwracana:

Funkcja zwróci liczbę z przedziału od 0 do 100 jako procent aktualnie ustawionej głośności.

Aby zwiększyć głośność o zadany procent:

```
GLvoid CMasterVolume::IncreasePercentVolume( GLuint uiPercent = 10 );
```

Parametry:

`uiPercent` Ilość procent a jaką zostanie zwiększona aktualna głośność.

Aby zmniejszyć głośność o zadany procent:

```
GLvoid CMasterVolume::DecreasePercentVolume( GLuint uiPercent = 10 );
```

Parametry:

`uiPercent` Ilość procent o jaką zostanie zmniejszona aktualna głośność.

Aby ustawić głośność na zadaną wartość:

```
GLvoid CMasterVolume::SetVolume( GLuint uiVolume );
```

Parametry:

`uiVolume` Wartość głośności jaką chcemy ustawić. Domyślny zasięg wynosi od 0 do 65535.

Aby pobtać aktualnie ustawioną wartość głośności:
```
GLuint CMasterVolume::GetVolume();
```

Wartość zwracana:
Funkcja zwróci domyślnie wartość z przedziału od 0 do 65535.

Aby zwiększyć głośność o zadaną wartość:
```
GLvoid CMasterVolume::IncreaseVolume( GLuint uiVolume = 4096 );
```

Parametry:
`uiVolume` Wartość o jaką chcmey zwiększyć głośność. Maksymalny zasięg wynosi 65535.

Aby zmniejszyć głośność o zadaną wartość:
```
GLvoid CMasterVolume::DecreaseVolume( GLuint uiVolume = 4096 );
```

Parametry:
`uiVolume` Wartość o jaką chcemy zmniejszyć głośność. Maksymalny zasięg wynosi 65535.

11. Wyświetlanie tekstu

OpenGLFramework zawiera pięć możliwości wyświetlania tekstu. Każda ma jakieś wady i zalety, jedne lepiej się prezentują inne szybciej się renderują itp. Poniżej dokładnie opisana jest każda z nich. Wszystkie klasy wiązane z wyświetlaniem tekstu znajdują się w projekcie `Fonts`.

11.1. Wyświetlanie tekstu za pomocą `CMyFont`

Aby użyć tej metody należy dołączyć plik *MyFont.h*, stworzyć obiekt tej klasy oraz wywołać funkcję `CMyFont::DrawText()` lub `CMyFont::DrawTextFormat()`:

```
void CMyFont::DrawText( LPCTSTR lpText,
                        GLfloat fX, GLfloat fY, GLfloat fZ,
                        COLORREF crColor = RGB(255, 255, 255),
                        GLfloat fSize = 1.0f, GLfloat fScale = 0.1f );

void CMyFont::DrawTextFormat( GLfloat fX, GLfloat fY, GLfloat fZ,
                              COLORREF crColor,
                              GLfloat fSize, GLfloat fScale,
                              LPCTSTR fmt, ... );
```

Parametry:
`lpText` Tekst jaki chcemy wyświetlić.

`fX, fY, fZ` Pozycja tekstu (tekst rysowany jest w przestrzeni 3D).

`crColor` Składowe koloru tekstu. Podajemy używając makra `RGB()`.

`fSize` Grubości tekstu (linii rysującej tekst).

`fScale` Wielkość / rozmiar tekstu.

Uwagi:
Jest to wymyślona przeze mnie metoda wyświetlania tekstu jeszcze ze starszego framework'a. Wyświetlany tekst rysowany jest za pomocą linii, dlatego też użycie tej metody jest jedną z najwydajniejszych metod, jej główną zaletą miało być także wyświetlanie polskich znaków diakrytycznych. Inną zaletą miło być wyświetlanie tekstu bez stałej szerokości znaku, ale to już zależy co komu pasuje – ja, np. takiego czegoś właśnie potrzebowałem. Nadmienię także, że używałem tych funkcji z powodzeniem w swoich starszych grach, więc metoda jest sprawdzona :). Minusem jest to, iż nie wygląda zbyt ładnie, ale to już kwestia gustu. Aby poprawić wygląd dobrze jest użyć antyaliasingu co najmniej z próbkowaniem x4.
W `CMyFont` brakuje rysowania znaku tyldy (~).
Za pomocą `CMyFont` można wyświetlać polskie znaki diakrytyczne (inne języki – poza angielskim i polskim – nie są wspierane).
Polecam użycie tej metody tylko w przypadku konieczności przyśpieszenia działania aplikacji.
Zalety:
- znaki diakrytyczne,
- bardzo szybkie rysowanie,
- możliwość obracania i przekręcania tekstu – o ile komuś to potrzebne :)

Wady:
- tylko polskie znaki diakrytyczne,
- tylko jeden rodzaj czcionki,
- brak możliwości wykorzystania czcionek systemowych,
- brak znaku tyldy,
- nie działa w trybie 2D.

11.2. Wyświetlanie tekstu za pomocą `CBitmapFont`

Aby użyć wyświetlania tekstu za pomocą klasy `CBitmapFont`, należy dołączyć plik nagłówkowy *BitmapFont.h* oraz stworzyć obiekt tej klasy:

```
CBitmapFont::CBitmapFont( HDC hDC, LPCTSTR lpFontName, GLint iHeight = -24 );
```

Parametry:

`hDC` Kontekst urządzenia GDI (należy pobrać go z klasy `CWindowData`).

`lpFontName` Nazwa czcionki zarejestrowanej w systemie.

`iHeight` Wielkość czcionki (należy podawać w wartościach ujemnych).

Po stworzeniu obiektu `CBitmapFont` możemy rysować tekst za pomocą dwóch funkcji:

```
GLvoid CBitmapFont::DrawText( GLfloat fX, GLfloat fY, LPCTSTR fmt );

GLvoid CBitmapFont::DrawTextFormat( GLfloat fX, GLfloat fY, LPCTSTR fmt, ... );
```

Parametry:

`fX, fY` Pozycja tekstu, gdzie 0, 0 to środek ekranu – tekst jest rysowany "na płaszczyźnie ekranu".

`fmt` Tekst jaki chcemy wyświetlić.

Funkcja `DrawTextFormat()` zawiera także możliwość dodawania opcjonalnych parametrów w celu formatowania tekstu.

Uwagi:
`CBitmapFont` buduje tzw. czcionki bitmapowe (rastrowe), które co prawda ładnie wyglądają (o ile ich za bardzo nie powiększysz), ale bardzo obciążają procesor. Dodatkową zaletą jest także możliwość wykorzystania czcionek zarejestrowanych w systemie oraz możliwość wyświetlania polskich znaków diakrytycznych (bez UNICODE).
Zalety:
- dość ładny wygląd,
- znaki diakrytyczne,
- możliwość wykorzystania czcionek systemowych,
- działa w trybie 2D.

Wady:
- bardzo obciąża procesor.

11.3. Wyświetlanie tekstu za pomocą `COutlineFont`

Aby użyć wyświetlania tekstu za pomocą klasy `COutlineFont`, należy dołączyć plik nagłówkowy *OutlineFont.h* oraz stworzyć obiekt tej klasy:

```
COutlineFont::COutlineFont (HDC hDC, LPCTSTR lpFontName );
```

Parametry:
`hDC` Kontekst urządzenia GDI (należy pobrać go z klasy `CWindowData`).

`lpFontName` Nazwa czcionki zarejestrowanej w systemie.

Po stworzeniu obiektu `COutlineFont` możemy rysować tekst za pomocą dwóch funkcji:

```
GLvoid COutlineFont::DrawText( LPCTSTR fmt );

GLvoid COutlineFont::DrawTextFormat( LPCTSTR fmt, ... );
```

Parametry:
`fmt` Tekst jaki chcemy wyświetlić.

Funkcja `DrawTextFormat()` zawiera także możliwość dodawania opcjonalnych parametrów w celu formatowania tekstu.

Uwagi:
`COutlineFont` buduje tzw. czcionki konturowe, które co prawda ładnie wyglądają, ale bardzo obciążają procesor. Dodatkową zaletą jest także możliwość wykorzystania czcionek zarejestrowanych w systemie oraz możliwość wyświetlania polskich znaków diakrytycznych (bez UNICODE).
Tekst uzyskany z `COutlineFont` jest trójwymiarowy (widzimy jego grubość). Tekst rysowany jest w przestrzeni 3D zatem pozycje tekstu musimy ustalić sobie sami, np. za pomocą funkcji `glTranslatef()` wywoływanej przed rysowaniem tekstu.
Zalety:
- ładny wygląd,

- znaki diakrytyczne,
- tekst 3D – możliwość obracania i przekręcania tekstu,
- możliwość wykorzystania czcionek systemowych.

Wady:
- nie za bardzo działa w trybie 2D,
- bardzo obciąża procesor.

11.4. Wyświetlanie tekstu za pomocą `CSDLFont`

Aby użyć wyświetlania tekstu za pomocą klasy `CSDLFont`, należy dołączyć plik nagłówkowy *SDLFont.h* oraz stworzyć obiekt tej klasy:

```
CSDLFont::CSDLFont( LPCTSTR lpFontFile = "fonts/FreeSansBold.ttf",
                    GLint iSizeFont = 20,
                    GLboolean bUseStorageBuffer = GL_TRUE );
```

Parametry:

`lpFontFile` Nazwa pliku z czcionką jakiej chcemy użyć (domyślnie `"fonts/FreeSansBold.ttf"`). OpenGLFramework zawiera własne darmowe czcionki – umieszczone są one w katalogu *fonts*. Można także wczytać pliki zawarte w katalogu *C:\Windows\Fonts*, należy jednak pamiętać, że musimy podać nazwę pliku a nie nazwę zarejestrowanej czcionki.

`iSizeFont` Wielkość czcionki.

`bUseStorageBuffer` Ustawiając tę flagę na `GL_TRUE`, nakazujemy buforowania wyświetlanych tekstów, dzięki czemu, rendering teksu będzie wykonywany znacznie szybciej. Oznacza to także trochę większe absorbowanie pamięci. Ustawiając na `GL_FALSE`, każde to samo wywołanie rysowania tekstu, będzie renderowane od nowa.

Po stworzeniu obiektu `CSDLFont` możemy rysować tekst za pomocą czterech funkcji. Na początek pierwszy zestaw:

```
GLvoid CSDLFont::DrawText( GLint iPosX, GLint iPosY,
                           COLORREF crColor,
                           LPCTSTR lpText, GLboolean bMidScreen = GL_FALSE );

GLvoid CSDLFont::DrawTextFormat( GLint iPosX, GLint iPosY,
                                 GLboolean bMidScreen, COLORREF crColor,
                                 LPCTSTR fmt, ... );
```

Parametry:

`iPosX, iPosY` Pozycja tekstu względem ekranu. Punkt 0x0 znajduje się w lewym górnym rogu ekranu (tekst jest rysowany "na płaszczyźnie ekranu"). Ważne jest aby rysować tekst tak jakby miał być rozmieszczony dla ekranu 1024x768 – wówczas będziemy mieli prawidłowe skalowanie tekstu względem rozdzielczości. Oznacza to, że jeżeli stosujemy rozdzielczość, np. 800x600 i chcemy wyświetlić tekst na samym spodzie okna, to nie ustawiamy `iPosY` na 600 tylko na 768!

`bMidScreen` Czy tekst ma być wyrównany do środka. Jeżeli ustawimy tę wartość na `GL_TRUE` to

tekst będzie wyrównany względem środka ekranu. Oznacza to, że wartość `iPosX` nie będzie brana pod uwagę, ponieważ program sam ją wyliczy. Gdy podamy `GL_FALSE` (wartość domyślna), wówczas tekst będzie pozycjonowany na podstawie `iPosX`.

`crColor` Składowe koloru tekstu. Należy użyć makra `RGB()`.

`lpText, fmt` Tekst jaki chcemy wyświetlić.

Funkcja `DrawTextFormat()` zawiera także możliwość dodawania opcjonalnych parametrów w celu formatowania tekstu.

Oraz drugi zestaw:

```
GLvoid DrawText( const RECT &rcRectangle, COLORREF crColor, LPCTSTR lpText );

GLvoid DrawTextFormat( const RECT &rcRectangle, COLORREF crColor,
                       LPCTSTR fmt, ... );
```

Parametry:

`rcRectangle` Prostokąt opisujący położenie tekstu. Pozycja tekstu zostanie wyliczona tak aby tekst narysował się dokładanie na środku prostokąta. Ważne jest aby pola struktury `RECT` zawierały pozycję prostokąta w pikselach dla rozdzielczości ekranu 1024x768 – wówczas będziemy mieli prawidłowe skalowanie tekstu względem rozdzielczości.

`crColor` Składowe koloru tekstu. Należy użyć makra `RGB()`.

`lpText, fmt` Tekst jaki chcemy wyświetlić.

Funkcja `DrawTextFormat()` zawiera także możliwość dodawania opcjonalnych parametrów w celu formatowania tekstu.

Użycie StorageBuffer

Aby zoptymalizować rendering tekstu, każdy obiekt klasy `CSDLFont` zwiera zmienną `m_bUseStorageBuffer` (ustawianą przez konstruktor) mówiącą o tym czy należy zapamiętywać wcześniej wyrenderowane teksty. Jeżeli StorageBuffer jest w użyciu to przy każdym wywołaniu metod `DrawText()` i `DrawTextFormat()` najpierw sprawdzane jest czy podany tekst nie został już wcześniej wyrenderowany (sprawdzany jest tekst i jego kolor). Jeżeli nie, to podany tekst zostanie wyrenderowany na nowo oraz zapisany do bufora. Czyli następnym razem przy kolejnym wywołaniu `DrawText()` lub `DrawTextFormat()` tekst (o ile jest identyczny) nie będzie musiał być renderowany od nowa tylko użyta zostanie wcześniej przygotowana i buforowana tekstura.
Jeżeli StorageBuffer nie jest używany to każde wywołanie metod `DrawText()` i `DrawTextFormat()` powoduje renderowanie tekstu od nowa co trwa znacznie dłużej.
Aby zminimalizować zużycie pamięci, podczas używania StorageBuffer, każdemu buforowanemu tekstowi przypisywany jest czas ostatniego odwołania się do niego. Następnie co dany kwartał czasu (domyślnie co 30 sekund) usuwane są z bufora stare teksty, których czas ostatniego wykorzystania przekracza określony limit (domyślnie 30 sekund).
Dodatkowo, każdy obiekt klasy `CSDLFont` zawiera limit na buforowanie tekstów – domyślnie obiekt może buforować maksymalnie 200 tekstów.
Włączyć i wyłączyć StorageBuffer możemy nie tylko poprzez konstruktor, ale także w dowolnej

chwili „życia” obiektu:

```
GLvoid CSDLFont::SetStorageBuffer( GLboolean bParam );
```

Parametry:
`bParam` Podając `GL_TRUE` uaktywniamy StorageBuffer, podając `GL_FALSE` wyłączamy go.

Za pomocą poniższej funkcji możemy sprawdzić czy StorageBuffer jest aktywny.

```
GLboolean CSDLFont::IsStorageBuffer();
```

Wartość zwracana:
`GL_TRUE` jeżeli aktywny, w przeciwnym wypadku `GL_FALSE`.

Uwagi:
`CSDLFont` buduje tekst za pomocą biblioteki `SDL_ttf`. Tekst taki wygląda bardzo dobrze oraz nie obciążają procesora tak bardzo jak fonty bitmapowe czy konturowe, a z użyciem StorageBuffer działa jeszcze szybciej. Aby umożliwić wyświetlanie polskich znaków należy zbudować aplikację w UNICODE (domyślnie OpenGLFramework używa już UNICODE). Ze względu na szybkość oraz ładny wygląd zalecam stosowanie tej klasy.
Zalety:
- względnie szybkie rysowanie,
- ładny wygląd,
- znaki diakrytyczne,
- możliwość wyrównania tekstu do środka ekranu,
- działa w trybie 2D.

Wady:
- znaki diakrytyczne tylko z UNICODE.

11.5. Wyświetlanie tekstu za pomocą `CTextureFont`

Aby użyć wyświetlania tekstu za pomocą klasy `CTextureFont`, należy dołączyć plik nagłówkowy *TextureFont.h* oraz stworzyć obiekt tej klasy:

```
CTextureFont::CTextureFont();
```

Po stworzeniu obiektu `CTextureFont` możemy rysować tekst za pomocą dwóch funkcji:

```
GLvoid CTextureFont::DrawText( GLfloat fPosX, GLfloat fPosY,
                               LPCTSTR fmt, GLint iFont,
                               GLfloat r = 1.0f, GLfloat g = 1.0f,
                               GLfloat b = 1.0f, GLfloat fScale = 0.00002f );

GLvoid CTextureFont::DrawTextFormat( GLfloat fPosX, GLfloat fPosY,
                                     GLint iFont,
                                     GLfloat r, GLfloat g, GLfloat b,
                                     GLfloat fScale, LPCTSTR fmt, ... );
```

Parametry:
`fPosX, fPosY` Pozycja tekstu w przestrzeni 3D.

`r, g, b` Składowe koloru tekstu.

`fmt` Tekst jaki chcemy wyświetlić.

`iFont` Rodzaj czcionki, do wyboru 0 i 1.

`fScale` Wielkość wyświetlanej linijki tekstu.

Funkcja `DrawTextFormat()` zawiera także możliwość dodawania opcjonalnych parametrów w celu formatowania tekstu.

Uwagi:
`CTextureFont` buduje tekst za pomocą tekstury zawierającej narysowane litery. Tekst taki nie obciąża zbytnio procesora tak bardzo jak fonty bitmapowe, konturowe czy też `CSDLFont`. Za pomocą tej metody nie możemy wyświetlać znaków narodowych oraz znaki zawsze rysowane są ze stałą szerokością (jak dla mnie nie za dobrze to wygląda – odstępy miedzy literami są zbyt duże). Polecam używanie tej metody jeżeli zależy nam na szybkość.
Zalety:
- szybkie rysowanie,
- działa w trybie 2D.

Wady:
- brak znaków diakrytycznych,
- brak możliwość wykorzystania czcionek systemowych,

12. Obliczanie FPS

Do obliczania ilość wyświetlanych klatek na sekundę służy klasa `CFps` projektu `Fps`. Ogólnie obsługa obliczania i wyświetlania FPS jest już zaimplementowana i nie musimy w nią ingerować. Wymagany jest jedynie timer odliczający czas co 1000 ms – domyślnie jest już ustawiony. Aby wyświetlić informacje o FPS musimy najpierw uaktywnić jego wyświetlanie poprzez metodę `ShowFps()`. Następnie w renderingu musimy wywołać funkcję `DrawFps()` – domyślnie jest już wywoływana. Jednakże najlepszym sposobem na uzyskanie informacji o FPS jest nieingerowanie w kod a jedynie skorzystanie z konsoli – patrz dział *14. Konsola*.

13. Kontrola prędkość animacji

Często (a właściwie zawsze) potrzebujemy niezależności sprzętowej jeżeli chodzi o szybkość jakiejś animacji. Po prostu wymagane jest aby np. jakiś obiekt przesuwał się po ekranie z tą samą prędkością i nie ważne jest czy program uruchomimy na Pentium II czy IV. Aby to osiągnąć musimy skorzystać z klasy `CSpeedControl` znajdującej się w projekcie `SpeedControl`. Ogólnie wszystko jest już zaimplementowane – naszym zadaniem jest tylko wywoływanie metody:

```
GLfloat CSpeedControl::GetMultiplier();
```

Wartość zwracana:
Funkcja ta zwróci nam wartość `float` i jest to wartość jaką musimy mnożyć przez wszelkie zmienne wykorzystywane do animacji obiektów, tj. przesuwania, obracania, itp.

Jeżeli nie odpowiada nam szybkość zwracana przez `CSpeedControl::GetMultiplier()`, możemy w konstruktorze klasy `CSpeedControl` spróbować zmodyfikować wartość przypisywaną do

zmiennej `CSpeedControl::m_fRegulationSpeed`. Jeżeli zmniejszymy przypisywaną wartość dla tej zmiennej, wówczas animacja będzie wolniejsza.

Musimy pamiętać aby każdy ruch, czy obrót jakichkolwiek obiektów mnożyć przez `CSpeedControl::GetMultiplier()`. Jeżeli będziemy tak postępować to w prosty sposób uzyskamy możliwość spauzowania gry. Wystarczy, że wywołamy poniższą metodę:

```
GLvoid CSpeedControl::SetPause( GLboolean bPause );
```

Parametry:

`bPause` Podając `GL_TRUE` uaktywnimy pauzę, podając `GL_FALSE` odblokujemy pauzę.

Jeżeli uaktywnimy pauzę to funkcja `CSpeedControl::GetMultiplier()` będzie zwracała `0.0f` i jeżeli wartość ta będzie mnożona przez wszelkie zmienne związane z ruchem i obrotem obiektów to w oczywisty sposób je unieruchomimy dając efekt spauzowania gry.

Klasa `CSpeedControl` dostarcza także metody umożliwiające kontrolowanie szybkość tworzonej przez nas animacji. Np. robimy grę 2D gdzie musimy podmieniać tekstury sprite'a obrazujące, np. chód żołnierza i chcemy aby te tekstury podmieniane były, np. co 200 milisekund. Gwarantuje nam to stałą prędkość animacji – niezależnie od danego komputera oraz mamy możliwość kontrolowania szybkości takiej animacji.
A więc do dzieła. Najpierw, najlepiej w konstruktorze `CSpeedControl`, musimy wywołać funkcję `CSpeedControl::CreateAnimationControl()`. Funkcja ta utworzy dla nas zmienną pamiętającą czas (liczbę milisekund) ostatniego wywołania. Jest to niezbędne do kontroli naszej animacji. Jeżeli potrzebujemy kontrolować więcej animacji, np. pięć to musimy po prostu pięć razy wywołać `CreateAnimationControl()`.
Następnie w funkcji renderującej należy wywołać:

```
GLuint CSpeedControl::CheckAnimationTime( unsigned long ulMs, GLint iIndex,
                                          GLboolean bUpdateIfPause = GL_TRUE );
```

Parametry:

`ulMs` Liczba milisekund, czyli tutaj określamy szybkość naszej animacji.

`iIndex` Indeks (liczony od 0) zwrócony przez `CreateAnimationControl()`. Indeksy przypisane są danym animacją. Przypisanie to jest dowolne i określane przez nas, np. ustalamy sobie, że indeks 0 będzie odnosił się do animacji żołnierza, indeks 1 do animacji lotu pociski, itd.

`bUpdateIfPause` Jeżeli ustawimy tę wartość na `GL_TRUE` (i tak jest domyślnie) to jeżeli aktywna będzie pauza to czas szybkości naszej animacji zostanie odświeżony aktualnym czasem. Jest to wymagane w przypadkach gdy nie chcemy aby po anulowaniu pauzy dana animacja od razu ruszyła (znaczy aby `CheckAnimationTime()` od razu zwrócił `GL_TRUE`) a stanie się tak jeżeli czas pauza potrwa dłużej niż podana `ulMs`. Dzięki ustawieniu tej wartości na `GL_TRUE` pauza nie będzie miała wpływu na czas – czas jakby stanie w miejscu.

Wartość zwracana:
0 – jeżeli jeszcze nie upłynęła podana liczba milisekund. Liczba większa od zera – jeżeli upłynęła podana liczba milisekund. Oznacza to, że możemy wykonać daną animację, np. podmienić tekstury.

Zwracana liczba większa od 0 to aktualny czas animacji jaki upłynął.

Przykład:

```
GLint g_iIndexAnim1 = 0;
GLint g_iIndexAnim2 = 0;

CSpeedControl::CSpeedControl(...)
{
      ...
      g_iIndexAnim1 = CreateAnimationControl(); //animacja o indeksie 0
      g_iIndexAnim2 = CreateAnimationControl(); //animacja o indeksie 1
}

GLboolean CGameControl::Draw()
{
      ...
      if( GetSpeedCtrl()->CheckAnimationTime( 200, g_iIndexAnim1 ) )
      {
            // realizujemy naszą pierwszą animację
            // animacja o indeksie 0 wywoływana
            // będzie co 200 milisekund
      }
      ...
      if( GetSpeedCtrl()->CheckAnimationTime( 350, g_iIndexAnim2 ) )
      {
            // realizujemy naszą drugą animację
            // animacja o indeksie 1 wywoływana
            // będzie co 350 milisekund
      }
}
```

Jeżeli nie zawsze w każdej klatce będziemy mogli wywołać `CheckAnimationTime()` bo, np. uzależnione to będzie od jakiejś instrukcji warunkowej, to jeżeli po długim okresie czasu, np. dwóch sekundach wywołamy, np. `CheckAnimationTime( 500, ... )` to funkcja ta zawsze zwróci wynik większy od 0. Jeżeli jednak chcemy aby odczekał te pół sekundy to w miejscu tegoż warunku tworzymy `else` i tam wywołujemy poniższą metodę:

```
GLvoid CSpeedControl::UpdateAnimationTime( GLint iIndex );
```

Parametry:

`iIndex` Indeks (liczony od 0) zmiennej stworzonej przez `CreateAnimationControl()`. Indeksy przypisane są danym animacją. Przypisanie to jest dowolne i określane przez nas, np. ustalamy sobie, że indeks 0 będzie odnosił się do animacji żołnierza, indeks 1 do animacji lotu pociski, itd.

Przykład użycia `UpdateAnimationTime()`:

```
...
GLint g_iIndex = CreateAnimationControl();
...

if( bCondition )
{
      if( CheckAnimationTime( 500, g_iIndex ) )
      {
            //podmieniamy animację
```

```
        }
    }
    else
    {
        //nie mogliśmy wywołać animacji, ale chcemy/musimy
        //uaktualnić czas dla naszej animacji:

        UpdateAnimationTime( g_iIndex );
    }
```

14. Konsola

W konsoli możemy wyświetlać wszelkie komunikaty pochodzące z programu, np. wczytywanie zasobów, informacje o błędach, czy informacje o wykonaniu (bądź nie) jakiejś czynności. Dzięki temu konsola umożliwia nam (w fazie projektowania i testowania) na bieżąco, tj. w trakcie działania programu, sprawdzanie wielu istotnych dla nas informacji. Poza tym konsola wyposażona jest we własny wiersz poleceń dzięki czemu możemy w trakcie działania programu uruchamiać różne funkcje, np. wpisując *fps* wyświetlamy uzyskiwaną ilość klatek na sekundę. Ale po kolei.
Aby w ogóle nasz program obsługiwał konsolę, należy uruchomić aplikację z parametrem */console*. Samą konsolę wyświetlamy i chowamy klawiszem [~].
Za obsługę konsoli odpowiada klasa `CConsole` znajdująca się w projekcie `OpenGLFramework`. Jej obiekt, jak i cała jej obsługa, jest już domyślnie stworzona. Możemy natomiast w dowolnym momencie programu wysyłać do konsoli tekst poprzez metody:

```
GLvoid CConsole::AddText( EConsoleMsgType eType, const char *fmt, ... );
GLvoid CConsole::AddText( const char *fmt, ... );
```

Oraz dla UNICODE:

```
GLvoid CConsole::AddText( EConsoleMsgType eType, const wchar_t *fmt, ... );
GLvoid CConsole::AddText( const wchar_t *fmt, ... );
```

Parametry:

`eType` Typ komunikatu jaki przekazujemy. Do wyboru:
- `EConsoleNormal` – normalny (tekst na biało),
- `EConsoleError` – błąd (tekst na czerwono),
- `EConsoleSuccess` – powodzenie (tekst na zielono),
- `EConsoleWarning` – ostrzeżenie (tekst na żółto).

Jeżeli funkcja nie zawiera tego parametru to automatycznie używa `EConsoleNormal`.

`fmt` Tekst jaki chcemy wyświetlić w konsoli.

`...` Ewentualne parametry dla formatowania.

Jeżeli chcemy możemy także dodać do konsoli własne komendy (wedle naszych potrzeb). Aby to uczynić należy w pliku *Console.cpp* zainteresować się funkcją `ExecuteCommand()`. To właśnie w tej funkcji obsługiwane są wprowadzane komendy. Aby wyświetli zawartą już listę komend należy w konsoli wpisać komendę *help*.

15. Obsługa UNICODE

OpenGLFramework wspiera możliwość budowania aplikacji w UNICODE i domyślnie jest on używany. Prawdę mówiąc potrzebowałem UNICODE jedynie do obsługi polskich znaków w `CSDLFont`, ale opłaciło się – mamy dość wydajne i ładnie wyglądające fonty.
Aby aplikacja wykorzystywała UNICODE, po prostu musi być zdefiniowane makro "UNICODE".
Dla VC++ 2005 Express Edition, makro UNICODE definiujemy poprzez właściwości projektu. Z menu „Project" wybieramy „Properties". Następnie przechodzimy do zakładki „Configuration Properties" -> „General" -> „Character Set".
Jeżeli korzystamy z opcji wielojęzyczności (patrz rozdział *16. Wielojęzyczność*) to należy także pamiętać, że pliki z tłumaczeniami także muszą być zapisane w UNICODE. Inaczej znaki narodowe danego kraju mogą nie być wyświetlane prawidłowo. Jednakże jeżeli nie zamierzamy korzystać z klasy `CSDLFont` i/lub nasz program wykorzystywać będzie jedynie język angielski to odradzam używania UNICODE. Powodem jest większe obciążenie procesora dla wyświetlania tekstu w UNICODE.

16. Wielojęzyczność

OpenGLFramework daje także możliwość obsługi wielojęzyczności w naszych grach. Aby tak się stało to po pierwsze należy przygotować odpowiednie pliki z tłumaczeniami tekstów na języki jakie chcemy wspierać. Pliki te umieszczamy w katalogu *languages* i zapisujemy je z rozszerzeniem *lng*. Jeżeli chcemy wykorzystać klasę `CSDLFont` do wyświetlania tekstów, to pliki językowe najlepiej jest zapisać w kodowaniu UNICODE[1]. Wtedy znaki narodowe danego kraju będą wyświetlane poprawnie.
Struktura takiego pliku musi być następująca:

- każda linijka tekstu traktowana jest jako jeden ciąg znaków, czyli przejście do nowej linii oznacza inny string,
- linijka rozpoczynająca się od znaku średnika jest ignorowana – traktowana jako komentarz,
- jeżeli w linijce tekstu zawarty będzie średnik, to cały tekst za średnikiem potraktowany będzie jako komentarz,
- linijka tekstu nie może przekraczać 255 znaków (razem z komentarzem),
- w tekście możemy zawierać typowe znaki formatowania – jak dla funkcji `printf` z `C`.

Kolejny plik jaki musimy przygotować (a konkretnie zmodyfikować wedle własnych potrzeb, gdyż plik ten już istnieje) jest *languages_def.xml* w katalogu *languages*. W pliku tym definiujemy języki jakie będą wspierane przez program. Możemy tu danemu językowi nadać nazwę oraz podać listę czcionek jakie będą wykorzystane do rysowania tekstu. Tak naprawdę plik *languages_def.xml* jest istotny tylko przy wykorzystaniu klasy `CSDLFont` gdyż wskazuje nam czcionki jakie mają być wykorzystane przez tą klasę. Oczywiście przy wspieraniu różnych języków musimy posiadać odpowiednie czcionki (pliki *ttf*), które będą zawierały odpowiednie znaki narodowe.
Standardowo plik *languages_def.xml* wygląda następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<languages>
      <language name="English" file="languages/english.lng">
            <font file="fonts/FreeSansBold.ttf"/>
            <font file="fonts/FreeSans.ttf"/>
            <font file="fonts/FreeMonoBold.ttf"/>
      </language>
```

1 Aby zapisać plik tekstowy w UNICODE – w notatniku mamy taką możliwość w oknie zapisywania pliku.

```
      <language name="Polski" file="languages/polski.lng">
            <font file="fonts/FreeSansBold.ttf"/>
            <font file="fonts/FreeSans.ttf"/>
            <font file="fonts/FreeMonoBold.ttf"/>
      </language>
</languages>
```

Tak, więc pliku *languages_def.xml* musi zawierać główny znacznik `<languages>`. W znaczniku głównym definiujemy wsparcie dla kolejnych języków poprzez znacznik `<language>`. Znacznik `<language>` musi zawierać parametry `name` oraz `file`. Parametrowi `name` musimy przypisać jakąś nazwę, najlepiej po prostu wpisać jaki to jest język. Natomiast parametrowi `file` musimy przypisać ścieżkę do pliku *lng* zawierającego tłumaczenia tekstów na dany język. Struktura plików *lng* opisana została powyżej, w tym rozdziale.
Następnie w znaczniku `<language>` możemy umieścić dowolną ilość znaczników `<font>` aby wskazać jakie czcionki mają być wykorzystane dla danego języka. Znacznik `<font>` musi zawierać parametr `file` z wartością zawierającą ścieżkę do pliku *ttf*. Należy także pamiętać, że ważna jest kolejność podawania znaczników <font> wraz z ich plikami *ttf* bowiem pierwszy plik *ttf* w języku, np. angielskim, po zmianie języka na, np. polski będzie odpowiadał pierwszemu plikowi *ttf* dla języka polskiego. Oznacza to także, że ilość znaczników `<font>` w poszczególnych znacznikach `<language>` musi być identyczna.
Domyślnie ustawionym językiem w programie będzie ten, który będzie zdefiniowany w pliku *languages_def.xml* jako pierwszy.

Do obsługi wielojęzyczności wykorzystywana jest klasa `CMultiLanguage` zawarta w projekcie `MultiLanguage`. Zadanie tej klasy polega jedynie na wczytaniu (na podstawie *languages_def.xml*) pliku z tekstami a następnie zwracania odpowiednich stringów.

Aby wczytać dany plik *lng* należy wywołać metodę:

```
GLboolean CMultiLanguage::LoadLanguage( LPCTSTR lpFileName );
```

Parametry:
`lpFileName` Nazwa pliku wraz ze ścieżką, przeważnie *languages/jakis_plik.lng*.

Wartość zwracana:
`GL_TRUE` w przypadku powodzenie wczytania pliku.

Jednak wywołanie metody `CMultiLanguage::LoadLanguage()` jest już zaimplementowane i jeżeli nie chcemy ręcznie wczytywać jakiegoś pliku *lng* (z pominięciem *languages_def.xml*) to nie musimy se nią zaprzątać głowy.

Aby pobrać nazwę języka, tj. wartość parametru `name` znacznika `<language>`, należy wywołać metodę:

```
LPTSTR CMultiLanguage::GetXMLLngName();
```

Wartość zwracana:
Nazwa aktualnie wczytanego języka pobrana z parametru `name` znacznika `<language>`.

Aby pobrać nazwę pliku *lng* dla aktualnie wczytanego języka, tj. pobrać wartość parametru `file` znacznika `<language>`, należy wywołać metodę:

```
LPTSTR CMultiLanguage::GetXMLLngFileLNG();
```

Wartość zwracana:
Nazwa pliku *lng* aktualnie wczytanego języka pobrana z parametru `file` znacznika `<language>`.

Aby pobrać nazwę pliku *ttf* dla aktualnie wczytanego języka, należy wywołać poniższą metodę:

```
char* CMultiLanguage::GetXMLLngFileTTF( GLint iIndexTTF );
```

Parametry:
`iIndexTTF` Indeks pliku *ttf* zawartego w pliku *languages_def.xml* w znaczniku `<font>`. Indeks jest wartością umowną liczoną od zera, pobieraną w kolejności podawania znaczników `<font>`. Np. z powyższego przykładu zawartości pliku *languages_def.xml* przyjęte jest, że czcionka `fonts/FreeSansBold.ttf` ma indeks 0, czcionka `fonts/FreeSans.ttf` ma index 1, itd.

Wartość zwracana:
Wskazana nazwa pliku ttf, dla aktualnie wczytanego języka. W przypadku błędu pusty string.

Uwagi:
Powyższa metoda `CMultiLanguage::GetXMLLngFileTTF()` jest bardzo istotna przy używaniu klasy `CSDLFont`, gdyż właśnie do konstruktora tej klasy należy podać plik *ttf*, za pomocą którego chcemy rysować tekst.

Następnie jeżeli chcemy wyświetlić dany ciąg znaków należy wywołać funkcję:

```
LPTSTR CMultiLanguage::GetLangText( GLint iIndex );
```

Parametry:
`iIndex` Numer linijki tekstu z pliku *lng*, licząc od zera.

Jeżeli wprowadzimy błędny indeks (poza zakresem) to zamiast oczekiwanego tekstu ujrzymy: „`Error: GetLanText(X)`”, gdzie zamiast „`X`” podany będzie nasz błędny indeks. Dzięki temu łatwo odnajdziemy błąd.
Kolejna rzecz, o której należy pamiętać to to, że obiekt klasy `CMultiLanguage` jest tworzony w `CWindowData` i należy dowoływać się do metod klasy `CMultiLanguage`, np. tak:
`CWindowData::GetInstance()->GetMultiLang()->GetLangText( 0 );`

17. Oświetlenie

Do wykorzystania oświetlenia oraz materiałów należy skorzystać z klasy `CLighting`. Zanim użyjemy światła musimy najpierw ustawić jego parametry. I tak mamy możliwość ustawienia światła otoczenia (ambiend), światła rozproszonego (diffuse), światło odbłysków (specular) oraz pozycje światła w układzie współrzędnych. Możemy także ustawić wartości ambiend, diffuse, specular i shinnese (połysk) dla materiałów. Poniżej opis wszystkich funkcji:

```
GLvoid CLighting::SetAmbient( GLuint uiLight,
                              GLfloat r = 0.5f, GLfloat g = 0.5f,
                              GLfloat b = 0.5f, GLfloat a = 1.0f );
```

Parametry:

`uiLight` Identyfikator światła, należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

`r, g, b, a` Składowe RGB koloru i natężenia światła otoczenia + alpha.

```
GLvoid CLighting::SetDiffuse( GLuint uiLight,
                              GLfloat r = 1.0f, GLfloat g = 1.0f,
                              GLfloat b = 1.0f, GLfloat a = 1.0f );
```

Parametry:

`uiLight` Identyfikator światła, należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

`r, g, b, a` Składowe RGB koloru i natężenia światła rozproszonego + alpha.

```
GLvoid CLighting::SetSpecular( GLuint uiLight,
                               GLfloat r = 0.0f, GLfloat g = 0.0f,
                               GLfloat b = 0.0f, GLfloat a = 1.0f );
```

Parametry:

`uiLight` Identyfikator światła, należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

`r, g, b, a` Składowe RGB koloru i natężenia światła odbłysków + alpha.

```
GLvoid CLighting::SetPosition( GLuint uiLight,
                               GLfloat x = 0.0f, GLfloat y = 0.0f,
                               GLfloat z = 2.0f );
```

Parametry:

`uiLight` Identyfikator światła, należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

`x, y, z` Pozycja położenia światła.

Ustawienie materiałów:

```
GLvoid CLighting::SetMatAmbient( GLfloat r = 0.2f, GLfloat g = 0.2f,
                                 GLfloat b = 0.2f, GLfloat a = 1.0f );

GLvoid CLighting::SetMatDiffuse( GLfloat r = 0.8f, GLfloat g = 0.8f,
                                 GLfloat b = 0.8f, GLfloat a = 1.0f );

GLvoid CLighting::SetMatSpecular( GLfloat r = 0.0f, GLfloat g = 0.0f,
                                  GLfloat b = 0.0f, GLfloat a = 1.0f );

GLvoid CLighting::SetMatEmission( GLfloat r = 0.0f, GLfloat g = 0.0f,
                                  GLfloat b = 0.0f, GLfloat a = 1.0f );

GLvoid CLighting::SetMatShinness( GLfloat s = 50.0f );
```

Po użyciu tych wszystkich powyższych funkcji ustawiających należy już tylko uaktywnić oświetlanie metodami OGL, np.:

```
glEnable( GL_LIGHT0 );
glEnable( GL_LIGHTING );
```

Klasa `CLighting` umożliwia także rysowanie "gwiazdki" w punkcie zaczepienia światła. Dzięki temu w trakcie testowania programu będziemy mogli podejrzeć pozycję naszych świateł (o ile nie będą ustawione poza obszarem rysowania). Do wizualizacji pozycji światła należy wywołać poniższą metodę:

```
GLvoid CLighting::DrawPositionLight( GLuint uiLight,
                                     GLfloat r = 1.0f, GLfloat g = 1.0f,
                                     GLfloat b = 1.0f );
```

Parametry:

`uiLight` Identyfikator światła, należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

`r, g, b` Kolor rysowanej "gwiazdki", domyślnie biały.

18. Antyaliasing

Domyślnie OpenGLFramework wykorzystuje pełnoekranowy antyaliasing o próbkowaniu x4 (jeżeli się nie powiedzie z próbkami x4, to automatycznie spróbuje z niższymi wartościami). Oczywiście jeżeli chcemy możemy antyaliasingiem sterować. W klasie `CWindowData` zdefiniowana jest zmienna **`GLboolean m_bFullscreenAntialiasing`**. W konstruktorze `CWindowData` zmienna ta jest ustawiana na `GL_TRUE` – co oznacza, że antyaliasing zostanie użyty. Aby całkowicie wyłączyć antyaliasing wystarczy zmienić wartość zmiennej `m_bFullscreenAntialiasing` na `GL_FALSE`. Mamy także możliwość zmiany próbek. W klasie `CARMBultisample` zdefiniowana jest zmienna **`GLint m_iAntialiasingSamples`**. Możemy modyfikować przypisywaną do niej wartość – domyślnie zmienna ta ustawiona jest na 4. Podmianę próbek możemy także zrealizować z konsoli (dla szczegółów wpisz w konsoli komendę *help*). Przy zmianie próbki zawsze nastąpi restart okna aplikacji.

19. Rejestrowanie do pliku – `CLogger`

Framework wyposażony jest także w prosty logger. Jest to prosta klasa z funkcją zapisującą do pliku przekazane jej parametry. Dzięki temu w fazie testowanie naszej gry będziemy mogli rejestrować to co się dzieje w programie. Obiekt klasy `CLogger` zawsze jest tworzony jednak domyślnie logowanie jest wyłączone. Aby je włączyć możemy w odpowiednim miejscu kodu wywołać poniższą funkcję (zaleca się jednak użyć parametru */logger* – patrz niżej):

```
GLboolean CLogger::Open( LPCTSTR lpFileName );
```

Parametry:

`lpFileName` Nazwa pliku (a raczej pierwszy człon nazwy pliku), do którego będą zapisywane informacje. Pełna nazwa pliku zawiera następujący format:
lpFileName – YYYY.MM.DD – HH.MM.SS.log
gdzie:
lpFileName – to opisywany tu parametr,
YYYY.MM.DD – data (rok, miesiąc i dzień) utworzenia pliku,
HH.MM.SS – czas utworzenia pliku, tj. godzina, minuty i sekundy,
.log – rozszerzenia pliku.

Wartość zwracane:

`GL_TRUE` jeżeli plik zostanie otwarty, w przeciwnym wypadku `GL_FALSE`.

Przy wywołaniu powyższej metody logger jest już gotowy do wprowadzania informacji do pliku a wykonujemy to poprzez funkcje:

```
GLvoid CLogger::LOG( const wchar_t *fmt, ... );
GLvoid CLogger::LOG( const char *fmt, ... );
```

Parametry:

`fmt` Tekst jaki chcemy zapisać do pliku, tekst może zawierać znaki formatowania, jeżeli chcemy użyć funkcji dla UNICODE to tekst należy podać w makrze `_T("...")` oraz musi być zdefiniowane makro `UNICODE`. Jeżeli chcemy użyć funkcji z `char*` to tekst podajemy normalnie w cudzysłowie bez makra.

`...` Opcjonalne parametry dla formatowania.

`CLogger` dostarcza także możliwość logowania z wywołaniem metody WinAPI `GetLastError()`. Oznacza to, że sprawdzony zostanie kod ostatniego błędu oraz sam numer kodu jak i jego opis zostaną zapisane do pliku. Aby skorzystać z tej możliwości należy wywołać metodę:

```
GLvoid CLogger::LOGGetLastError( LPCTSTR lpWhatCallError = _T("") );
```

Parametry:

`lpWhatCallError` Tutaj podajemy dowolny tekst dzięki, któremu będziemy wiedzieli co wywołało błąd.

Np. wywołując powyższą funkcję w poniższy sposób:

```
LOGGetLastError( _T("JakasFunkcja()") );
```

w pliku **.log* otrzymamy tekst, np:

```
 0001. JakasFunkcja() - ErrCode: 2, Nie można odnaleźć określonego pliku.
```

Jak widzimy otrzymamy nasz tekst, następnie kod błędu zwrócony przez `GetLastError()` oraz opis błędu. Jak dla mnie niekiedy opisy trafiają się jakieś, takie nijakie. Dlatego też dobrze jest zajrzeć do MSDN gdzie opisane są wszystkie kody błędów w języku angielskim, które jak dla mnie klarowniej wyjaśniają powstały błąd.

Mamy także możliwość tymczasowego wyłączania i włączania logger'a. Aby tego dokonać należy wywołać:

```
GLvoid CLogger::SetActive( GLboolean bMode );
```

Parametry:

`bMode` Podając `GL_TRUE` uaktywnimy logger, podając `GL_FALSE` logger stanie się nieaktywny.

Uwagi:

Innym i chyba właściwszym sposobem otwarcia logger'a jest uruchomienie programu z parametrem "/logger". Wówczas program sam wywoła metodę `CLogger::Open()` oraz logger od razu zostanie uaktywniony (dane będą dopisywane do pliku *Log – data – czas.log*). Mamy także możliwość włączania i wyłączanie logger'a z konsoli, gdzie w linii komend wpisujemy polecenie: *log*.

Domyślnie ustawione jest także logowanie wszystkiego co wyświetlamy w konsoli. Poza tym każdy przekazany tekst automatycznie będzie przechodził do nowej linii, tak więc nie musimy pamiętać o znaku `'\n'`.
Obiekt klasy `CLogger` tworzony jest w klasie `CWindowData`, więc aby dostać się do funkcji logger'a z różnych klas będziemy musieli „skakać" po wskaźnikach, np:
```
CGameControl::GetInstance()->GetWinData()->GetLogger()->LOG(...);
```
lub bezpośrednio z klasy `CWindowData`:
```
CWindowData::GetInstance()->GetLogger()->LOG(...);
```

Najlepszym rozwiązaniem jednak będzie użycie makr:

```
__LOG( msg );
__LOGFormat( fmt, ... );
__LOGGetLastError( msg );
```

Parametry:

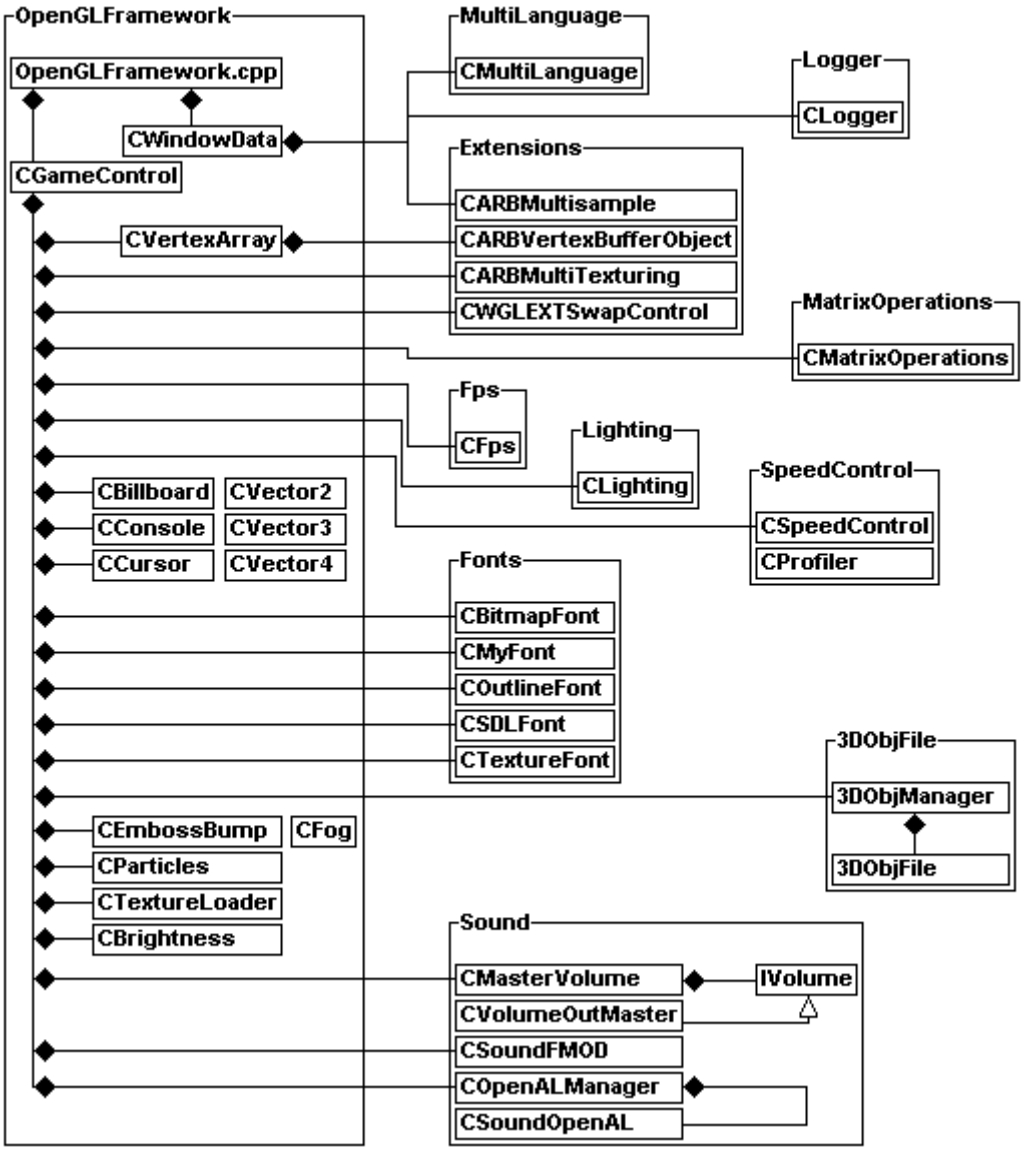
`msg`, `fmt` Tekst jaki chcemy zapisać do pliku.

`...` Opcjonalny parametr dla formatowania.

Powyższe makra będziemy mogli używać wszędzie tam gdzie „doinkudujemy" sobie pliku Logger.h.

20. Schemat Framework'a oraz dołączanie własnych klas

Poniższy schemat przedstawia zależność klas oraz podpowiada w jaki sposób budować kolejne klasy. Głównym plikiem programu jest *OpenGLFramework.cpp*, plik ten zawiera główną funkcję programu `WinMain` oraz funkcję przetwarzania komunikatów `WndProc`. W pliku tym tworzone są także dwa podstawowe obiekty: `CWindowData` oraz `CGameControl`. Klasy te są singletonami i nie możemy ponownie tworzyć któregoś z obiektów. `CWindowData` zawiera wszelkie metody związane z oknem (tworzenie, usuwanie, restartowanie), tworzy obiekt `CARBMultisample` dla wykorzystania pełnoekranowego antyaliasingu oraz tworzy obiekt klasy `CLogger` do generowania pliku rejestru. Natomiast `CGameControl` jest sercem naszej gry. Klasa ta tworzy wszelkie obiekty zaimplementowane już we framework'u takie jak wyświetlanie teksu, wczytywanie tekstur, obsługa dźwięku itp. Ważną funkcją jest `CGameControl::Draw()` – jest to funkcja rysująca wszystko to co chcemy rysować w naszej grze. Funkcja ta jest już wywoływana w funkcji `DrawGLScene()` w pliku *OpenGLFramework.cpp*. `CGameControl` otrzymuje także od *OpenGLFramework.cpp* wszelkie zdarzenia naciskania klawiszy czy klikania myszką.
Tworząc naszą grę na podstawie opisywanego tu framework'a, będziemy musieli dodawać do naszego projektu nowe klasy, np. obsługujące menu główne gry, obsługujące rysowanie jakiś postaci czy zajmujących się mechaniką gry. Wskazane jest tworzenie nowych obiektów naszych klas właśnie w `CGameControl`. Oznacza to, że nasze klasy poprzez `CGameControl` będą miały dostęp do wszystkich zaimplementowanych już klas jak `CSDLFont` czy `CSoundFMOD` potrzebnych do wyświetlania tekstu czy też odgrywania muzyki itp. Aby tak się stało wystarczy pobrać wskaźnik na obiekt `CGameControl` za pomocą statycznej metody `CGameControl::GetInstance()`.



Rys. 1 Schemat framework'a (**UWAGA! Nieaktualny**)

Poniżej przykład tworzenia naszej nowo dodanej klasy:

```
//plik MyClass.h

class CMyClass
{
public:
	CMyClass();
	virtual ~CMyClass();

private:
	void MyMethod();
};

//plik MyClass.cpp
#include "stdafx.h"
#include "GameControl.h"
#include "SoundFMOD.h"
#include "MyClass.h"
```

```
CMyClass::CMyClass()
{
}

CMyClass::~CMyClass()
{
}

void CMyClass::MyMethod()
{
	//np. chcemy odegrać jakiś dźwięk
	CGameControl::GetInstance()->GetSoundFMOD()->PlaySample(...);
}
```

Musimy także stworzyć obiekt naszej, powyższej klasy w `CGameControl`, tak więc w pliku *GameControl.h*, przed definicją klasy (`class CGameControl`) dopisujemy:

```
class CMyClass;
```

W klasie `CGameControl` w sekcji `public` dopisujemy:

```
CMyClass *m_pMyClass;
```

W pliku *GameControl.cpp* odszukujemy funkcję `CGameControl::CreateObjects()` i dopisujemy w niej:

```
m_pMyClass = new CMyClass();
```

W funkcji `CGameControl::DeleteObjects()` usuwamy obiekt, czyli dopisujemy:

```
delete m_pMyClass;
```

W ten sposób nasza nowo stworzona klasa będzie współpracować z framework'iem wg. powyższego schematu (rys. 1).

Powyższy przykład naszej nowo dodanej klasy jest bardzo ubogi i niewystarczający. Dlatego też przedstawię teraz w jaki sposób obsłużyć rysowanie oraz zdarzenia klawiatury czy myszy.

Po pierwsze rysowanie. Zapewne będziemy chcieli w naszej nowo dodanej klasie coś rysować. Np. będzie to klasa obsługująca menu główne gry (czyli wyświetlająca napisy, np. „Nowa gra", „Wczytaj grę", „Wyjście" itp.). Sprawa jest prosta, tworzymy w naszej nowej klasie, jakąś metodę rysującą, np. `CMyClass::DrawMenu()`. W metodzie tej rysujemy sobie co nam się podoba. Następnie naszą metodę musimy wywołać w `CGameControl::Draw()` (plik *GameControl.cpp*) – inaczej nic nie zobaczymy, np:

```
GLvoid CGameControl::Draw()
{
	m_pMyClass->DrawMenu();
}
```

Oczywiście w metodzie `CGameControl::Draw()` rysowanie naszego menu wypadałoby ograniczyć bo, np. w trakcie rozgrywki rysowanie menu jest niepożądane. Najlepiej do tego celu stworzyć sobie jakiś typ wyliczeniowy:

```
GLvoid CGameControl::Draw()
{
	switch( enumModeGame )
	{
```

```
            case EDrawMenu:
                  m_pMyClass->DrawMenu();
                  break;
            case EGame:
                  ...
                  break;
      }
}
```

Kolejna sprawa to przekazywanie zdarzeń naciskania klawiszy czy klikania myszą. Jak już było wspomniane wcześniej, klasa `CGameControl` otrzymuje owe zdarzenia od głównej funkcji przetwarzania komunikatów (z pliku *OpenGLFramework.cpp*). Teraz naszym zadaniem jest po prostu przekazać te zdarzenia dalej, czyli z klasy `CGameControl` do naszej własnej klasy. Opisze tu tylko zdarzenia kliknięcia myszą (co do klawiatury jest analogicznie – szczegół w pliku *GameControl.cpp*). Aby obsłużyć zdarzenie kliknięcia myszą do naszej nowej klasy dodajemy funkcję, np:

```
GLvoid MouseLButtonDown(GLint iX, GLint iY);
```

W funkcji tej będziemy reagowali na kliknięcie myszą. Parametrami funkcji są współrzędne pozycja kursora w chwili kliknięcia. Teraz w pliku *GameControl.cpp* odszukujemy funkcję `CGameControl::MouseLButtonDown(GLint iX, GLint iY)`. To właśnie wewnątrz tej funkcji przekazujemy zdarzenie kliknięcia do naszej klasy, np:

```
GLvoid CGameControl::MouseLButtonDown( GLint iX, GLint iY )
{
      switch( enumModeGame )
      {
            case EDrawMenu:
                  m_pMyClass->MouseLButtonDown( iX, iY );
                  break;
            case EGame:
                  ...
                  break;
      }
}
```

Jak widać wywołanie kliknięcia myszy także uzależniamy od jakiegoś typu wyliczeniowego – zależnie od tego co w danej chwili wyświetlamy, tak aby zdarzenie trafiło do właściwej klasy.

21. Tryb grafiki 2D

Domyślnie framework pracuje w trybie 3D (patrz `ReSizeGLScene(...)` w pliku *WindowData.cpp*). Istnieje jednak możliwość, w dowolnej chwili, przełączania się w tryb 2D i na odwrót. Służą do tego dwie funkcje klasy `CGameControl`:

`GLvoid CGameControl::Enable2D()` – włączenie trybu 2D
`GLvoid CGameControl::Disable2D()` – wyłączenie trybu 2D – powrót do 3D

Poza tym jeżeli nasza gra będzie wykorzystywać tylko tryb 2D, to wystarczy (np. w pliku *define.h*) zdefiniować makro:

```
#define _USE_2D_ONLY_
```

Wtedy to o funkcjach `Enable2D()` oraz `Disable2D()` możemy zapomnieć :)

22. Tryb pełnoekranowy

Framework ma możliwość uruchamiania się w trybie okienkowym oraz pełnoekranowym. Za to czy program uruchomi się na pełnym ekranie czy też nie, odpowiada pole `GLboolean bFullscreen` struktury `SETTING_FILE`. W metodzie `CWindowData::SaveSettingFile` dla zapisu domyślnego, pole to ustawione jest na `GL_TRUE` co oznacza, że domyślnym trybem jest tryb pełnoekranowy – oczywiście możemy to zmienić. Framework daje także możliwość przełączania się pomiędzy trybami pełnoekranowym o okienkowych – odpowiada za to funkcja:

```
GLvoid CWindowData::FullscreenSwitch();
```

Powyższa funkcja spowoduje restart okna, jednak dane zawarte w klasach `CWindowData` oraz `CGameControl` nie będą naruszone, co nie zaburzy aktualnego stanu gry.

Poza tym mamy możliwość, w trakcie uruchamiania programu, zapytania użytkownika czy program ma się uruchomić w trybie pełnoekranowym czy też nie. Aby takie pytanie się pojawiło (w formie MessageBox'a) należy zdefiniować makro `_QUESTION_FULLSCREEN_` – patrz rozdział *7. Plik define.h*.

23. Wykorzystanie VertexArrays

Obsługą tablic wierzchołków zajmuje się klasa `CVertexArrays` zawarta w projekcie `Draw`. Przechowuje i inicjalizuje ona naszą tablicę, która może przechowywać informacje nie tylko o wierzchołkach, ale także o koordynatach tekstur, wektorach normalnych oraz kolorze poszczególnego wierzchołka. Klasa ta zawiera po dwa zestawy tablic dla metody rysującej bez indeksowania (`glDrawArrays()`) oraz wydajniejszej, z indeksowaniem (`glDrawElements()`) jak i tablice indeksów. Tablice z indeksowaniem zostały stworzone głównie dla wsparcia rysowanie modeli 3D, wczytywanych z własnego formatu pliku *3DObj*. Natomiast tablice bez indeksów (czyli z powtarzającymi się wierzchołkami) możemy wykorzystać przy własnoręcznie tworzonych, prymitywnych modelach. Dlatego też zestaw funkcji dla tablic indeksowych jest uboższy. Framework umożliwi nam jednoczesne rysowanie modeli za pomocą `glDrawArrays()` jak i `glDrawElements()`. Poniższy tekst będzie tyczył się użycia VertexArrays bez indeksowania, jednak funkcje wykorzystujące indeksy są podobne – mają jedynie na końcach nazw funkcji dopisane słówko *Elem*.

Zanim zaczniemy cokolwiek rysować za pomocą naszej tablicy, musimy najpierw utworzyć ową tablicę oraz ją zainicjalizować. Należy to uczynić przed renderingiem, np. w metodzie `CGameControl::Initialization()`. Do budowania tablicy klasa `CVertexArrays` dostarcza nam szereg funkcji, za pomocą których możemy dodawać poszczególne składniki jak wierzchołki, normalne itp.:

Pojedynczy wierzchołek możemy dodać za pomocą poniższych funkcji:

```
GLvoid CVertexArrays::AddVertex( GLfloat x, GLfloat y, GLfloat z );
GLvoid CVertexArrays::AddVertex( const CVector3 &in_cVec );
```

Parametry:

`x, y, z` Współrzędne nowego wierzchołka.

`in_cVec` Adres klasy `CVector3` zawierającej współrzędne wierzchołka.

Poniższe funkcje umożliwiają nam dodanie do tablicy koordynatów tekstury jakie zostaną przypisane wierzchołkom o tym samym indeksie tablicy:

```
GLvoid CVertexArrays::AddTexCoord( GLfloat s, GLfloat t );
GLvoid CVertexArrays::AddTexCoord( const CVector2 &in_cVecTCoord );
```

Parametry:

`s, t` Koordynaty położenia tekstury.

`in_cVecTCoord` Adres klasy `CVector2` z koordynatami tekstury.

Poniższe funkcje służą do ustawiania normalnych dla danego wierzchołka o tym samym indeksie. Pierwsze dwie funkcje ustawiają normalną dla jednego wierzchołka, natomiast za pomocą dwóch ostatnich funkcji możemy ustawić normalną dla kilku wierzchołków (np. tworzących ten sam poligon):

```
GLvoid CVertexArrays::AddNormalSingle( GLfloat x, GLfloat y, GLfloat z );
GLvoid CVertexArrays::AddNormalSingle( const CVector3 &in_cVec );
GLvoid CVertexArrays::AddNormals( GLfloat x, GLfloat y, GLfloat z,
                                  GLint iNumberVertex);
GLvoid CVertexArrays::AddNormals( const CVector3 &in_cVec,
                                  GLint iNumberVertex );
```

Parametry:

`x, y, z` Współrzędne wektora normalnego.

`in_cVec` Adres klasy `CVector3` stanowiący wektor normalny.

`iNumberVertex` Liczba wektorów następujących po sobie, do których zostanie przypisana normalna.

Następnie mamy dwie funkcje służące do przypisywania koloru dla danego wierzchołka lub jak w przypadku normalnych dla grupy wierzchołków:

```
GLvoid CVertexArrays::AddColorSingle( GLfloat r, GLfloat g, GLfloat b,
                                      GLfloat a = 1.0f);
GLvoid CVertexArrays::AddColors( GLfloat r, GLfloat g, GLfloat b, GLfloat a,
                                 GLint iNumberVertex );
```

Parametry:

`r, g, b, a` Składowe koloru RGB oraz kanał alfa.

`iNumberVertex` Liczba wektorów następujących po sobie, do których zostanie przypisany kolor.

Mamy także możliwość dodania wierzchołka ze wszelkimi innymi parametrami za pomocą jednej z dwóch, poniższych funkcji:

```
GLvoid CVertexArrays::AddFullVertex( GLfloat vx, GLfloat vy, GLfloat vz,
                                     GLfloat s, GLfloat t,
                                     GLfloat nx, GLfloat ny, GLfloat nz,
```

```
                                        GLfloat r = 1.0f, GLfloat g = 1.0f,
                                        GLfloat b = 1.0f, GLfloat a = 1.0f );

GLvoid CVertexArrays::AddFullVertex( const CVector3 &in_cVecV,
                                     const CVector2 &in_cVecTCoord,
                                     const CVector3 &in_cVecN,
                                     GLfloat r = 1.0f, GLfloat g = 1.0f,
                                     GLfloat b = 1.0f, GLfloat a = 1.0f );
```

Parametry:

`vx, vy, vz` Współrzędne nowego wierzchołka.

`s, t` Koordynaty położenia tekstury.

`nx, ny, nz` Współrzędne wektora normalnego.

`in_cVecV` Adres klasy `CVector3` zawierającej wspórzędne wierzchołka.

`in_cVecTCoord` Adres klasy `CVector2` z koordynatami tekstury.

`in_cVecN` Adres klasy `CVector3` stanowiący wektor normalny.

`r, g, b, a` Składowe koloru RGB oraz kanał alfa, domyślnie ustawione na kolor biały.

Kolejna, poniższa funkcja umożliwia dodanie wierzchołków, ale bez wektorów normalnych. Te zostaną samodzielnie obliczone za pomocą kolejnych dwóch funkcji, które należy samodzielnie wywołać. Tak, więc dodawanie wierzchołka bez normalnych realizujemy za pomocą:

```
GLvoid CVertexArrays::AddFullVertexCalcNormal( GLfloat vx, GLfloat vy,
                                               GLfloat vz,
                                               GLfloat s, GLfloat t,
                                               GLfloat r = 1.0f,
                                               GLfloat g = 1.0f,
                                               GLfloat b = 1.0f,
                                               GLfloat a = 1.0f );
```

Parametry:

`vx, vy, vz` Współrzędne nowego wierzchołka.

`s, t` Koordynaty położenia tekstury.

`r, g, b, a` Składowe koloru RGB oraz kanał alfa, domyślnie ustawione na kolor biały.

Następnie w sekcji inicjalizującej musimy wywołać jedną z dwóch funkcji, która automatycznie obliczy nam wektory normalne, wedle kolejności rysowania poszczególnych wierzchołków dla danych poligonów. Pierwsza funkcja obliczy normalne z cieniowaniem płaskim, to znaczy, że wektory normalne dla danych wierzchołków będą prostopadłe do poligonu, dla którego przypisane są dane wierzchołki.
Druga funkcja natomiast obliczy nam normalne z cieniowaniem wygładzonym, gdzie wektor normalny dla danego wierzchołka będzie wektorem wypadkowym obliczonym ze wszystkich sąsiadujących poligonów.

```
GLvoid CVertexArrays::CalcAndAddNormalsFlat( GLenum eMode,
```

```
                                                    GLint iFirst, GLsizei iCount,
                                                    GLboolean bTurnSign = GL_FALSE );

GLvoid CVertexArrays::CalcAndAddNormalsSmooth( GLenum eMode,
                                                 GLint iFirst, GLsizei iCount,
                                                 GLboolean bTurnSign = GL_FALSE );
```

Parametry:

`eMode` Rodzaj poligonu, do wyboru tylko dwa: `GL_QUADS` lub `GL_TRIANGLES`.

`iFirst` Pierwszy wierzchołek z tablicy wierzchołków VertexArrays, od którego rozpocznie się obliczanie normalnych.

`iCount` Liczba kolejnych wierzchołków, dla których obliczane będą normalne.

`bTurnSign` Czy zwrot wektorów normalnych na być odwrócony. Jeżeli podamy `GL_FALSE` (wartość domyślna) to wektory normalne będą obliczane z wierzchołków tworzących poligon w kolejności przeciwnej do ruchu wskazówek zegara. Gdy `GL_TRUE` to zgodnie dla ruchem wskazówek zegara.

Przykład użycia z obliczaniem normalnych:

```
GLvoid CGameControl::CreateVertices()
{
      CVertexArrays *pVA = CVertexArrays::GetInstance();

      //tworzymy cztery wierzchołki, za pomocą których narysujemy kwadrat
      //podając wierzchołki nie przypisujemy im ręcznie normalnych:
      pVA->AddFullVertexCalcNormal( -1.0f, -1.0f, 1.0f, 0.0f, 0.0f );
      pVA->AddFullVertexCalcNormal(  1.0f, -1.0f, 1.0f, 1.0f, 0.0f );
      pVA->AddFullVertexCalcNormal(  1.0f,  1.0f, 1.0f, 1.0f, 1.0f );
      pVA->AddFullVertexCalcNormal( -1.0f,  1.0f, 1.0f, 0.0f, 1.0f );

      //obliczamy normalne:
      pVA->CalcAndAddNormalsFlat( GL_QUADS, 0, 4 );

      //inicjalizujemy tablicę VertexArray:
      pVA->InitialArrays();
}

//rysowania naszago kwadratu w metodzie renderującej:
GLvoid CGameControl::Draw()
{
      ...
      glDrawArrays( GL_QUADS, 0, 4 );
      ...
}
```

Po dodaniu odpowiedniej ilości wierzchołków należy naszą tablicę zainicjalizować. Dokonamy tego za pomocą jednej z trzech poniższych funkcji, przy czym druga zainicjalizuje nam tablice z wykorzystaniem multiteksturingu a trzecia z wykorzystaniem emboss bump mappingu:

```
GLboolean CVertexArrays::InitialArrays();
GLboolean CVertexArrays::InitialArraysWithMultiTex( GLsizei iCount = 2 );
GLboolean CVertexArrays::InitialArraysWithEmbossBump();
```

Parametry:

`iCount` Ilość tekstur dla multiteksturingu.

Jednak gdy chcemy korzystać z multiteksturingu nie musimy od razu inicjalizować tablicy z tym wsparciem. Możemy bowiem w każdej chwili renderingu przełączać się za pomocą poniższych funkcji. Pierwsza funkcja przeinicjalizuje nam tablice bez użycia multiteksturingu, druga zapewni nam wsparcie dla multiteksturingu. Patrz także rozdział *9.1. Multiteksturing*.

```
GLvoid CVertexArrays::EnableTextureCoord();
GLvoid CVertexArrays::EnableTextureCoordMultiTex( GLsizei iCount = 2 );
```

Parametry:
`iCount` Ilość tekstur dla multiteksturingu.

W końcu po utworzeniu i zainicjalizowaniu tablicy możemy rysować jej wybrane wierzchołki za pomocą, zapewne znanej nam metody `glDrawArrays()`.
Klasa `CVertexArrays` współpracuje także z `CARBVertexBufferObject`, szczegóły w następnym rozdziale.

24. Wykorzystanie Vertex Buffer Object

Obsługą VBO zajmuje się klasa `CARBVertexBufferObject` zawarta w projekcie `Extensions` i jest ściśle powiązana z klasą `CVertexArrays`. Aby wykorzystać VBO po pierwsze w konstruktorze klasy `CARBVertexBufferObject` należy ustawić zmienną `m_bUse` na `GL_TRUE` i tak jest też domyślnie. Ustawiając tę zmienna na `GL_FALSE`, VBO nie będzie w użyciu. Następnie w trakcie inicjalizacji, należy wywołać szereg funkcji `Create...Buffer()`, domyślnie są one wywoływane w metodzie `CVertexArrays::InitialArrays()`:

```
GLvoid CARBVertexBufferObject::CreateVerticesBuffer( const GLvoid *lpVoid,
                                                     GLsizei nSize );
GLvoid CARBVertexBufferObject::CreateNormalsBuffer( const GLvoid *lpVoid,
                                                    GLsizei iSize );
GLvoid CARBVertexBufferObject::CreateColorsBuffer( const GLvoid *lpVoid,
                                                   GLsizei iSize );
GLvoid CARBVertexBufferObject::CreateTexCoordsBuffer( const GLvoid *lpVoid,
                                                      GLsizei iSize );
```

Parametry:
`lpVoid` Wskaźnik na daną tablicę.

`iSize` Rozmiar tablicy.

Następnie w metodzie renderującej, na jej początku (a przynajmniej przed wywołaniem `glDrawArays()`) należy wywołać poniższą funkcję:

```
GLboolean CVertexArrays::DrawArrays();
```

i/lub z wykorzystaniem multiteksturingu:

```
GLboolean CVertexArrays::DrawArraysWithMultiTex( GLsizei iCount );
```

Parametry:
`iCount` Ilość tekstur dla multiteksturingu.

Wartość zwracana:
`GL_TRUE` w przypadku powodzenia.

Na końcu w metodze renderującej musimy wywołać:

```
GLvoid CVertexArrays::DisableClientState();
```

Na zakończenie programu, należy usunąć VBO poprzez metody `CARBVertexBufferObject::Delete...Buffers()`. Metody te są już wywoływane w destruktorze klasy `CVertexArrays`.

25. Własny format pliku *3DObj*

Format pliku *3DObj* jest tworzony poprzez program 3DSLoader (dodatkowego narzędzia stworzonego zresztą na tym samym frameworku), którego zadaniem jest importowanie plików 3ds i na ich podstawie stworzenie plików *3DObj*. Za obsługę plików *3DObj* odpowiada klasa `C3DObjManager` zawarta w projekcie `Draw`. W jednym pliku *3DObj* znajduje się jeden obiekt 3D (tj. jednolita siatka tworząca obiekt), który możemy wczytać za pomocą poniższej metody:

```
GLboolean C3DObjManager::Create3DObject( LPCTSTR lpFileName,
                                         GLint iTextureFirst,
                                         GLint iTextureCount = -1,
                                         GLint iTextureEmbossBump = -1 );
```

Parametry:

`lpFileName` — Nazwa pliku z rozszerzeniem *3DObj*, który chcemy wczytać.

`iTextureFirst` — Indeks tekstury jaka zostanie nałożona na obiekt. Indeksy liczone są od 0 w kolejności wczytywania tekstur. Jeżeli zamierzamy wykorzystać multiteksturing, parametr ten stanowi indeks pierwszej tekstury jaka zostanie nałożona na obiekt.

`iTextureCount` — Dla multiteksturingu – ilość kolejnych tekstur po `iTextureFirst`, jakie zostaną nałożone na obiekt. Aby miało sens, należy podać co najmniej wartość 2. Jeżeli podamy wartość mniejszą od 2, multiteksturing nie zostanie użyty. Domyślna wartość dla `iTextureCount` wynosi -1, czyli bez multiteksturingu.

`iTextureEmbossBump` — Indeks tekstury jaka zostanie nałożona na obiekt w celu uzyskania efektu wybojów. Musi to być specjalnie do tego celu przygotowana tekstura, najczęściej w odcieniach szarości. Należy pamiętać, że jeżeli chcemy wykorzystać teksturę bump to parametr `iTextureCount` musi być równy wartości domyślnej, tj. -1. Jeżeli nie chcemy wykorzystać mapowania wybojów to należy parametr ten ustawić na wartość domyślną, czyli -1.

Wartość zwracana:
Indeks nowo stworzonego obiektu. W przypadku błędu zwracana jest wartość -1.

Metoda `Create3DObject()` automatycznie utworzy nowe wierzchołki do odpowiednich tablic zawartych w klasie `CVertexArrays`.

Po wczytaniu obiektu, możemy w trakcie renderingu wywołać poniższe metody w celu narysowania wcześniej wczytanego obiektu:

```
GLvoid C3DObjManager::Draw3DObject( GLint iIndex,
                                    GLboolean bUseEmbossBump = GL_FALSE,
                                    GLuint uiLight = GL_LIGHT0 );

GLvoid C3DObjManager::Draw3DObject_Lists( GLint iIndex,
                                          GLboolean bUseEmbossBump = GL_FALSE,
                                          GLuint uiLight = GL_LIGHT0 );
```

Parametry:

`iIndex` — Indeks obiektu jaki chcemy narysować. Indeksy liczone są od 0 w kolejności wczytywania obiektów, tj. w kolejności wywoływania metody `Create3DObject()`.

`bUseEmbossBump` — Ustawiając na `GL_TRUE` obiekt będzie rysowany z efektem mapowania wybojów. Aby jednak tak się stało należy uprzednio w metodzie `Create3DObject()` podać indeks tekstury bump oraz ilość kolejnych tekstur dla multiteksturingu pozostawić w wartości domyślnej. Ustawiając na `GL_FALSE` (wartość domyślna) efekt mapowania wybojów nie będzie używany przy rysowaniu obiektu, nawet jeżeli przy tworzeniu obiektu podaliśmy teksturę bump.

`uiLight` — Identyfikator światła, używany jest tylko dla efektu mapowania wybojów. Należy użyć `GL_LIGHTx`, gdzie `x` to cyfra od 0 do 7.

Dla zwiększenia wydajność rysowania obiektów zaleca się używać metody `Draw3DObject_Lists()`, która to rysuje obiekty z wcześniej przygotowanych list wyświetlania OpenGL.

Poniżej znajduje się opis formatu pliku *3DObj*. Właściwie to są dwa formaty, nieco różniące się od siebie. Jeden z wykorzystaniem indeksowania wierzchołków a drugi bez. To jaki plik *3DObj* zostanie stworzony zależy w jaki sposób go stworzymy narzędziem 3DSLoader. Jednak nie musimy się tym przejmować – przy wczytywaniu plików *3DObj*, metoda `Create3DObject()` sama rozpozna, z którym formatem ma do czynienia. Dla lepszej wydajności zaleca się używać indeksowania wierzchołków to też poniżej znajduje się opis formatu pliku *3DObj* wykorzystujący indeksowanie.

Na początku zawsze mamy nagłówek pliku:

```
14 bajtów – musi zawierać tekst zakończony zerem: „OGLF PlayeRom”,
2 bajty (unsigned short) – główna wersja pliku, musi być 1,
2 bajty (unsigned short) – pomniejsza wersja pliku, musi być 0,
1 bajt (unsigned char) – czy obiekt używa indeksowania wierzchołków, jeżeli 1 to
tak, 0 – nie.
```

Tu się kończy nagłówek. Dalej mamy:

```
2 bajty (unsigned short) – ilość wierzchołków,
```

Następnie, cyklicznie – tyle razy ile mamy wierzchołków, wczytywane są następujące dane:

```
12 bajtów (3 * float) – współrzędne X, Y, Z wierzchołka,
 8 bajtów (2 * float) – koordynaty tekstury,
```

12 bajtów (3 * float) – współrzędne X, Y, Z wektora normalnego,

Następnie mamy informacje o tablicy indeksów:
2 bajty (unsigned short) – ilość indeksów (czyli ilość trójkątów * 3),

Następnie, cyklicznie – tyle razy ile mamy indeksów:
2 bajty (unsigned short) – indeks wierzchołka.

26. Synchronizacja pionowa

OpenGLFramework umożliwia włączanie bądź wyłączanie synchronizacji pionowej w trakcie działania programu. Pamiętajmy aby jednak umożliwić kontrolę synchronizacji pionowej przez aplikację musimy jej to umożliwić przez sterownik naszej karty graficznej.
Za kontrolę synchronizacji pionowej odpowiada klasa `CWGLEXTSwapControl` z projektu `Extensions`, która wyposażona jest w poniższe funkcje.
Aby wyłączyć synchonizację pionową należy wywołać:

```
GLvoid CWGLEXTSwapControl::DisableVSync();
```

Jeżeli chcemy włączyć synchronizacje pionową należy wywołać:

```
GLvoid CWGLEXTSwapControl::EnableVSync();
```

Wywołanie poniższej metody spowoduje, że synchronizacja pionowa zostanie włączona tylko wtedy gdy ustawienia sterownikow graficznych na to pozwalają, czyli gdy `CWGLEXTSwapControl::m_iDefaultSwapInterval` będzie równy 1.

```
GLvoid CWGLEXTSwapControl::DefaultVSync();
```

Za pomocą poniższej funkcji możemy sami sobie ustwić synchronizacje pionową:

```
GLvoid CWGLEXTSwapControl::SetSwapInterval( GLint iInterval );
```

Parametry:
`iInterval` Podanie 0 wyłącza synchronizację pionową, 1 – włącza.

27. Cząsteczki

//TODO:

28. Rysowania billboard'ów

Billboard jest to sprite, który zawsze zwrócony jest przodem do widza (kamery), nawet wtedy (a zwłaszcza wtedy) gdy kamera lub sprite zmienia swoje położenie. Aby móc rysować takiego sprite'a należy skorzystać z klasy `CBillboard` projektu `Draw`. Klasa ta zawiera tylko jedną metodę przeznaczoną właśnie to rysowania billboard'owego sprite'a:

```
GLvoid CBillboard::DrawQuadBillboard( const CVector3 &in_cPos,
                                      GLfloat fScale = 1.0f,
                                      GLenum eMode = GL_TRIANGLE_STRIP,
                                      GLfloat fTexCoordX1 = 0.0f,
```

```
                                        GLfloat fTexCoordX2 = 1.0f,
                                        GLfloat fTexCoordY1 = 0.0f,
                                        GLfloat fTexCoordY2 = 1.0f );
```

Parametry:

`in_cPos` Centralny punkt położenia sprite'a.

`fScale` Rozmiar sprite'a, standardowo 1.0f, podając np. 2.0f sprite będzie dwa razy większy.

`eMode` Tryb rysowania sprite'a. Możemy tu podać `GL_TRIANGLE_STRIP` (standardowo), wówczas narysowany zostanie quad lub `GL_LINE_LOOP`, gdzie narysowany zostanie kwadratowy kontur z linii.

`fTexCoordX1` Koordynaty tekstury x1.

`fTexCoordX2` Koordynaty tekstury x2.

`fTexCoordY1` Koordynaty tekstury y1.

`fTexCoordY2` Koordynaty tekstury y2.

Uwagi:
Jeżeli nie chcemy nic mieszać z teksturami, pozostawmy domyślne parametry.

29. Profiler

Za pomocą profilera możemy zmierzyć czas wykonywania się poszczególnych funkcji. Dzięki temu możemy zdiagnozować, które metody spowalniają nam program i które należałoby zoptymalizować. Zadanie pomiaru czasu wykonywania się kodu spoczywa na klasie `CProfiler` zawartej w projekcie `SpeedControl`. Klasa `CProfiler` jest singletonem, jej instancja jest już we frameworku tworzona i usuwana automatycznie, więc naszym zadaniem jest tylko wywoływanie odpowiednich metod. Dla łatwego włączenia/wyłączenia użycia profilera (bez konieczności usuwania metod profilera z wersji release) zaleca się ustawić makro `_USE_PROFILER_` na odpowiednio 1 lub 0. Dla użycia profilera należy wywołać w odpowiednich miejscach następujące makra:

```
PROFILER_DEGIN( strDesc );
PROFILER_END();
```

Parametry:

`strDesc` Komentarz jaki zostanie zapisany w logu profilera. Najlepiej jest podać tu nazwę funkcji, którą testujemy.

Tak więc mamy parę metod: `Begin()` oraz `End()`. Teraz jeżeli chcemy zmierzyć czas wykonywania się jakiejś funkcji, to na samym początku danej funkcji wywołujemy `Begin()`, gdzie dobrze jest podać jako parametr nazwę metody w jakiej wywołujemy `Begin()`. Natomiast na zakończenie danej funkcji wywołujemy metodę `End()`. Należy pamiętać, że musi być tyle samo wywołań metod `Begin()` oraz `End()`. Czyli przed każdym wcześniejszym wyjściem z testowanej funkcji, należy przed każdą instrukcją `return` wywołać `End()`.

Jeżeli wewnątrz danej funkcji chcemy sprawdzić czas wykonania się innej funkcji, wywoływanej z pierwszej, to nie ma najmniejszego problemu – postępujemy zawsze tak samo: na początek `Begin()`, na zakończenie `End()`.
Rezultat pracy profilera zostanie zapisany do pliku *ProfilerLog – rrrr.mm.dd – HH.MM.SS.log*.

Przykład:

```
GLint CMyClass::MyMethod()
{
	PROFILER_BEGIN( _T("CMyClass::MyMethod()") );

	...
	... jakieś obliczenia
	...
	//wywołanie innej metody, w której także wywołujemy Begin() i End():
	MyAnotherMethod();
	...
	... jakieś obliczenia
	...
	if( bError )
	{
		//gdzieś w środku wychodzimy z funkcji, więc pamiętajmy o End()

		PROFILER_END();
		return -1;
	}
	...
	... jakieś obliczenia
	...

	PROFILER_END();

	return 0;
}

GLvoid CMyClass::MyAnotherMethod()
{

	PROFILER_BEGIN( _T("CMyClass::MyAnotherMethod()") );

	...
	... jakieś obliczenia
	...

	PROFILER_END();
}
```

Z powyższego przykładu otrzymamy, np. taki log:

```
000001. MAIN LOOP. Time: 16 [ms], Frame: 16 [ms], Portion: 100.00%
000002.   CMyClass::MyMethod(). Time: 16 [ms], Frame: 16 [ms], Portion: 100.00%
000003.       CMyClass::MyAnotherMethod(). Time: 2 [ms], Frame: 16 [ms], Portion: 12.50%
... itd.
```

Jak widzimy w pliku zostanie zarejestrowane rozpoczęcie wywołania danej funkcji, oraz jej zakończenie. Najpierw zapisany jest komentarz jaki podaliśmy w makrze `PROFILER_BEGIN`. Następnie `Time` informuje o czasie wykonania danej funkcji w milisekundach, `Frame` mówi nam ile milisekund trwało wygenerowanie jednej ramki. `Portion` wskazuje w procentach udział

wykorzystania czasu ramki dla badanej funkcji. Zagnieżdżenie funkcji obrazowane jest odpowiednią ilością tabulatorów, dzięki czemu log jest czytelniejszy.

30. Sterowanie jasnością obrazu

Możliwość zmiany jasności rysowanego obrazu daje nam klasa `CBrightness` zwarta w projekcie `Draw`. Obiekt tej klasy jest już tworzony w `CGameControl` oraz „rysowanie jasności" także jest już wywołane w `CGameControl::Draw()`. Sterowanie jasnością jest zrealizowane na zasadzie sztuczki z rysowaniem prostokąta rozciągniętego na całym ekranie. Prostokąt ten może być rysowane w dwóch kolorach: czarnym lub białym – zależnie czy obraz chcemy przyciemnić czy rozjaśnić. Prostokąt ten jest rysowany także z odpowiednią przezroczystością i właśnie gęstość tej przezroczystości określa nam jak bardzo obraz ma być przyciemniony lub rozjaśniony. Aby zmienić ów gęstość przezroczystości prostokąta, czyli ustawić jasność obrazu należy wywołać poniższą metodę:

```
GLvoid CBrightness::SetBrightness( GLint iBrightness = 0 );
```

Parametry:

`iBrightness` Wartość jasności od -255 do 255, gdzie -255 to całkowite przyciemnienie (obraz będzie czarny) a 255 to całkowite rozjaśnienie (obraz będzie biały). Podając 0 (wartość domyślna) obraz będzie rysowany w oryginalnej jasności.

31. Shadow volume

OpenGLFramework umożliwia rysowanie dynamicznych cieni (tj. zmieniających się pod wpływem animacji obiektu rzucającego cień) metodą shadow volume tj. z wykorzystaniem bufora szablonu. Rysowaniem cieni zajmuje się klasa `CStencilShadow` zawarta w projekcie `Draw`. Jednak pomimo tej klasy dającej nam wsparcie, należy samodzielnie w odpowiedni sposób rysować obiekty rzucające cienie. Poniżej mamy przykłady rysowania cieni. Pierwszy przykład gdzie rysujemy obiekt wczytany z pliku *3DObj* (`CStencilShadow` współpracuje tylko z obiektami *3DObj*), ale bez cieni. W poniższych przykładach `m_pGameCtrl` jest wskaźnikiem na obiekt klasy `CGameControl`.

```
////////////////////////////////////////////////////////////////////////////
////////////////////// Pierwszy przykład – normalnie bez cienia ////////////////
////////////////////////////////////////////////////////////////////////////

// Pomocnicze dla skrócenia zapisu:
C3DObjManager  *p3DObjMgr = CGameCtontrol::GetInstance()->Get3DObjManager();

glLoadIdentity();

glTranslatef( 20.0f, 30.0f, -100.0f );    // Przesuwamy obiekt
glRotatef( 45.0f, 0.0f, 1.0f, 0.0f );     // Obracamy obiekt

// Rysujemy obiekt:
p3DObjMgr->Draw3DObject_Lists( i3DObjIndex, GL_TRUE, GL_LIGHT1 );

////////////////////////////////////////////////////////////////////////////
```

Teraz na podstawie powyższego przykładu, przerobimy kod tak aby rysowany obiekt rzucał dynamiczny cień. Zakładam, że cień będzie rzucany na podstawie światła `GL_LIGHT1` oraz, że

parametry tego światłą są już wcześniej ustawione.

```
////////////////////////////////////////////////////////////////////////////
//////////////////// Drugi przykład – z cieniem ////////////////////////////
////////////////////////////////////////////////////////////////////////////

// Pomocnicze dla skrócenia zapisu:
C3DObjManager  *p3DObjMgr = CGameCtontrol::GetInstance()->Get3DObjManager();
CStencilShadow *pShadow   = CGameCtontrol::GetInstance()->GetStencilShadow();
CLighting      *pLighting = CGameCtontrol::GetInstance()->GetLighting();

// Nowa zmienna pomocnicza, zapamiętująca wyliczoną pozycję światła dla cienia:
CVector4 m_cLightPosShadow;

if( „rysuj z cieniem” )
{
      glLoadIdentity();

      // Tutaj wykonujemy wszelkie operacje jak oryginalne, ale w odwrotnej
      // kolejności oraz z przeciwnym znakiem:
      glRotatef( -45.0f, 0.0f, 1.0f, 0.0f );    // Obracamy obiekt
      glTranslatef( -20.0f, -30.0f, 100.0f );   // Przesuwamy obiekt

      CVector4 vLight1Pos( pLighting->GetPosition( GL_LIGHT1 ) );

      // Teraz należy wywołać funkcję wyliczającą pozycję światła, na podstawie
      // przeciwnych operacji niż oryginalnie:
      m_cLightPosShadow = pShadow->GetLightPos( vLight1Pos );
}

glLoadIdentity();

glTranslatef( 20.0f, 30.0f, -100.0f );    // Przesuwamy obiekt
glRotatef( 45.0f, 0.0f, 1.0f, 0.0f );     // Obracamy obiekt

// Rysujemy obiekt:
p3DObjMgr->Draw3DObject_Lists( i3DObjIndex, GL_TRUE, GL_LIGHT1 );

if( „rysuj z cieniem” )
{
      // Na koniec należy wywołać poniższą metodą aby narysować cień dla
      // naszego pojedynczego obiektu:
      pShadow->DrawShadowForObj( i3DObjIndex, m_cLightPosShadow );
}

////////////////////////////////////////////////////////////////////////////
```

Jak widzimy w powyższym przykładzie, ważne jest wywołanie metody `CStencilShoadow::GetLightPos()` po wykonaniu wszelkich operacji, ale w sposób całkowicie odwrotny od oryginalnego. Następnie wykonujemy wszelkie translacje w sposób oryginalny i rysujemy obiekty tak jak w pierwszym przykładzie. Następnie wystarczy wywołać metodę rysującą cień `CStencilShoadow::DrawShadowForObj()` gdzie bardzo istotne jest podanie pozycji cienia zwróconej przez `CStencilShoadow::GetLightPos()`.

Powyższa metoda opisana na przykładzie, rzuca cień tylko dla jednego obiektu 3D. Czyli rysując np. w pętli kilkanaście obiektów 3D za każdym razem musimy dla każdego obiektu wywołać metodę `DrawShadowForObj`. Jednak `CStencilShadow` umożliwia nam także rysowanie cieni od razu dla kilku obiektów 3D. Polega to na tym, że w trakcie rysowania obiektów 3D

zapisujemy za pomocą dostępnych metod w `CStencilShadow` wszelkie informacje o przekształceniach macierzy widoku. Następnie na sam koniec rysowania, należy wywołać metodę `DrawShadowForObjects`, która to od razu narysuje wszystkie cienie. Poniżej znajduje się przykład wykorzystania opisywanej tu metody rysowania cieni dla kilku obiektów na raz. Nowe wywołania funkcji w odniesieniu do powyższego przykładu, zostały pogrubione, oraz dla lepszej czytelności usunięto warunek sprawdzania czy „rysujemy z cieniem".

```
////////////////////////////////////////////////////////////////////////////
////////////////// Trzecie przykład – z cieniem ////////////////////////////
////////////////////////////////////////////////////////////////////////////

// Pomocnicze dla skrócenia zapisu:
C3DObjManager  *p3DObjMgr = CGameCtontrol::GetInstance()->Get3DObjManager();
CStencilShadow *pShadow   = CGameCtontrol::GetInstance()->GetStencilShadow();
CLighting      *pLighting = CGameCtontrol::GetInstance()->GetLighting();

// Rozpoczęcie zbierania informacji o przekształceniach
pShadow->Begin_ObjTransfInfo();

// Pętla po obiektach 3D:
for( GLint i = 0; i < „Ilość obiektów"; ++i )
{
      CVector4 m_cLightPosShadow;
      glLoadIdentity();
      glRotatef( -45.0f, 0.0f, 1.0f, 0.0f );
      glTranslatef( -20.0f, -30.0f, 100.0f );
      CVector4 vLight1Pos( pLighting->GetPosition( GL_LIGHT1 ) );
      m_cLightPosShadow = pShadow->GetLightPos( vLight1Pos );

      // Bardzo ważne wywołanie metody Set_ObjTransfInfo:
      pShadow->Set_ObjTransfInfo( i, m_cLightPosShadow );

      glLoadIdentity();

      glTranslatef( 20.0f, 30.0f, -100.0f );

      // Zapisujemy informację o zajściu translacji:
      pShadow->Translate_ObjTransfInfo( CVector3( 20.0f, 30.0f, -100.0f ) );

      glRotatef( 45.0f, 0.0f, 1.0f, 0.0f );

      // Zapisujemy informację o zajściu rotacji:
      pShadow->Rotate_ObjTransfInfo( 45.0f, CVector3( 0.0f, 1.0f, 0.0f ) );

      p3DObjMgr->Draw3DObject_Lists( i, GL_TRUE, GL_LIGHT1 );

      // Bardzo ważne wywołanie zakończenia zbierania informacji o
      // przekształceniach dla danego obiektu:
      pShadow->End_ObjTransfInfo();
}

// Na sam koniec, poza pętlą, rysujemy cienie dla wszystkich obiektów:
pShadow->DrawShadowForObjects();

////////////////////////////////////////////////////////////////////////////
```

32. Bibliografia

Strony, które okazały się pomocne przy tworzeniu frameworka:

http://nehe.gamedev.net
http://www.warp.pl
http://www.codeprojects.com
http://xion.gamedev.pl/texts/megatutorial.xml

Wykorzystywane biblioteki:

http://www.libsdl.org/
http://www.libsdl.org/projects/SDL_ttf/
http://www.fmod.org/
http://www.openal.org/
http://www.vorbis.com/