1 Renderowanie tekstu

Włączanie tekstu do widoku renderowanej sceny sprowadza się zasadniczo do przygotowania zestawu znaków [1], możliwych do obsadzenia w widoku sceny w postaci tekstur. W zależności od przyjętego sposobu rzutowania możliwe jest obsadzenie tekstu we wnętrzu sceny z uwzględnieniem przesłonięć i operacji transformacji geometrycznych 3D, takich jak translacje czy rotacje. Wcześniejsze wersje OpenGL oferowały ponadto możliwość zapisania fragmentu mapy bitowej a buforze kolorów OpenGL, przez co możliwe było ńałożenie"tekstu na pierwszy plan wyrenderowanej sceny. Tego typu technika nadal jest osiągalna z wykorzystaniem zewnętrznych wobec biblioteki mechanizmów WinAPI. Obecnie omówimy pierwszą z technik renderowania tekstu z uwzględnieniem różnych sposobów projekcji sceny. W drugiej części ćwiczeń omówione zostanie polecenie odczytu fragmentu bufora kolorów aby uzyskać efekt tworzenia zrzutów ekranowych z widoku sceny.

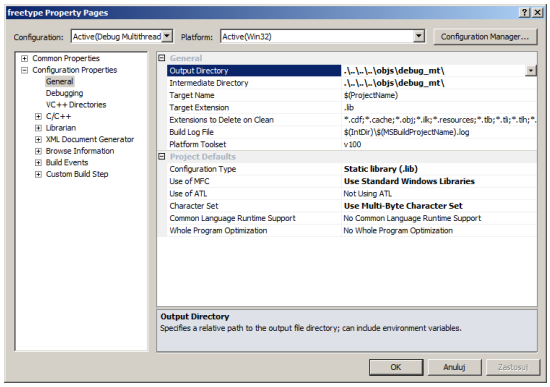
1.1 Generowanie wzorców znaków i napisów z biblioteką FreeType

Kroje czcionek dostępnych w systemie operacyjnym są przechowywane w plikach `*.ttf` Informacje o kroju czcionki uwzględniają szereg informacji typograficznych, takich jak odstępy między poszczególnymi znakami, szerokość, wysokość poszczególnych znaków, odległości między liniami, sposób skalowania dla różnych rozmiarów, tworzenie znaków narodowych oraz wiele innych. Biblioteka FreeType jest otwartą biblioteką, pozwalającą renderować znaki z wybranej czcionki w postaci map bitowych. Ponadto biblioteka udostępnia interfejs metryki czcionki [2] wyciągający z definicji czcionki wszystkie krytyczne informacje gabarytowe. Obecnie omówimy sposób włączania biblioteki FreeType do projektu aplikacji OpenGL realizowanego z użyciem VisualStudio.

1.1.1 Przebieg ćwiczenia

1. Rozpakuj pliki źródłowe do lekcji.
2. Rozpakuj archiwum freetype-2.4.0.zip do wybranego folderu
3. Przejdź do folderu z rozpakowanym archiwum i wybierz podfolder `\builds\win32\vc2008`
4. Otwórz rozwiązanie `freetype.sln` w Visual Studio
5. Dla Visual Studio w wersji wyższej niż 2008 uruchomi się kreator konwersji. Przejdź kroki kreatora aby wykonać konwersję i załadować projekt w środowisku.
6. Kliknij prawym klawiszem na `freetype` w drzewie `Solution Explorer`. Wybierz opcję `Preferences` Upewnij się, że wybrana jest konfiguracja `Debug` jak na rys. 1
7. Skompliuj projekt przyciskiem `F7`
8. Upewnij się, że w podfolderze `\objs\win32\vc2008` pojawił się plik `freetype240MT_D.lib`
9. Utwórz nowy projekt programistyczny i podłącz do niego pliki źródłowe do lekcji.
10. W analogiczny sposób, jak poprzednio wywołaj okno właściwości dla utworzonego projektu. Przejdź do pozycji `VC++ Directories`. Uzupełnij pozycje `IncludeDirectories` `LibraryDirectories` `SourceDirectories` o ścieżki dostępu do podfolderów `include` oraz `objs\win32\vc2008`
11. Przejdź do zakładki `Linker->Input`
12. Uzupełnij pozycję `AdditionalDependencies` o `freetype240MT_D.lib`

[1] ang. glyphs
[2] ang. font metrics



Rysunek 1: Okno właściwości projektu

13. Uzupełnij plik `printer.h` o dyrektywy włączające główny plik nagłówkowy biblioteki do projektu.

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

14. Skompiluj i uruchom program. W razie problemów z kompilacją prześledź ponownie powyższą listę poleceń.

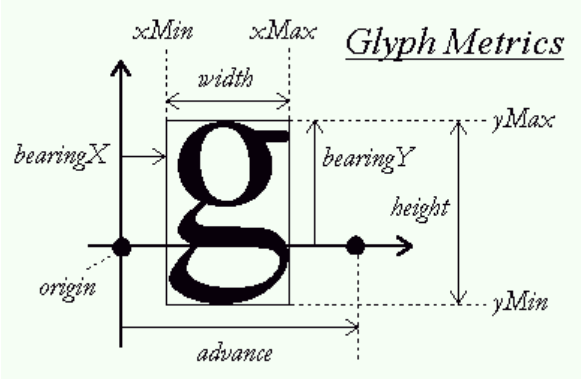
Na bieżącym etapie rozwoju aplikacji tworzona jest scena z teksturowaną kulą. Obserwator porusza się wokół obiektu po wysterowaniu klawiszami `WSAD`. Obiekt można obracać z wykorzystaniem klawiszy strzałkowych. W projekcie znajdują się pliki `printer.cpp printer.h` implementujące klasę prostego generatora znaków. Podstawową metodą z tej klasie jest `CreateGlyph`, która wykorzystuje bibliotekę FreeType do wytworzenia bitmapy zawierającej widok znaku i użycia tej bitmapy do wytworzenia tekstury.

1.1.2 Przebieg ćwiczenia

1. Zapoznaj się z konstruktorem klasy `glPrinter`. Zwróć uwagę na spodziewane parametry wywołania konstruktora.
2. Zapoznaj się z treścią metody `CreateGlyph`. Zwróć uwagę na sposób pobierania i przechowywania metryki tworzonego znaku. Geometryczną interpretację zmiennych występujących w kodzie zawarto na rys. 2 Co jest przechowywane w zmiennych `CharWidth` oraz `CharHeight`?
3. Wprowadź jako składową prywatną klasy `Scene` obiekt `Prn`, który będzie reprezentował instancję klasy `glPrinter`
4. W metodzie `PrepareObjects` wywołaj

```
Prn = new glPrinter("tahoma.ttf",24);
```

5. W kodzie rysowania sceny wywołaj `Prn->Draw('A')`
6. Uruchom aplikację. Odczytaj w oknie Log informacje o wymiarach największego spośród załadowanych znaków.
7. Rozbuduj kod rysowania sceny, aby wprowadzić skalowanie względem osi `X` oraz `Y` tak, aby największy znak tekstu zmieścił się w obszarze o wymiarach 0.5 na 0.5 we współrzędnych sceny. W tym celu rozbuduj kod rysowania sceny do postaci



Rysunek 2: Parametry geometryczne tekstu w orientacji poziomej.

```
        mTransform = glm::scale(mTransform,
               glm::vec3(0.5/float(Prn->CharWidth),0.5/float(Prn->CharHeight),1.0));
        glUniformMatrix4fv(_NormalMatrix, 1, GL_FALSE,
               glm::value_ptr(glm::transpose(glm::inverse(mTransform))));
        glUniformMatrix4fv(_ModelView, 1, GL_FALSE,
               glm::value_ptr(mModelView*mTransform));
        Prn->Draw('A');
```

Pamiętaj, aby modyfikacja znalazła się bezpośrednio przed rysowaniem litery.

W bieżącej postaci kod generuje pojedynczy czworokąt z nałożoną teksturą litery. Zwróćmy uwagę na geometrię tego czworokąta w odniesieniu do geometrii znaku. Ponadto czworokąt generowany jest z czarnym tłem, które przesłania scenę bezpośrednio za nim. Aby nadać przezroczystość tekstowi można wprowadzić mechanizm mieszania (blending) z równaniem, które jako kanał Alpha będzie pobierało wartość koloru pikseli z tekstury ze znakiem. W ten sposób obszar znaku będzie rysowany założonym kolorem, a obszar jego dopełnienia (tła znaku) będzie traktowany jako przezroczysty. Realizacja tego efektu wymaga modyfikacji shadera fragmentów oraz kodu rysowania sceny.

1. Rozbuduj kod generowania znaku do następującej postaci:

```
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glDisable(GL_DEPTH_TEST);
        Prn->Draw('A');
        glEnable(GL_DEPTH_TEST);
        glDisable(GL_BLEND);
```

2. Zmodyfikuj kod shadera wierzchołków do następującej postaci:

```
void main()
{
        vTexColor = texture2D(gSampler, texCoord);
        float LightDiffuse = max(0.0, dot(normalize(vNormal), -LightDirection));
        //outputColor = kolorek*vTexColor*vec4(LightColor*(LightAmbient+LightDiffuse),1.0);
        outputColor = kolorek*(1.0,1.0,1.0,vTexColor.r);
```

```
}
```

3. Rozbuduj projekt, aby na scenie pojawiło się Twoje imię przechowywane w tablicy znaków. Wykorzystaj operacje translacji, aby zapewnić możliwość przesuwania miejsca rysowania kolejnych znaków.

4. Rozbuduj projekt, aby na scenie znalazła się aktualna wartość zmiennej `rot_y`

Zwróć uwagę, że tekst generowany na scenie z wykorzystaniem rzutowania perspektywicznego może być przekształcany z wykorzystaniem klasycznych przekształceń geometrycznych. Wprowadzenie przekształceń perspektywy może w ogólności zaburzyć czytelność tekstu. W związku z czym korzystnie jest renderować tekst na scenie z wykorzystaniem rzutowania ortogonalnego. Obecnie omówimy przykład projektu, w którym scena renderowana jest z wykorzystaniem dwóch różnych sposobów rzutowania.

1.1.3 Przebieg ćwiczenia

1. Usuń transformację skali przy renderowaniu tekstu.

2. W miejscu kodu renderującego napis wprowadź dyrektywy zmieniające sposób rzutowania zgodnie z poniższym listingiem.

```
        //-----------------------------------------------
        // Rysowanie w trybie ortogonalnym
        //-----------------------------------------------
        glm::mat4 mOrto = glm::ortho(0.0f,float(width),
               0.0f,float(height));
        mModelView = glm::mat4(1.0);
        mTransform = glm::mat4(1.0);

        // ustaw macierz projekcji na ortogonalna
        glUniformMatrix4fv(_Projection, 1, GL_FALSE,
               glm::value_ptr(mOrto));
        // ustaw przeksztalcenia macierzowe
        glUniformMatrix4fv(_ModelView, 1, GL_FALSE,
                glm::value_ptr(mModelView*mTransform));


        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glDisable(GL_DEPTH_TEST);

    // rysowanie tekstu
        Prn->Draw('A');
        glEnable(GL_DEPTH_TEST);
        glDisable(GL_BLEND);
```

3. Skompiluj i uruchom program. Porównaj uzyskiwane efekty.

4. Wywołaj rysowanie obiektu `Moon` w trybie perspektywicznym i renderowanie tekstu w rzutowaniu ortogonalnym

5. Zwróć uwagę na problem związany z użyciem shadera fragmentów zarówno do teksturowania obiektu jak i tworzenia napisów. Wykorzystaj zmienną jednorodną `ShadingMode` wprowadzoną do shadera oraz instrukcje warunkowe, aby dla obiektów rysowanych w rzutowaniu perspektywicznym zachodziło klasyczne teksturowanie, a dla rysowania tekstu zachodziło teksturowanie z wykorzystaniem koloru tekstury jako kanału alpha w procesie blendingu.

1.2 Zapis widoku sceny do pliku

Biblioteka *OpenGL* udostępnia programiście szereg buforów przechowujących pośrednie wyniki renderowania geometrii trójwymiarowej. Wśród nich wyróżnia się bufor kolorów [3], który zawiera bieżący widok wygenerowanej sceny. Dzięki metodom programowego odczytu tablicy pikseli tworzących bufor istnieje możliwość zapisania sekwencji widoków sceny do pliku. Termin mapa bitowa pierwotnie odnosił się do zbioru bajtów, których poszczególnej bity reprezentowały stany zapalenia lub zgaszenia pikseli na ekranach/wyświetlaczach monochromatycznych. Wprowadzenie wyświetlaczy kolorowych uogólniło ten termin do postaci tablicy bajtów reprezentujących wartości (kolory) pikseli obrazu dwuwymiarowego. Plik mapy bitowej składa się z nagłówka oraz właściwej tablicy bajtów reprezentujących kolory pikseli. Nagłówek o standardowym rozmiarze 54 bajtów definiuje podstawowe ustawienia obrazu (rozmiar, kompresja, rozdzielczość, głębia kolorów. Rysunek 3 prezentuje opis pól nagłówka dla bitmapy przystosowanej do wyświetlania niezależego od urządzenia [4] Tablica pikseli występuje bezpośrednio za nagłówkiem mapy bitowej. W przypadku map 24-o bitowych kolejne trójki bajtów obrazu reprezentują nasycenie barw podstawowych B,G,R kolejnych pikseli.

Offset	Size	Hex Value	Value	Description
0h	2	42 4D	"BM"	Magic Number (unsigned integer 66, 77)
2h	4	46 00 00 00	70 Bytes	Size of the BMP file
6h	2	00 00	Unused	Application Specific
8h	2	00 00	Unused	Application Specific
Ah	4	36 00 00 00	54 bytes	The offset where the bitmap data (pixels) can be found.
Eh	4	28 00 00 00	40 bytes	The number of bytes in the header (from this point).
12h	4	02 00 00 00	2 pixels	The width of the bitmap in pixels
16h	4	02 00 00 00	2 pixels	The height of the bitmap in pixels
1Ah	2	01 00	1 plane	Number of color planes being used.
1Ch	2	18 00	24 bits	The number of bits/pixel.
1Eh	4	00 00 00 00	0	BI_RGB, No compression used
22h	4	10 00 00 00	16 bytes	The size of the raw BMP data (after this header)
26h	4	13 0B 00 00	2,835 pixels/meter	The horizontal resolution of the image
2Ah	4	13 0B 00 00	2,835 pixels/meter	The vertical resolution of the image
2Eh	4	00 00 00 00	0 colors	Number of colors in the palette
32h	4	00 00 00 00	0 important colors	Means all colors are important
Start of Bitmap Data				
36h	3	00 00 FF	0 0 255	Red, Pixel (0,1)
39h	3	FF FF FF	255 255 255	White, Pixel (1,1)
3Ch	2	00 00	0 0	Padding for 4 byte alignment (Could be a value other than zero)
3Eh	3	FF 00 00	255 0 0	Blue, Pixel (0,0)
41h	3	00 FF 00	0 255 0	Green, Pixel (1,0)
44h	2	00 00	0 0	Padding for 4 byte alignment (Could be a value other than zero)

Rysunek 3: Przykładowy plik mapy bitowej zawierającej 4 piksele.

[3] ang. *color buffer*
[4] DIB - ang. *Device Independent Bitmap*

1.2.1 Przebieg ćwiczenia

1. Zapoznaj się z treścią kodu metody `SaveAsBmp()` w pliku `scene.cpp`

   Cechą charakterystyczną dla plików DIB jest wymóg, aby każdy z wierszy obrazu był zapisany w pliku w postaci ciągu bajtów o sumarycznej liczbie podzielnej przez 4. W związku z tym definiuje się tzw. bajty wyrównujące [5] Fragment kodu

   ```
   int _width = this->width; // bitmap height
   int _height = this->height; // bitmap width
   while (_width*3 % 4) _width--; // adjust bmp width to meet 4B row padding rule
   int img_size = _width*_height*3; // image size (each pixel is coded by 3Bytes)
   ```

   zapewnia automatyczne ograniczanie szerokości obrazu odczytywanego z bufora kolorów *OpenGL* tak, aby spełnić regułę 4-o bajtowego dopełnienia[6]

2. Rozbuduj kod generujący nagłówek funkcji do postaci zgodnej z formatem BMP/DIB

3. Wprowadź kod odpowiedzialny za pobieranie zbioru pikseli z bufora kolorów *OpenGL*

   ```
   unsigned char *pixels; // room for pixeltable
   pixels = (unsigned char *) malloc(img_size*sizeof(unsigned char));

   // read pixels from colorbuffer
   glReadPixels(0,0,_width,_height,GL_BGR_EXT,GL_UNSIGNED_BYTE,pixels);
   // store pixels in the file
   fwrite(pixels,1,img_size,fil);
   fflush(fil); // clear file cache
   fclose(fil); // close the file
   free(pixels); // release memory
   ```

4. Obsłuż wywołanie funkcji w programie w odpowiedzi na zdarzenie naciśnięcia spacji. W tym celu rozbuduj kod metody `Keypressed()` klasy implementującej scenę.

5. Skompiluj i uruchom program. Wymuś kilkukrotne wyrenderowanie widoku sceny, a następnie podejrzyj zawartość wygenerowanego pliku bmp w edytorze graficznym.

6. Zmień kolejność odczytu z bufora bajtów reprezentujących piksel z `GL_BGR_EXT` na `GL_RGB`. Sprawdź jak to wpłynie na wygląd mapy bitowej

[5] ang. *padding bytes*
[6] ang. *padding rule*