

Politechnika Świętokrzyska w Kielcach

Wydział Elektroniki, Automatyki i Informatyki

Projekt: Algorytmy grafiki komputerowej – projekt

Dokumentacja Techniczna

Temat numer: 9
Symulacja przezroczystości na przykładowej
scenie 3D (np. budynek z oknami, witryny
sklepowe, pojazdy).

Grupa: 1ID23A
Adrian Chmielowiec
90092

1. Temat projektu

Tematem projektu jest symulacja przezroczystości na przykładowej scenie 3D.

2. Opis zastosowanych bibliotek i algorytmów

W projekcie wykorzystano następujące biblioteki:

- Biblioteka Assimp została użyta do wczytywania modeli 3D. Pozwala na łatwe importowanie modeli do programu i manipulację nimi.
- Biblioteka Glad zapewnia interfejs do ładowania funkcji OpenGL. Umożliwia prostą integrację programu z OpenGL i zapewnia dostęp do wszystkich potrzebnych funkcji tej biblioteki.
- Biblioteka Glfw dostarcza narzędzia do tworzenia i zarządzania oknami, obsługi zdarzeń, wejścia od użytkownika (klawiatura, mysz) oraz kontekstu OpenGL. Ułatwia inicjalizację i obsługę okien oraz interakcję użytkownika z programem.
- Biblioteka Glm jest matematycznym narzędziem dla OpenGL, zapewniającym funkcje i typy danych do operacji na macierzach, wektorach i transformacjach. Ułatwia obliczenia związane z grafiką 3D, takie jak przekształcenia, projekcje, obliczenia oświetleniowe itp.
- Biblioteka Stbimage jest używana do ładowania tekstur z plików obrazowych. Zapewnia prosty sposób wczytywania tekstur i ich przetwarzania w programie.

Ponadto, w projekcie zastosowano algorytm sortowania obiektów na scenie, wykorzystujący blending. Algorytm ten pozwala na odpowiednie renderowanie obiektów transparentnych, tak aby były widoczne poprzez inne elementy sceny. Dzięki temu efektowi przezroczystości można otrzymać poprawne wyświetlanie okien, umożliwiające podglądanie obiektów znajdujących się za nimi.

Algorytm sortowania:

1. Zdefiniuj pozycje okien
2. Narysuj najpierw wszystkie okna obiekty.
3. Oblicz odległość od kamery dla każdego okna.
4. Sortowanie z wykorzystaniem mapy według odległości od kamery
5. Renderowanie okien

3. Opis implementacji z fragmentami kodu źródłowego

Przedstawię tylko najważniejsze fragmenty dotyczące programu.

Pierwszym z nich jest fragment odpowiedzialny za główny temat projektu czyli symulacja przezroczystości. Zrealizowana została przy pomocy włączenia blendingu, odpowiedniego rysowania obiektów i sortowania.

```
1. ...
2.     std::vector<glm::vec3> windows
3.     {
4.
5.         glm::vec3(-24.5f, 34.0f, -12.0f),
6.         glm::vec3(-55.0f, 34.0f, -12.0f),
7.         glm::vec3(-39.5f, 50.0f, -12.0f),
8.         glm::vec3(-34.5f, 17.5f, -12.0f)
9.     };
10.
11.     ...
12.
13.     //mapa do przechowywania okien w odpowiedniej kolejności
14.     std::map<float, glm::vec3> sorted;
15.
16.     ...
17.
18.     //Petla glowna
19.     while (!glfwWindowShouldClose(window))
20.     {
21.         //Pobieranie czasu ostatniej klatki
22.         float currentFrame = static_cast<float>(glfwGetTime());
23.         deltaTime = currentFrame - lastFrame;
24.         lastFrame = currentFrame;
25.
26.         //input
27.         mappingInput(window);
28.
29.
30.         //czyszczenie okna kolorem
31.         glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
32.         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
33.
34.         //Aktywacja tekstur
35.         texture4.useTexture();
36.         for (unsigned int i = 0; i < trees.size(); i++)
37.         {
38.             tree.setModelMatrix(glm::mat4(1.0f));
39.             tree.translate(trees[i]);
40.             //Ustawienie uniformow i shadera
41.             tree.setShaderUniforms(LightingObject.getShaderProgram(),
camera.getViewMatrix(), camera.getProjectionMatrix(), glm::vec3(1.0f, 1.0f, 1.0f),
glm::vec3(1.0f, 1.0f, 1.0f), lightPos, camera.getCameraPosition());
42.             tree.draw();
43.         }
44.
45.
46.
47.         //Ustawienie uniformow i shadera
48.         house.setShaderUniforms(LightingObject.getShaderProgram(), camera.getViewMatrix(),
camera.getProjectionMatrix(), glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 1.0f, 1.0f), lightPos,
camera.getCameraPosition());
49.         texture5.bindTexture();
50.         house.draw();
51.
52.         //Ustawienie uniformow i shadera
```

```

53.         floor.setShaderUniforms(LightingObject.getShaderProgram(), camera.getViewMatrix(),
camera.getProjectionMatrix(), glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 1.0f, 1.0f), lightPos,
camera.getCameraPosition());
54.         texture6.bindTexture();
55.         floor.draw();
56.
57.
58.         //Dla slonca
59.         sun.setModelMatrix(glm::mat4(1.0f));
60.         sun.scale(glm::vec3(0.5f));
61.         sun.translate(lightPos);
62.         sun.setShaderUniforms(SunObject.getShaderProgram(), camera.getViewMatrix(),
camera.getProjectionMatrix(), glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 1.0f, 1.0f), lightPos,
camera.getCameraPosition());
63.         sun.draw();
64.
65.         sorted.clear();
66.         // sortowanie okien przed renderowaniem tak aby były renderowane od najdalszych do
najbliższych
67.         for (unsigned int i = 0; i < windows.size(); i++)
68.         {
69.             // obliczamy odległość od kamery poprzez obliczenie długości wektora między
pozycją kamery a okna
70.             float distance = glm::length(camera.getCameraPosition() - windows[i]);
71.             // przypisanie tej odległości jako klucz do mapy a jako wartość pozycje okna
72.             sorted[distance] = windows[i];
73.         }
74.
75.         texture3.bindTexture();
76.         // iteracja przez posortowaną mapę, która przechowuje odległość od kamery do okien i
renderowanie w odwrotnej kolejności zaczyna od najdalszych i kończy na najbliższych
77.         for (std::map<float, glm::vec3>::reverse_iterator it = sorted.rbegin(); it !=
sorted.rend(); ++it)
78.         {
79.             window_object.setModelMatrix(glm::mat4(1.0f));
80.             window_object.scale(glm::vec3(0.15f));
81.             window_object.rotate(glm::radians(90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
82.             // ustawiamy macierz dla obiektu przesuwając go do położenia w przestrzeni
świata
83.             window_object.translate(it->second);
84.             // Sprawdzamy, czy to jest okno, które chcemy obrócić
85.             if (it->second == windows[0] || it->second == windows[1]) {
86.                 // Obrót okna o 90 stopni wokół osi Y
87.                 window_object.rotate(glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
88.             }
89.             // przesyłanie macierzy model do shadera
90.             window_object.setShaderUniforms(LightingObject.getShaderProgram(),
camera.getViewMatrix(), camera.getProjectionMatrix(), glm::vec3(1.0f, 1.0f, 1.0f),
glm::vec3(1.0f, 1.0f, 1.0f), lightPos, camera.getCameraPosition());
91.             window_object.draw();
92.         }
93.
94.
95.         //Podwójne buforowanie
96.         glfwSwapBuffers(window);
97.         //Funkcja do sprawdzenia czy zostało wykonane jakieś wydarzenie i aktualizowania
stanu okna
98.         glfwPollEvents();
99.     }
100.
101. ...

```

Powyższy fragment kodu rozpoczyna się od ustalenia pozycji okien na scenie 3D. Pozycje są zapisywane w wektorze, a następnie tworzona jest mapa, która przechowuje odległość od kamery jako klucz i pozycję okna na scenie jako wartość. Dzięki automatycznemu sortowaniu mapy po wartościach klucza, nie ma potrzeby dodatkowego sortowania obiektów.

W pętli głównej programu, po części odpowiedzialnej za pomiar czasu, następuje proces renderowania nieprzezroczystych obiektów. Każdy obiekt jest reprezentowany przez instancję klasy Object, która wczytuje model obiektu i przekazuje odpowiednie dane do shadera dotyczące tego obiektu. Klasa Object również wykonuje przekształcenia oraz rysowanie obiektu.

Od liniiki 65 znajduje się fragment kodu, który oblicza odległość od kamery dla każdego okna i zapisuje te wartości do mapy. Następnie, w pętli renderowania, okna są renderowane zaczynając od tych znajdujących się najdalej od kamery i kończąc na najbliższych. Okna są również poddawane transformacjom, które lepiej dopasowują je do sceny.

Drugą rzeczą która miała znaleźć się w programie jest oświetlenie. Zostało zastosowane oświetlenie phonga. Poniżej znajduje się kod vertex shadera oraz fragment shadera, który realizowany jest na obiektach na scenie:

```
1. #version 330 core
2. layout (location = 0) in vec3 aPos;
3. layout (location = 1) in vec3 aNormal;
4. //layout (location = 1) in vec3 aColor;
5. layout (location = 2) in vec2 aTexCoord;
6.
7. //out vec3 ourColor;
8. out vec2 TexCoord;
9. out vec3 ourNormal;
10. out vec3 ourFragPos;
11.
12. uniform mat4 model;
13. uniform mat4 view;
14. uniform mat4 projection;
15. //uniform mat4 transform;
16.
17. void main()
18. {
19.     //ourColor = aColor;
20.     TexCoord = aTexCoord;
21.     //Pozycja fragmentu (wspolrzedne w przestrzeni swiata)
22.     ourFragPos = vec3(model * vec4(aPos, 1.0));
23.     //ourNormal = aNormal;
24.     //Macierz normalna
25.     ourNormal = mat3(transpose(inverse(model))) * aNormal;
26.
27.     gl_Position = projection * view * model * vec4(aPos, 1.0);
28. }
```

```
1. #version 330 core
2. out vec4 FragColor;
3.
4. //in vec3 ourColor;
5. in vec2 TexCoord;
6. in vec3 ourNormal;
7. in vec3 ourFragPos;
8.
9. uniform sampler2D texture1;
10. //uniform sampler2D texture2;
11.
12. uniform vec3 objectColor;
13. uniform vec3 lightColor;
14. uniform vec3 lightPos;
15. uniform vec3 viewPos;
16.
17. void main()
18. {
19.
20.     vec4 texColor = texture(texture1, TexCoord);
21. }
```

```

22.     float ambientStrength = 0.1;
23.     vec3 ambient = ambientStrength * lightColor;
24.
25.     //wektor kierunku miedzy zrodlem swiatla a pozycja fragmentu
26.     //normalizacja w celu osiagniecia wektora jednostkowego
27.     vec3 norm = normalize(ourNormal);
28.     vec3 lightDir = normalize(lightPos - ourFragPos);
29.     //obliczenie wplywu swiatla na biezacy fragment
30.     //iloczyn skalarny miedzy wektorami (max by nigdy nie bylo ujemne)
31.     float diff = max(dot(norm, lightDir), 0.0);
32.     vec3 diffuse = diff * lightColor;
33.
34.
35.     //Intensywnosc odblyskow
36.     float specularStrength = 0.9;
37.     //Obliczenie wektora kierunku widoku i wektor odbicia wzdloz normalnej
38.     vec3 viewDir = normalize(viewPos - ourFragPos);
39.     //negujemy poniewaz oczekujemy ze wektor bedzie wskazywal od zrodla swiatla
40.     vec3 reflectDir = reflect(-lightDir, norm);
41.     //Obliczanie składowej zwierciadlanej
42.     //mnozenie składowej kierunku patrzenia i odbicia i podnoszenie do potegi
32(liczba to oznacza wartosc polysku)
43.     float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
44.     vec3 specular = specularStrength * spec * lightColor;
45.
46.
47.     // vec3 result = (ambient + diffuse + specular) * objectColor;
48.     vec3 result = (ambient + diffuse + specular) * objectColor;
49.     FragColor = texColor * vec4(result,1.0) ;
50. }

```

Do wczytywania modeli została zastosowana biblioteka assimp i klasa Object.

```

1. #pragma once
2. #include <glad/glad.h>
3. #include <glm/glm.hpp>
4. #include <glm/gtc/matrix_transform.hpp>
5. #include <glm/gtc/type_ptr.hpp>
6. #include <string>
7. #include <iostream>
8. #include <vector>
9. #include <ObjLoader.h>
10.
11. #include <assimp/Importer.hpp>
12. #include <assimp/scene.h>
13. #include <assimp/postprocess.h>
14.
15. // Definicja struktury Vertex, przechowującej informacje o wierzchołkach obiektu
16. struct Vertex {
17.     glm::vec3 Position;    // Pozycja w przestrzeni 3D
18.     glm::vec3 Normal;     // Wektor normalny wierzchołka
19.     glm::vec2 TexCoords;  // Współrzędne tekstury
20. };
21.
22. // Klasa Object reprezentująca obiekt na scenie
23. class Object
24. {
25. public:
26.     // Konstruktor przyjmujący ścieżkę do pliku obiektu
27.     Object(const std::string& path);
28.
29.     // Identyfikatory buforów OpenGL
30.     GLuint VAO, VBO, EBO;
31.     // Wektor wierzchołków obiektu
32.     std::vector<Vertex> vertices;
33.     // Liczba wierzchołków
34.     size_t vertexCount;
35.     // Wektor indeksów wierzchołków dla renderowania obiektu

```

```

36.     std::vector<unsigned int> indices;
37.     // Macierz modelu
38.     glm::mat4 modelMatrix;
39.
40.     // Funkcja wczytująca model obiektu z pliku
41.     bool loadModel(const std::string& path);
42.     // Funkcja przetwarzająca siatkę (mesh) obiektu
43.     void processMesh(const aiMesh* mesh);
44.     // Inicjalizacja buforów OpenGL
45.     void initBuffers();
46.     // Ustawianie wartości uniformów w shaderze
47.     void setShaderUniforms(GLuint shaderProgram, const glm::mat4& viewMatrix, const
glm::mat4& projectionMatrix, const glm::vec3& objectColor, const glm::vec3& lightColor, const
glm::vec3& lightPos, const glm::vec3& viewPos);
48.     // Renderowanie obiektu
49.     void draw();
50.     // Ustawianie macierzy modelu
51.     void setModelMatrix(const glm::mat4& model);
52.     // Przesunięcie obiektu
53.     void translate(const glm::vec3& translation);
54.     // Obrót obiektu
55.     void rotate(float angle, const glm::vec3& axis);
56.     // Skalowanie obiektu
57.     void scale(const glm::vec3& scale);
58. };

```

Plik nagłówkowy klasy Object zawiera informacje o wierzchołkach dla wczytywanego modelu. Struktura klasy przechowuje również bufor VAO (Vertex Array Object), VBO (Vertex Buffer Object) i EBO (Element Buffer Object), a także macierz modelu. Kluczowe metody dostępne w klasie Object obejmują:

1. Metoda wczytywania modelu z pliku: Ta metoda umożliwia wczytanie modelu z pliku do struktury danych w klasie Object, w tym informacji o wierzchołkach, teksturach, normalnych, itp.
2. Metoda przetwarzania siatki obiektu: Ta metoda pozwala na przetworzenie danych dotyczących siatki obiektu, takich jak obliczenie normalnych, obliczenie tangensów, generowanie buforów itp.
3. Metoda inicjalizacji buforów: Ta metoda służy do inicjalizacji buforów VAO, VBO i EBO oraz przekazania do nich odpowiednich danych.
4. Metoda ustawiania wartości uniformów: Ta metoda umożliwia ustawienie wartości uniformów w shaderze, takich jak macierz projekcji, macierz widoku, światło, tekstury, itp.
5. Metoda renderowania obiektów
6. Metody transformacji obiektu: Klasa Object udostępnia różne metody do manipulowania obiektem, takie jak skalowanie, obracanie, przesuwanie, co pozwala na dokonywanie różnych transformacji na obiekcie.

Dzięki tym funkcjom klasa Object umożliwia efektywne zarządzanie obiektami, wczytywanie ich z plików, renderowanie ich na scenie oraz wykonywanie różnych transformacji w prosty i intuicyjny sposób.

Ostatnią jedną z ważniejszych utworzonych klas jest klasa do zarządzania teksturami.

```

1. #pragma once
2.
3. #include <glad/glad.h>
4. #include <string>
5. #include <iostream>
6.
7. //Klasa dla tekstur
8. class Texture
9. {

```

```

10. private:
11.     //Id tekstury
12.     GLuint textureID;
13.     //Parametry tekstury
14.     int width, height, channels;
15.     //Dane załadowanej tekstury
16.     unsigned char* texData;
17. public:
18.     //Konstruktor do ustawienia zmiennych
19.     Texture();
20.     //Konstruktor, który ładuje teksturę o wskazanej ścieżce
21.     Texture(const std::string_view& path);
22.     //Destruktor
23.     ~Texture();
24.     //Metoda do pobrania id tekstury
25.     GLuint getTextureID() const;
26.     //Metoda do generowania id tekstur i ustawieniu identyfikatora
27.     void genTextures();
28.     void bindTexture() const;
29.     //Metoda do aktywacji tekstury
30.     void useTexture() const;
31.     //Metoda do aktywacji tekstury
32.     void useTexture(GLuint textureNum) const;
33.     //Metoda do deaktywacji tekstury
34.     void unbind() const;
35. };
36.

```

Powyżej znajduje się plik nagłówkowy klasy Texture, która odpowiada za obsługę tekstur w programie. Klasa ta zawiera prywatne pola, w tym identyfikator tekstury (textureID), parametry tekstury (szerokość, wysokość, liczba kanałów) oraz dane tekstury (texData), które zostaną załadowane.

Klasa Texture dostarcza również odpowiednie metody, które umożliwiają obsługę tekstur. Metoda genTextures() generuje identyfikator tekstury i przydziela mu miejsce w pamięci. Konstruktor z parametrem path umożliwia ładowanie tekstur z określonej ścieżki. Metoda bindTexture() przypisuje utworzoną teksturę do bieżącego kontekstu OpenGL, a metoda useTexture() aktywuje tę teksturę. Istnieje również metoda useTexture(textureNum), która aktywuje teksturę o określonym numerze. Natomiast metoda unbind() służy do deaktywacji tekstury.

Dzięki tym funkcjom klasa Texture umożliwia kompletną obsługę tekstur w programie, włączając w to generowanie identyfikatorów, ładowanie z pliku, aktywację i deaktywację, co jest niezbędne do poprawnego renderowania obiektów z teksturami.

4. Wyniki testów

Testy przeprowadzone na programie wykazały jego poprawne i stabilne działanie. Wszystkie aspekty renderowania sceny, takie jak przezroczystość okien, tekstury i oświetlenie, są zachowane zgodnie z oczekiwaniami. Użytkownik może swobodnie poruszać się po scenie bez żadnych problemów.

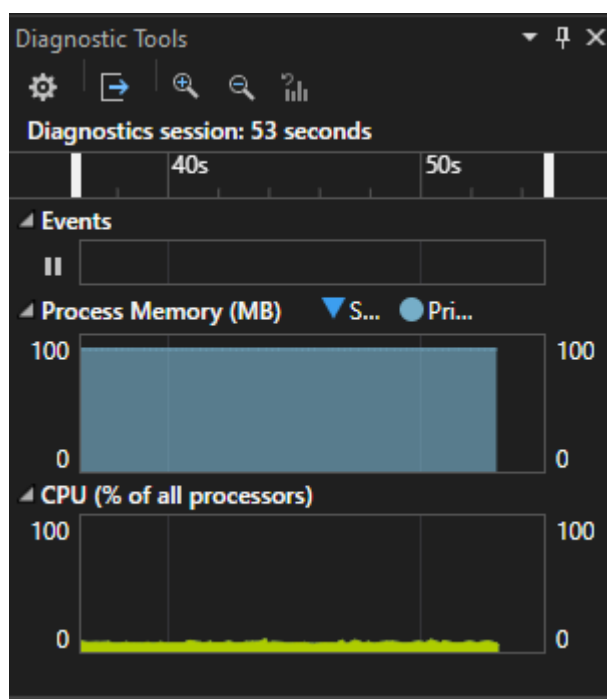
Rysunek 1 przedstawia wyniki testów dotyczących zużycia pamięci RAM oraz procentowego użycia procesora podczas 50-sekundowej sesji w programie. Możemy zauważyć, że program wymaga bardzo niewielkiej ilości zasobów i wykresy są stabilne oraz stałe przez cały okres testu.

Rysunek 2 prezentuje wykresy dotyczące zużycia karty graficznej (w tym przypadku RTX 3070). Widzimy, że akcelerator 3D jest wykorzystywany na poziomie około 10%. Nagły wzrost na początku i końcu testu wynika jedynie z uruchomienia i zakończenia programu.

Ostatni rysunek pokazuje liczbę klatek na sekundę (FPS), która została zmierzona za pomocą biblioteki chrono w języku C++. Każda klatka jest mierzona, obliczając czas renderowania, a następnie aktualizując wartości najwyższego, najniższego i całkowitego FPS oraz zwiększając licznik klatek. Po zakończeniu renderowania obliczana jest średnia ilość FPS, dzieląc sumę wszystkich FPS przez liczbę klatek.

Program osiągnął średnią ilość klatek na poziomie 63, co jest wynikiem bardzo dobrym, zwłaszcza biorąc pod uwagę ograniczenie do częstotliwości odświeżania monitora 60 Hz przez vsync.

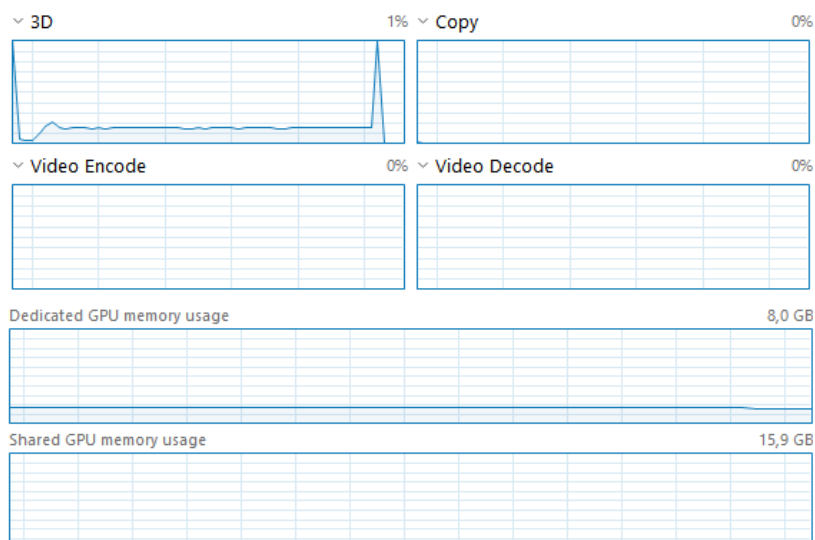
Wszystkie te wyniki potwierdzają wysoką jakość działania programu, jego optymalizację i efektywność.



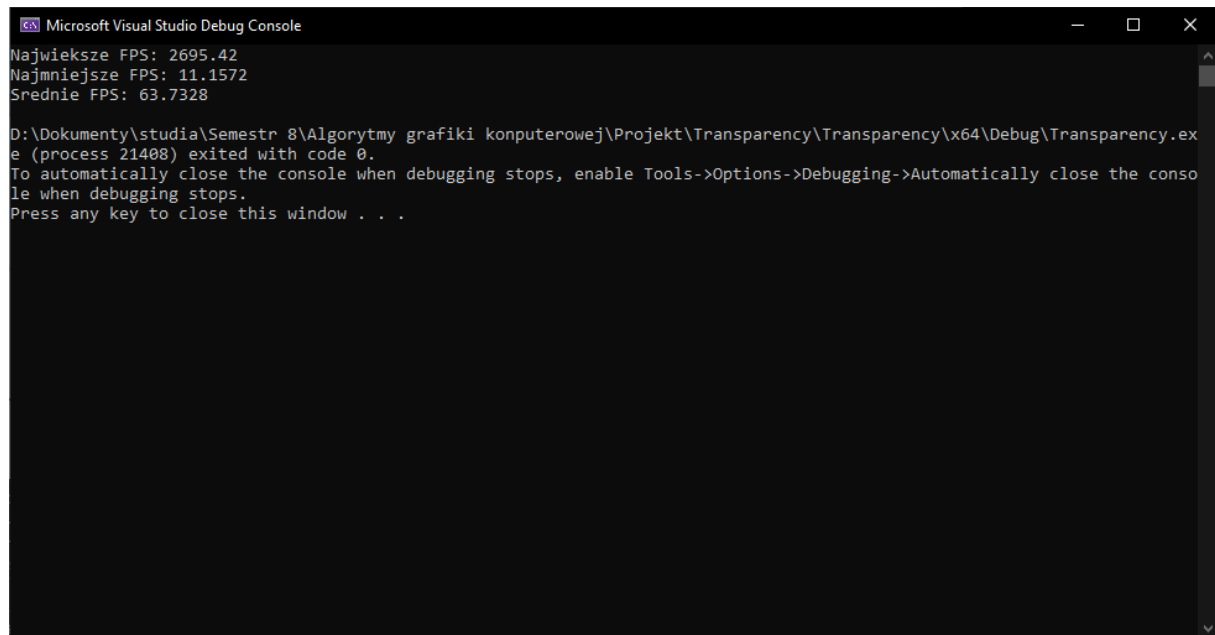
Rysunek 1 Zużycie RAM i CPU

GPU

NVIDIA GeForce RTX 3070



Rysunek 2 Zużycie GPU



The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard Windows window controls (minimize, maximize, close). The console output displays the following text:

```
Najwieksze FPS: 2695.42  
Najmniejsze FPS: 11.1572  
Srednie FPS: 63.7328  
  
D:\Dokumenty\studia\Semestr 8\Algorytmy grafiki komputerowej\Projekt\Transparency\Transparency\x64\Debug\Transparency.exe (process 21408) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

Rysunek 3 Ilość uzyskanych FPS