Opis implementacji

Wydaje mi się, że samo wypunktowanie funkcjonalności i ich miejsc może sprawić, że ich implementacja będzie ciężka do analizy, więc opisałem z grubsza cały potok renderowania. Wymagane funkcjonalności w dokumencie pogrubiłem, by łatwo można było znaleźć ich opis (i jakie metody za nie odpowiadają)

Nie będę tutaj się skupiał na implementacji funkcjonalności pokroju dodawanie figur na scenę lub usuwanie, gdyż chyba nie to jest celem projektu.

Reprezentacja figur. Wszystkie klasy figur dziedziczą po abstrakcyjnej klasie `Figure`. Jeśli chodzi o przestrzeń modelu, klasa zawiera właściwość `int[] Triangles` oraz pola `double[,] modelNormals`, `double[,] modelBinormals`, `double[,] modelTangents` i `double[,] modelPointsCoords`. `modelPointsCoords` jest tablicą o 4 wierszach określających konkretne współrzędne punktu w przestrzeni modelu i odpowiedniej liczbie kolumn. Tablice `modelNormals`, `modelBinormals`, `modelTangents` mają 3 wiersze i taką samą liczbę kolumn jak tablica `Triangles`. W tablicy `Triangles` każde 3 kolejne elementy oznaczają 3 wierzchołki pewnego trójkąta tworzącego siatkę figury, tj. elementy o indeksach 0, 1, 2 oznaczają pierwszy trójkąt, elementy o indeksach 3, 4, 5 oznaczają kolejny trójkąt itp. Ogólnie, `(i%3)`-ty wierzchołek z `(i/3)`-tego trójkąta ma współrzędną `modelPointsCoords[Triangles[i]]`. Ponadto, takiemu wierzchołkowi odpowiadają `i`-te elementy z tablic z wektorami (tzn. `i`-te kolumny). Zauważmy, że w ten sposób, każdy wierzchołek ma własny wektor normalny, binormalny, styczny (i każdemu wektorowi odpowiada konkretny wierzchołek z konkretnego trójkąta), ale za to jedna współrzędna punktu może odpowiadać kilku wierzchołkom z kilku trójkątów (wtedy, gdy w tablicy `Triangles` pewien indeks występuje kilkukrotnie). Zaoszczędzam zawsze trochę czasu przy późniejszym mnożeniu macierzy `modelPointsCoords` (nie mam zduplikowanych współrzędnych, czyli nie mnożę tej samej rzeczy parę razy). I tak np. w prostopadłościanie `modelPointsCoords` ma 8 kolumn (tyle jest różnych współrzędnych wierzchołków), zaś wszystkie pozostałe tablice (`Triangles` i tablice z wektorami) mają $6 \cdot 2 \cdot 3$ kolumn (6 ścian, na każdej 2 trójkąty, każdy 3 wierzchołki).

W klasie `Figure` zdefiniowane są też analogiczne właściwości dla przestrzeni świata - `double[,] WorldPointsCoords`, `double[,] WorldNormals`, `double[,] WorldBinormals` i `double[,] WorldTangents`. Teraz przejdę do ich wyliczania. Figury posiadają szereg właściwości do transformacji. Atrybuty dla przestrzeni świata są wyliczane po zmianie jakiegokolwiek parametru transformacji (i oczywiście w konstruktorach klas pochodnych). Wywoływana jest wówczas metoda `void Figure.UpdateModelMatrixAndWorldsData()`, która z kolei wywołuje kolejno `void Figure.UpdateModelMatrix()` i `void Figure.UpdateWorldData()`. **W metodzie `Figure.UpdateModelMatrix()` wyliczana jest macierz modelu i macierz normalna** (do przekształceń wektorów). W metodzie `Figure.UpdateWorldData()` dokonywane jest faktyczne **wyliczanie atrybutów w przestrzeni świata** (współrzędnych i wektorów) przez mnożenie odpowiednich macierzy i normalizację wektorów (w szczególności wektory mnożę przez macierz normalną). Zaznaczę jeszcze, że w tej klasie znajduje się abstrakcyjna metoda `abstract void Figure.UpdateTriangles()`.

Opisałem mniej więcej klasę abstrakcyjną, teraz jeszcze krótko o klasach pochodnych. Weźmy przykład klasy `Sphere` (w pozostałych klasach analogiczna implementacja). Poza

właściwościami z klasy bazowej, ma ona właściwości związane z konkretną figurą, czyli w tym przypadku promień, liczbę podziałów siatki w pionie i w poziomie. Zmiana którejkolwiek z tych właściwości wywołuje metodę `void UpdateTrianglesAndWorldsData()` (jest ona też wywoływana w konstruktorze), która wywołuje kolejno `void UpdateTriangles()` i `void UpdateWorldData()`. `UpdateTriangles()` służy wypełnieniu tablic: z indeksami współrzędnych `Triangles`, ze współrzędnymi `modelPointsCoords`, wektorami (`modelNormals`, …) i współrzędnymi tekstury (`TextureCoords`) dla każdego wierzchołka każdego trójkąta w przestrzeni modelu. To w tej metodzie (`void UpdateTriangles()`) **figura dzielona jest na trójkąty**. Nie będę się wgłębiał w szczegóły tych metod, bo są one mało ciekawe, można sprawdzić ich efekty przełączając daną figurę w tryb wypełnienia tylko krawędziami trójkątów. W tej metodzie dbam o poprawną orientację tworzonych trójkątów (clockwise). O `UpdateWorldData()` już wspominałem i tylko przypomnę, że w niej wyliczane są atrybuty przestrzeni świata mnożąc odpowiednie macierze.

Kolejny krok, czyli **pomnożenie współrzędnych przez macierze *PV*** jest dokonywany w metodzie `void pictureBox_Paint(object sender, PaintEventArgs e)` klasy `MainForm`. Wyliczenie macierzy realizowane jest w tych instrukcjach:

```
var P = currentCamera.GetProjectionMatrix((double)pictureBox.Width /
      pictureBox.Height)
var V = currentCamera.GetViewMatrix();
var PV = Utils.MultiplyMatrices(P, V);
```

(można zajrzeć do odpowiednich metod klasy `Camera` i tam zweryfikować **tworzenie macierzy projekcji i widoku**) i potem w pętli dla każdej figury jest obliczane:

```
var clippingSpaceCoords =
      Utils.MultiplyMatrices(PV, figure.WorldPointsCoords);
```

Kolejnym krokiem jest obcinanie algorytmem Sutherlanda-Hodgmana:

```
var clippedTriangles = SutherlandHodgman.ClipTriangles(
     clippingSpaceCoords, figure);
```

Obcinanie zaimplementowane jest w klasie statycznej `SutherlandHodgman`. Statyczna metoda do obcinania ma nagłówek:

```
static List<Triangle> ClipTriangles(double[,] pointCoords, Figure figure)
```

Implementacja obcinania jest identyczna jak ta ze slajdów wykładowych z tą różnicą, że w przypadku szukania przecięcia, poza znalezieniem współrzędnych obcinania przecięcia znajdujemy też inne atrybuty punktu przecięcia (wektory, współrzędne świata i współrzędne tekstury). Oczywiście, wzorowałem się też na informacjach z pdfa do projektu, czyli obcinam po kolei względem sześciu płaszczyzn i w metodzie `static bool IsInside(double[] coords, int plane_index)` wykorzystuję indeks płaszczyzny do wyznaczenia odległości od konkretnej płaszczyzny (jeśli jest nieujemna to `IsInside` zwraca `true`). Przy obcinaniu dbam o to, by zachować odpowiednią orientację trójkątów (ze względu na backface culling).
Metoda `SutherlandHodgman.ClipTriangles` zwraca listę obiektów klasy `Triangle`. Pojedynczy taki obiekt zawiera 3-elementowe tablice współrzędnych w przestrzeni obcinania, świata, wektorów normalnych, binormalnych, stycznych i współrzędnych tekstury.
Wracając do metody `MainForm.pictureBox_Paint(…)` kolejnym krokiem jest już właściwie rasteryzacja, czyli wywołanie dla każdego trójkąta:

```
clippedTriangles[i].Draw(bitmap,
```

```
backfaceCullingOn, zBufferArray, perspectiveCorrectionOn,
lights, currentCamera.P);
```

, gdzie `lights` jest listą wszystkich świateł na scenie (być może `null` w przypadku wyłączonego modelu oświetlenia), zaś `currentCamera.P` jest pozycją kamery, którą obserwujemy scenę. Tablica `zBufferArray` jest `null` w przypadku wyłączonego buforowania głębi lub ma wymiary bitmapy i jest zainicjowana dziesiątkami w przypadku włączonego buforowania głębi. W metodzie `Draw(…)` następuje wybór konkretnej metody rasteryzacji – rysowania krawędzi Bresenhamem lub też wypełniania algorytmem scanlinii. W obu przypadkach, przed rasteryzacją wyznaczam współrzędne w przestrzeni ekranu wierzchołków trójkąta metodą `VertexData[] GetVertexDatas(int width, int height)`. `VertexData` jest klasą, która przechowuje wszystkie niezbędne mi atrybuty w wierzchołku (czyli współrzędne ekranu, świata, współrzędną w_c przestrzeni obcinania do ewentualnej korekcji perspektywy, wektory, itp.). Zarówno w przypadku rysowania krawędzi trójkąta Bresenhamem, jak i wypełniania sprawdzamy, czy mamy włączone obcinanie ścian tylnych, ewentualnie sprawdzamy orientację trójkąta i w razie czego wychodzimy z funkcji. Czyli **obcinanie ścian tylnych** jest zaimplementowane na samym początku metod: `Triangle.DrawBorders(…)` i `Triangle.FillTriangle(…)`.W przypadku, gdy figura ma być narysowana Bresenhamem, wywoływana jest metoda `Triangle.DrawBorders(DirectBitmap bitmap, bool backfaceCulling, double[,] zBuffer)`. W tej metodzie rasteryzujemy wszystkie 3 krawędzie trójkąta korzystając z klasy statycznej `Bresenham` z pierwszego projektu wzbogaconej o możliwość buforowania głębi (jeśli tablica z zBuforem nie jest `null`). Czyli **w przypadku Bresenhama, implementacja buforowania głębi** znajduje się w przeciążonych metodach w klasie statycznej `Bresenham`.
Teraz skupię się na opisaniu metody

```
Triangle.FillTriangle(DirectBitmap bitmap,
     bool backfaceCulling, double[,] zBuffer,
     bool perspectiveCorrection,
     List<Light> lights, double[] cameraCoords)
```

Metoda ta używana jest do **wypełniania algorytmem scanlinii**, zarówno przy włączonym świetle, jak i wyłączonym, przy włączonej lub wyłączonej korekcji perspektywy i przy włączonym lub wyłączonym buforowaniu głębi. Scanlinię zaimplementowałem zoptymalizowaną dla trójkątów (czyli algorytm z pdfa do projektu). W zależności od zestawu włączonych opcji, przy scanlinii konieczne może być interpolowanie innych atrybutów (np. bez włączonego światła nie musimy interpolować wektorów, przy włączonym świetle i wypełnieniu jednym kolorem musimy interpolować wektor normalny, zaś przy włączonym świetle i wypełnieniu teksturą musimy interpolować wszystkie 3 wektory oraz współrzędne tekstury). Z tego powodu, stworzyłem interfejs `IVerticalInterpolationData` reprezentujący interpolowane dane na lewym lub prawym odcinku (na tym odcinku, na którym poruszamy się w pionie, inkrementując lub dekrementując współrzędną y) oraz interfejs `IHorizontalInterpolationData` (czyli interpolowane dane poruszając się po konkretnej poziomej scanlinii). Interfejsy te znajdują się w klasie `Triangle`.W szczególności, obiekty implementujące te interfejsy mają metody `Increment` oraz `HasFinished`, co umożliwiło mi łatwe stworzenie pętli w algorytmie wypełniania (np. dla obiektów `IHorizontalInterpolationData`, `Increment()` będzie oznaczać inkrementację współrzędnej x (i ewentualnych innych atrybutów zależnych od konkretnej klasy)). Wracając do metody `Triangle.FillTriangle(…)`, odpowiednie obiekty `IVerticalInterpolationData` otrzymuję przez wywołanie metody `IVerticalInterpolationData GetVerticalInterpolationData(…)` i tu można sprawdzić, że w zależności od włączonych świateł oraz sposobu wypełnienia wybierany

zwracany jest odpowiedni obiekt. Nie będę się wgłębiał tutaj bardzo w szczegóły tych obiektów, ważne jest to, że każdy dba o określone inkrementowane atrybuty. Ponadto, obiekty `IHorizontalInterpolationData` posiadają metodę `Color CalculateColor()`, która służy do policzenia koloru w konkretnym pikselu, w którym taki obiekt się aktualnie znajduje. Jeśli chodzi o `IVerticalInterpolationData`, to u mnie rolę odgrywają 4 klasy:

- `VerticalBasicInterpolation` - klasa do wypełniania bez oświetlenia obiektów jednokolorowych,
- `VerticalTextureInterpolation` - klasa do wypełniania bez oświetlenia obiektów oteksturowanych,
- `VerticalPhongInterpolation` - klasa do wypełniania z oświetleniem obiektów jednokolorowych,
- `VerticalTexturePhongInterpolation` - klasa do wypełniania z oświetleniem obiektów oteskturowanych.

Wewnątrz każdej z tych klas zdefiniowana jest odpowiadająca klasa implementująca `IHorizontalInterpolationData`. **Cieniowanie Phonga** jest realizowane w klasach `VerticalPhongInterpolation` i `VerticalTexturePhongInterpolation` jako interpolowanie odpowiednich atrybutów oraz oczywiście w ich wewnętrznych klasach `HorizontalPhongInterpolation` i `HorizontalTexturePhongInterpolation` odpowiadających interpolowaniu w poziomie. W szczególności, **mapowanie normalnych** znajdziemy w metodzie `CalculateColor()` w klasie `HorizontalTexturePhongInterpolation`, która napisana jest wewnątrz klasy `VerticalTexturePhongInterpolation`. Samo wyliczanie koloru w **modelu Phonga** jest napisane w statycznej klasie `PhongLightingModel` (w implementacji pojawia się `Math.Max(0d,…)` w przypadku iloczynów skalarnych, co wzorowałem na stronie https://learnopengl.com/ i jest to konieczne, by np. światło nie działało na część figury odwróconą tyłem do źródła). Tam też jest oczywiście uwzględniony **zasięg światła**.

Wracając do pętli w algorytmie wypełniania w metodzie `Triangle.FillTriangle(…)`, przed pokolorowaniem piksela znajduje się oczywiście **sprawdzenie zBufora**:

```
if (zBuffer == null || horizontalData.Z <=
    zBuffer[horizontalData.X, horizontalData.Y])
{
    bitmap.SetPixel(horizontalData.X, horizontalData.Y,
        horizontalData.CalculateColor());
    if (zBuffer != null)
        zBuffer[horizontalData.X, horizontalData.Y] =
              horizontalData.Z;
}
```

Jest jeszcze jedna bardzo ważna rzecz, którą muszę podkreślić odnośnie algorytmu scanlinii. Ja stosuję interpolację liniową i ma ona oczywiście różny wzór w zależności od tego, czy korekcja perspektywy jest włączona. Z tego powodu, tutaj również zrobiłem ogólny interfejs `Triangle.ILinearIntepolation`, który służy interpolacji różnych atrybutów (ma metody do interpolowania różnych typów). Nie chciałem przy tym za każdym wywołaniem interpolowania wyliczać na nowo od zera współczynników we wzorach w interpolacji, dlatego obiekty `Triangle.ILinearIntepolation` mają metodę `void IncrementDistance()`, która służy zwiększeniu odległości od punktu, w którym zaczynaliśmy o 1 (przy czym ta odległość niekoniecznie jest Euklidesowa, w scanlinii dla lewego i prawego odcinka odległość jest wyrażona w odległości po y, zaś już przy samej poziomej scanlinii wyrażona jest oczywiście jako odległość po x) i tym samym odpowiednim inkrementacyjnym zaktualizowaniu współczynników w interpolacji (np. zaktualizowaniu q i $1-q$ w przypadku interpolacji bez korekcji perspektywy). Konkretnie, za **interpolację z korekcją perspektywy** odpowiada

klasa `LinearIntepolationWithPC` (cały czas znajdujemy się wewnątrz klasy `Triangle`, czyli w pliku `Triangle.cs`).
Ogólnie klasa `Triangle` jest u mnie bardzo rozbudowana (tak jak widać, tu się dzieje w zasadzie cały ostatni etap rasteryzacji). Nie chciałem w tym opisie za bardzo wnikać w szczegóły, mam nadzieję, że dobrze opisałem idee, ewentualnie można bardziej dokładnie przejrzeć interpolowanie atrybutów dla konkretnych klas `IVerticalInterpolationData` i ich wewnętrznych `IHorizontalInterpolationData`.

Opiszę krótko **poruszanie kamery**. W przypadku scrollowania, mamy do czynienia z metodą `pictureBox_MouseWheel` w klasie `MainForm`. Tutaj, w zależności od wciśnięcia lub nie LSHIFT wywoływana jest odpowiednia metoda klasy `Camera`. Podobnie w przypadku przesuwania myszką po ekranie, wywołuje się metoda `pictureBox_MouseMove` i w niej podobnie w zależności od LSHIFT wywoływana jest odpowiednia metoda klasy `Camera`. Przy zmianie odległości pozycji kamery od punkty, na który patrzę (scrollowanie z shiftem), czyli w metodzie `void Camera.ChangeDistanceFromTarget(int offset)` korzystam ze współrzędnych biegunowych. Podobnie przy obracaniu `void Camera.Rotate(double horAngle, double verAngle)` korzystam ze współrzędnych biegunowych. Oczywiście pamiętam, że po obróceniu się kamerą należy zaktualizować jej charakterystyczne wektory metodą `void Camera.UpdateVectors()`. Samych metod, które przesuwają kamerę/obracają nie będę tutaj opisywał, bo wydaje mi się, że są w miarę przystępnie napisane w kodzie.

Zliczanie FPS – za każdym razem, gdy pictureBox jest przerysowywany inkrementowana jest zmienna `framesCount`. Ponadto, w tle działa też drugi timer, który co sekundę (w praktyce przy dużym obciążeniu może trochę więcej niż sekundę) odpala metodę `void fpsTimer_Tick(object sender, EventArgs e)`, która uaktualnia stan napisu z liczbą klatek na sekundę oraz zeruje licznik.

Zapisywanie/wczytywanie z pliku zrobiłem jako serializację binarną. Mam pomocniczą klasę `SceneData`, która zawiera wszystkie informacje serializowane do pliku. Warto tutaj zaznaczyć, że bitmapy z teksturami nie są serializowane. Serializowana jest ścieżka do nich i ze względu na przenośność pliku ze sceną, przy wczytywaniu tekstur zmieniam stringa ze ścieżką bezwzględną na ścieżkę względną (przy czym zakładam, że tekstury wczytuje się z folderu Textures projektu). Klasa `Figure` ma metodę oznaczoną `[OnDeserialized]`, która po deserializacji próbuje załadować tekstury z zapisanych ścieżek. Dbam też o wyjątki, gdyby coś się jednak nie udało.