

# Politechnika Świętokrzyska w Kielcach

Wydział Elektrotechniki, Automatyki i Informatyki



Politechnika Świętokrzyska  
Kielce University of Technology

## Podstawy Grafiki Komputerowej 2

Projekt

### Gra Chińczyk na własnym Silniku 3D

#### Skład zespołu:

Oleksandr Popina

Roman Vyhnevskyi

Anton Yatsenko

Kierunek/specjalność: Informatyka

Studia: stacjonarne

Numer grupy: 2ID15A

# **Spis Treści**

1. Wprowadzenie	3
1.1. Krótki opis aplikacji:	3
1.2. Wykorzystane technologie oraz narzędzia:	3
2. Implementacja	3
2.1. Opis głównych funkcjonalności aplikacji:	3
2.2. Prezentacja zrzutów ekranu (screeny) prezentujących działanie aplikacji	5
2.3. Wybrane fragmenty kodu z kluczowymi funkcjonalnościami	11
3. Instrukcja dla dewelopera	18
3.1. Instalacja narzędzi, bibliotek:	18
3.2. Załadowanie projektu w IDE/silniku, komplikacja:	19
4. Instrukcja obsługi	19
4.1. Interfejs użytkownika	19
4.2. Schemat sterowania	20
5. Literatura, Źródła, wykorzystane zasoby zewnętrzne	20
6. Wnioski	20

## 1. Wprowadzenie

### 1.1. Krótki opis aplikacji:

**Chińczyk 3D** to komputerowa wersja klasycznej gry planszowej *Chińczyk (Ludo)*, opracowana w języku **C++** przy wykorzystaniu **autorskiego silnika graficznego 3D**. Aplikacja umożliwia rozgrywkę dla **2 do 4 graczy**, oferując w pełni odwzorowaną logikę gry, w tym:

- animowaną kostkę
- ruch pionków (wychodzenie z domku, poruszanie się po planszy, zbijanie przeciwników)
- obsługę tur oraz warunek zwycięstwa
- dynamiczny ekran zwycięzcy

Dzięki zastosowaniu elementów interfejsu graficznego, użytkownik może wprowadzać nicki graczy oraz w intuicyjny sposób obsługiwać grę. Całość została osadzona w środowisku trójwymiarowym, z własnym systemem rysowania i zarządzania obiektami, światłem oraz cieniowaniem.

### 1.2. Wykorzystane technologie oraz narzędzia:

**Język programowania:** C++

**Silnik graficzny:** autorski silnik 3D oparty na **OpenGL**

**Biblioteki zewnętrzne:**

- **SFML** – obsługa okien(menu, winner), tekstu, ładowania bitmap.
- **GLUT** – zarządzanie kontekstem OpenGL i renderowaniem.
- **GLEW** – inicjalizacja i zarządzanie rozszerzeniami OpenGL.
- **Assimp** – importowanie i ładowanie modeli 3D z pliku .obj.

**Modelowanie 3D:** Plik .obj reprezentujący pionki.

**IDE:** Microsoft Visual Studio 2022

**System operacyjny:** Windows 11

**Kontrola wersji:** Git / GitHub

**Repozytorium projektu:** [github.com/olekpopina/Silnik\\_3D](https://github.com/olekpopina/Silnik_3D)

## 2. Implementacja

### 2.1. Opis głównych funkcjonalności aplikacji:

Aplikacja „**Chińczyk 3D**” oferuje kompletne środowisko do rozgrywki w klasyczną grę planszową, przedstawione w nowoczesnej formie 3D. Kluczowe funkcjonalności systemu obejmują:

- **Rozgrywka wieloosobowa**  
Obsługa od 2 do 4 graczy z możliwością wprowadzenia własnych nicków przed rozpoczęciem rozgrywki. Każdy z graczy kontroluje indywidualny zestaw pionków.
- **Interaktywna kostka**  
Kostka znajduje się w centralnym punkcie planszy i wyposażona jest w realistyczną animację rzutu. Gracze aktywują rzut kliknięciem myszy, zgodnie z aktualną turą.
- **Zarządzanie pionkami**  
Każdy gracz dysponuje czterema niezależnymi pionkami. System umożliwia:
  - wprowadzenie pionka z domku po wyrzuceniu 6,
  - ruch po planszy zgodnie z liczbą oczek,
  - zajmowanie tego samego pola co przeciwnik oraz
  - zbijanie przeciwnych pionków (z powrotem do domku).
- **System tur i wyjątków**  
Gra działa w trybie naprzemiennym: każda tura przypisana jest konkretnemu graczowi. Dodatkowo implementowane są wyjątki zgodne z zasadami:
  - ponowny rzut po wyrzuceniu 6,
  - pominięcie tury po trzech kolejnych 6.
- **Pełna logika gry Ludo**  
Zasady gry są zaimplementowane zgodnie z klasycznym Chińczykiem:
  - wyjście pionków z domku,
  - ponowny rzut po wyrzuceniu 6,
  - zbijanie i cofanie,
  - prowadzenie pionków do mety oraz zakończenie gry po umieszczeniu wszystkich 4 pionków w bazie końcowej.
- **Ekran zwycięstwa**  
Po spełnieniu warunku wygranej (wszystkie pionki danego gracza w strefie końcowej), wyświetlany jest ekran zwycięzcy z odpowiednim komunikatem i tłem graficznym.
- **Renderowanie 3D**  
Plansza, pionki oraz kostka prezentowane są w środowisku trójwymiarowym. Modele 3D (format [.obj](#)) oraz kamery umożliwiają dynamiczne przedstawienie sceny. Ruch, oświetlenie i tekstury renderowane są z użyciem OpenGL.
- **Interfejs użytkownika (GUI)**  
Menu startowe umożliwia wprowadzanie nicków i konfigurację gry. Interfejs zawiera przyciski:
  - „Start gry”
  - „Zaloguj się gracz 1”
  - „Zaloguj się gracz 2”
  - „Zaloguj się gracz 3”
  - „Zaloguj się gracz 4”

oraz komunikaty związane z przebiegiem gry.

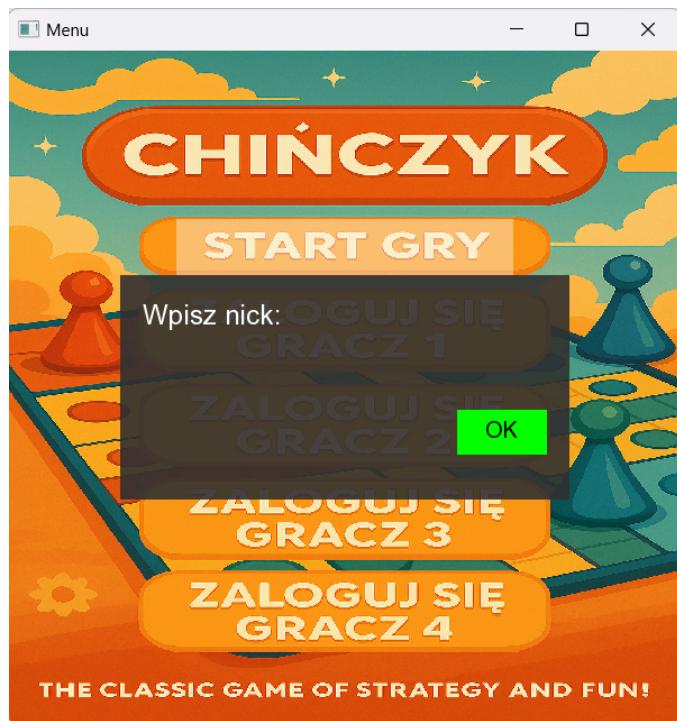
2.2. Prezentacja zrzutów ekranu (screeny) prezentujących działanie aplikacji



Rys. 1. Menu startowe

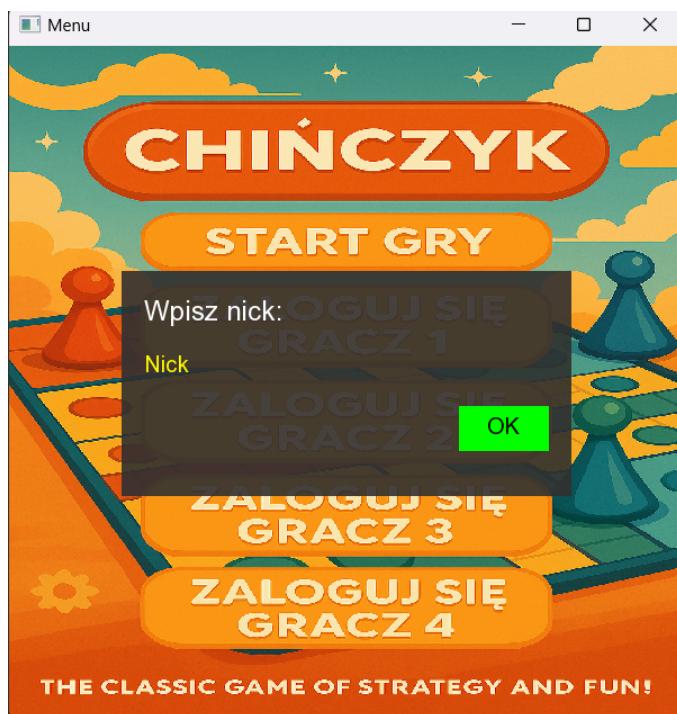


Rys. 2. Jeżeli kurSOR myszy znajduje się na przycisku, to on się świeci przezroczysto



Rys. 3. Żeby się zalogować należy kliknąć na jakiś przycisk Zaloguj się

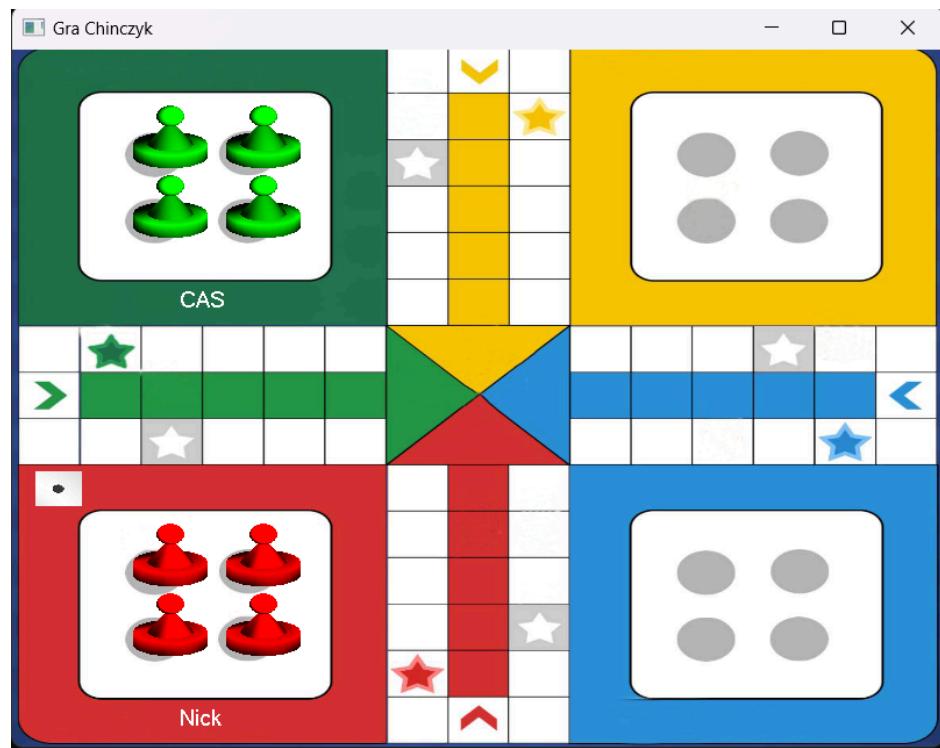
- Zaloguj się gracz 1 - czerwony pionek
- Zaloguj się gracz 2 - niebieski pionek
- Zaloguj się gracz 3 - zielony pionek
- Zaloguj się gracz 4 - żółty pionek



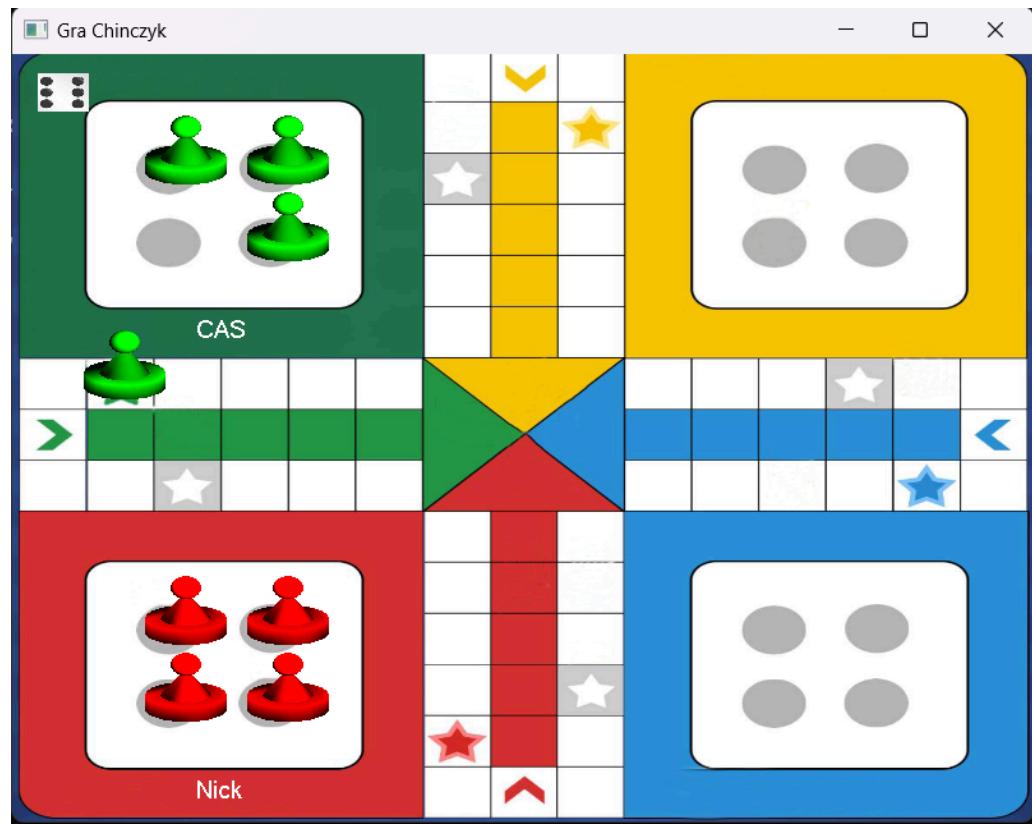
Rys. 4. Po wpisaniu Nick'u należy kliknąć zielony przycisk „OK” i gracz jest już zalogowany



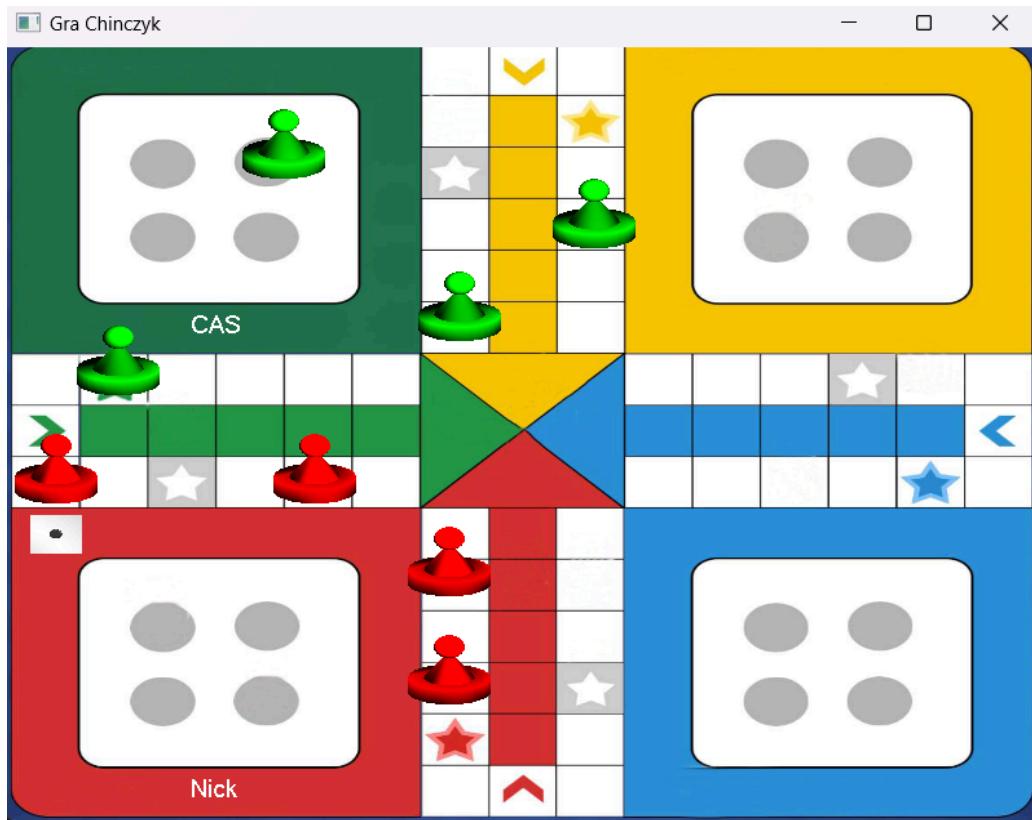
Rys. 5. Jeżeli nie podano żadnego Nick'u lub podano mniej jak dwa to wyświetla się na górze taki komunikat



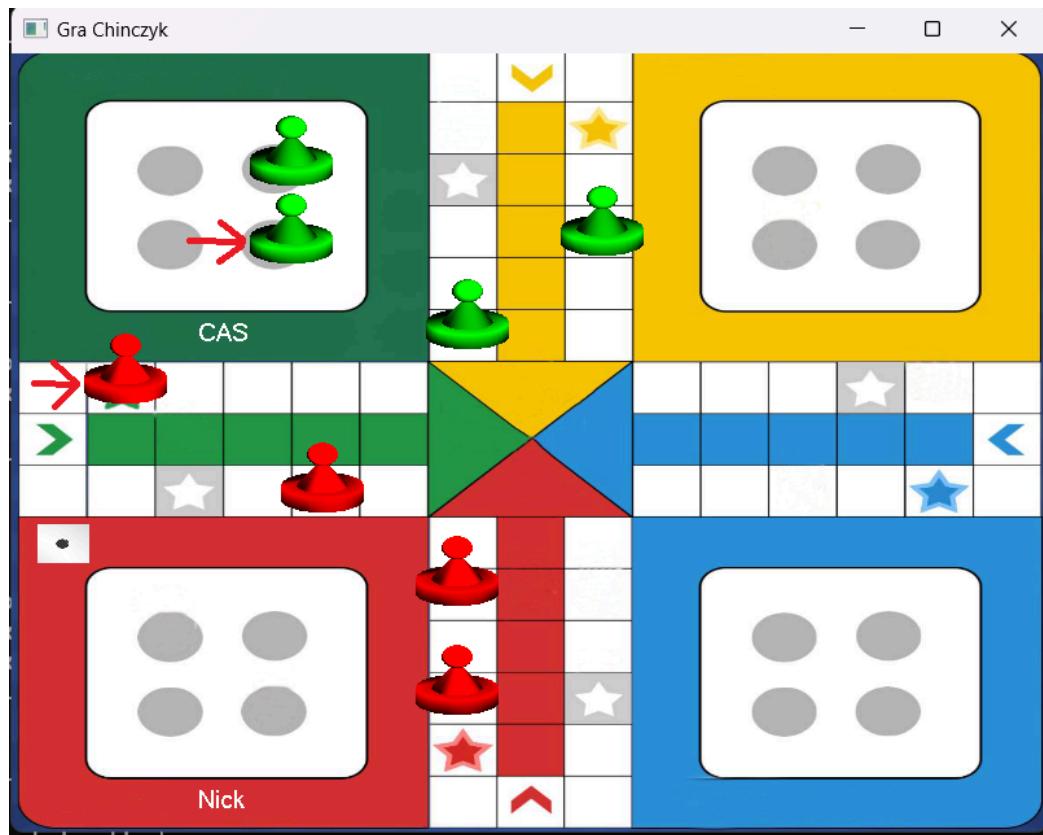
Rys. 6. Jeżeli podano tylko dwa Nick'i, to będzie gra odbywać się pomiędzy dwoma graczami



Rys. 7. Jeżeli wylosowała się 6, to gracz może wyciągnąć 1 pionek z domku



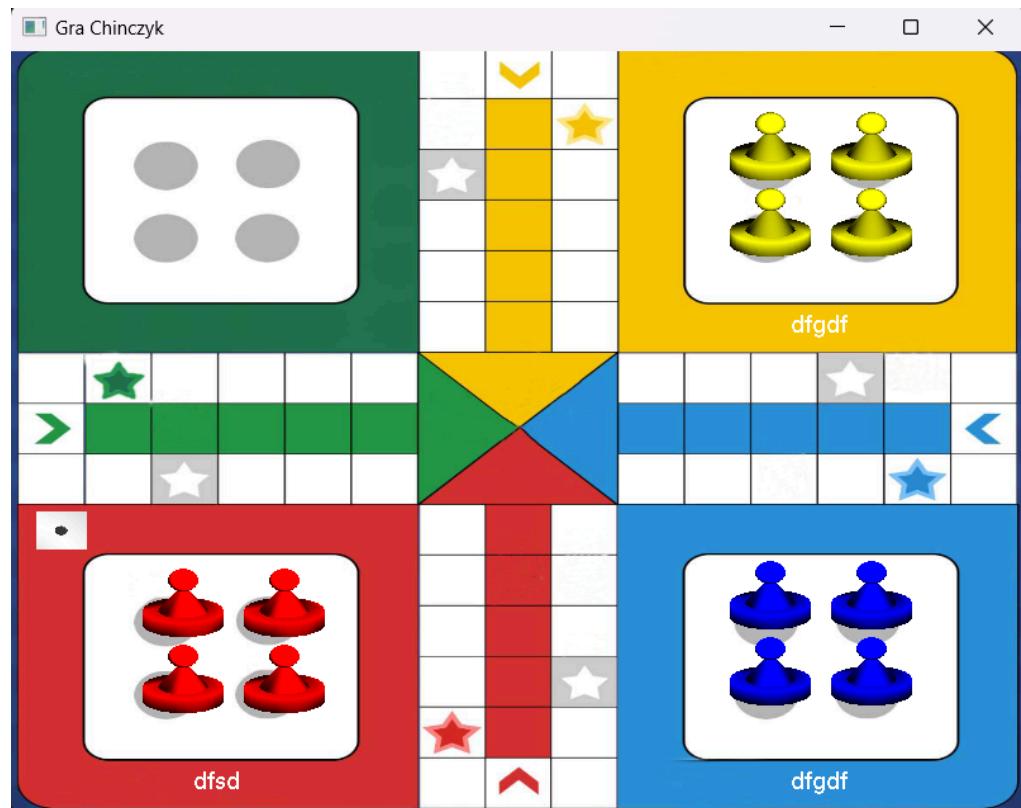
Rys. 8. Możliwość wyciągnięcia wszystkich pionków domku(na przykładzie czerwonych)



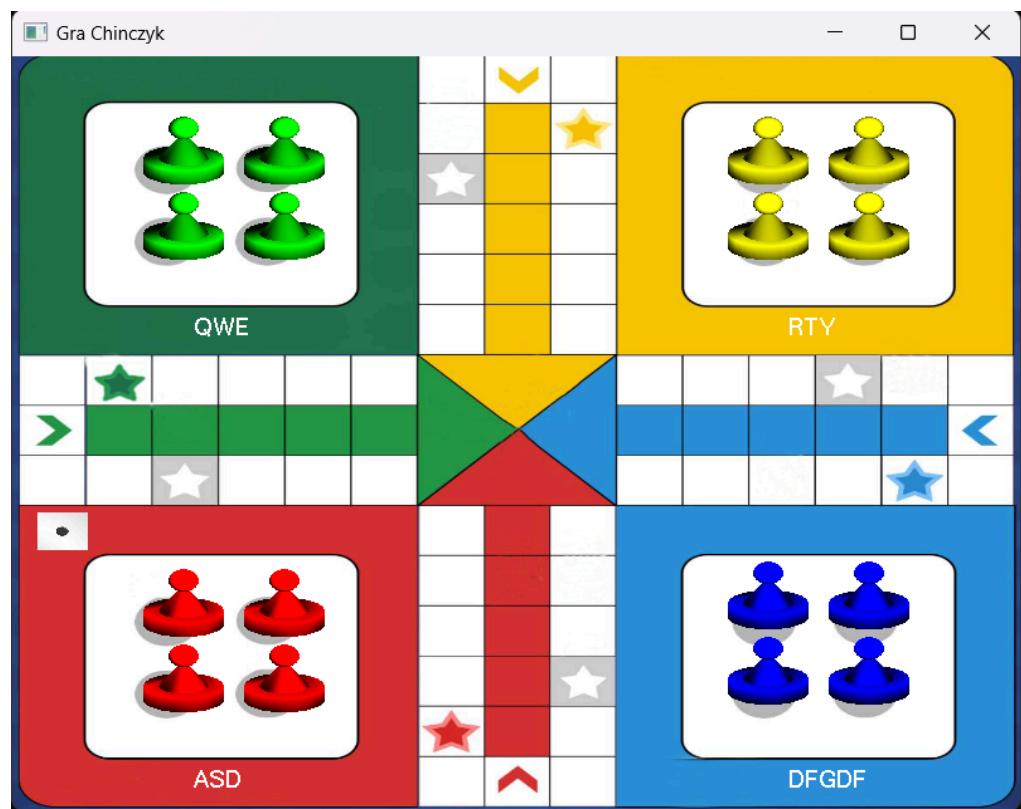
Rys. 9. Jeżeli któryś z pionku dotrze do miejsca gdzie już stoi pionek to ten pionek który już był w tym miejscu wraca do domku(czyli zostaje zbity)



Rys. 10. Jeżeli z któryś z graczy dotrze 4 pionkami do środka to wyświetla się okno Winner z Nick'iem gracza



Rys. 11. Wygląd planszy kiedy zalogowano 3 graczy



Rys. 12. Zalogowano 4 graczy, więc pojawiły się 4 kolory pionków na planszę

### 2.3. Wybrane fragmenty kodu z kluczowymi funkcjonalnościami

```
bool BitmapHandler::loadTextures(const std::vector<std::string>& texturePaths) {
    std::vector<GLuint*> textures = {
        &textureBackground, &texture1, &texture2, &texture3, &texture4, &texture5, &texture6,
        &texture_pionek, &texture_pionek2
    };

    if (texturePaths.size() != textures.size()) {
        std::cerr << "Niepoprawna liczba sciezek do tekstur!" << std::endl;
        return false;
    }

    bool allLoaded = true;
    for (size_t i = 0; i < texturePaths.size(); ++i) {
        *textures[i] = loadSingleTexture(texturePaths[i]);
        if (!glIsTexture(*textures[i])) { // Sprawdź, czy tekstura została poprawnie załadowana
            std::cerr << "Nie udało się załadować tekstury: " << texturePaths[i] << std::endl;
            allLoaded = false;
        }
    }
    return allLoaded;
}
```

Listing 1. Funkcja loadTextures odpowiadająca za ładowanie tekstur kostki i tła Chińczyk.

Tworzy lokalną listę wskaźników na zmienne GLuint, które reprezentują różne tekstury w grze. Sprawdza, czy liczba dostarczonych ścieżek zgadza się z liczbą tekstur – jeśli nie, wypisuje błąd i przerwia. W pętli ładuje każdą teksturę za pomocą funkcji pomocniczej loadSingleTexture. Sprawdza poprawność załadowania każdej tekstury przy użyciu glIsTexture. Jeśli choć jedna tekstura się nie załaduje, funkcja zwraca false.

```
void BitmapHandler::drawBackground() {
    if (!glIsTexture(textureBackground)) return; // Sprawdź, czy tekstura jest załadowana

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0.0, 1.0, 0.0, 1.0); // Ustaw układ współrzędnych dla widoku 2D

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glDisable(GL_DEPTH_TEST); // Wyłącz test głębokości
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureBackground);

    glBegin(GL_QUADS); // Rysowanie prostokąta z teksturą
    glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 1.0f);
    glEnd();
}
```

```

glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 1.0f);
glEnd();

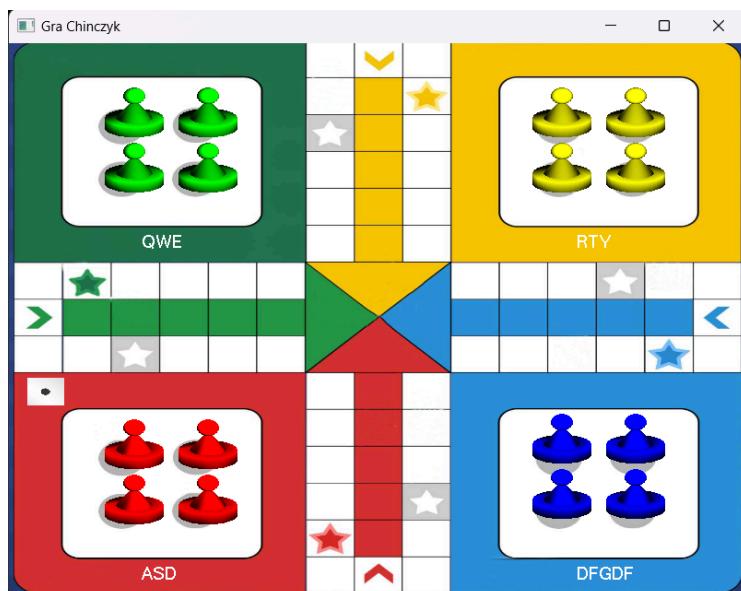
glBindTexture(GL_TEXTURE_2D, 0); // Odłącz teksturę
glEnable(GL_DEPTH_TEST); // Włącz test głębokości

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
}

```

Listing 2. Funkcja drawBackground odpowiadająca za rysowanie tła Chińczyk.

Sprawdza czy tekstura textureBackground została poprawnie załadowana (glIsTexture()). Ustawia rzutowanie ortograficzne 2D (gluOrtho2D), aby móc narysować tekstury w przestrzeni ekranu. Wyłącza test głębokości, by tło nie kolidowało z obiektami 3D. Rysuje prostokąt (GL\_QUADS) od (0,0) do (1,1), pokryty tekstonią tła. Resetuje stan OpenGL – odłącza teksturę, włącza z powrotem test głębokości, przywraca poprzednie macierze rzutowania i modelu.



Rys. 13. Narysowane tło Chińczyk oraz kostka za pomocą tych funkcji

```

bool Pawn3D::loadModel() {
    std::string path = "images/pionek.obj"; // <<--- Ścieżka do modelu 3D pionka

    scene = importer.ReadFile(path,
        aiProcess_Triangulate | aiProcess_GenSmoothNormals);

    if (!scene) {
        std::cerr << "Błąd ładowania modelu: " << importer.GetErrorString() << std::endl;
        return false;
    }
    return true;
}

```

Listing 3. Funkcja loadModel klasy Pawn3D odpowiadająca za ładowanie pliku .obj.

Ustawia ścieżkę do pliku .obj z modelem pionka. Używa Assimp::Importer::ReadFile(...) do wczytania modelu z dwoma podstawowymi opcjami przetwarzania siatki. Sprawdza, czy model został poprawnie załadowany – jeśli nie, wypisuje błąd i zwraca false. W przypadku sukcesu przypisuje wynik do wskaźnika scene i zwraca true.

```
void Pawn3D::draw() const {
    if (!scene) return;

    for (unsigned int i = 0; i < scene->mNumMeshes; ++i) {
        const aiMesh* mesh = scene->mMeshes[i];
        const aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];

        aiColor3D diffuse(0.f, 0.f, 0.f);
        if (material->Get(AI_MATKEY_COLOR_DIFFUSE, diffuse) == AI_SUCCESS) {
            float brightnessScale = 3.0f;
            glColor3f(
                std::min(1.0f, diffuse.r * brightnessScale),
                std::min(1.0f, diffuse.g * brightnessScale),
                std::min(1.0f, diffuse.b * brightnessScale));
        }
        else {
            std::cout << "[WARN] Brak koloru w materiale – ustawiam szary domyślny." << std::endl;
            glColor3f(0.8f, 0.8f, 0.8f);
        }
        glBegin(GL_TRIANGLES);
        for (unsigned int j = 0; j < mesh->mNumFaces; ++j) {
            const aiFace& face = mesh->mFaces[j];
            for (unsigned int k = 0; k < face.mNumIndices; ++k) {
                unsigned int index = face.mIndices[k];
                aiVector3D vertex = mesh->mVertices[index];

                if (mesh->HasNormals()) {
                    aiVector3D normal = mesh->mNormals[index];
                    glNormal3f(normal.x, normal.y, normal.z);
                }

                glVertex3f(vertex.x, vertex.y, vertex.z);
            }
        }
        glEnd();
    }
}
```

Listing 4. Funkcja draw klasy Pawn3D odpowiadająca za rysowanie modelu 3D pionku.

Sprawdza, czy model (scene) został załadowany – jeśli nie, przerwuje rysowanie.

Dla każdej siatki (mesh): Pobiera materiał i próbuje odczytać jego kolor diffuse. Ustawia kolor w OpenGL, wzmacniając go mnożnikiem (brightnessScale), aby model był jaśniejszy. Jeśli brak koloru w materiale, używa domyślnego koloru szarego.

Dla każdej twarzy (face) rysuje trójkąty (GL\_TRIANGLES): Pobiera współrzędne wierzchołków (vertex). Jeśli dostępne – ustawia normalne wektory (normal) dla poprawnego cieniowania.



Rys. 14. Załadowany 3D model pionku z pliku .obj (funkcja draw rysuje pionek bez oświetlenia, czyli jego kolor będzie szary)

```
std::vector<std::pair<float, float>> Paths::getRedHouse() {
    return {
        {0.12f, 0.12f}, {0.12f, 0.22f}, {0.22f, 0.12f}, {0.22f, 0.22f}
    };
}

std::vector<std::pair<float, float>> Paths::getRedPath() {
    return {
        {0.38f, 0.06f}, {0.38f, 0.14f}, {0.38f, 0.21f}, {0.38f, 0.28f}, {0.38f, 0.35f}, {0.31f, 0.4f}, {0.25f, 0.4f},
        {0.18f, 0.4f}, {0.11f, 0.4f}, {0.04f, 0.4f}, {0.00f, 0.4f}, {0.00f, 0.47f}, {0.00f, 0.54f},
        {0.06f, 0.54f}, {0.13f, 0.54f}, {0.20f, 0.54f}, {0.26f, 0.54f}, {0.33f, 0.54f}, {0.39f, 0.61f},
        {0.39f, 0.68f}, {0.39f, 0.75f}, {0.39f, 0.82f}, {0.39f, 0.89f}, {0.39f, 0.94f}, {0.46f, 0.94f},
        {0.52f, 0.94f}, {0.52f, 0.87f}, {0.52f, 0.80f}, {0.52f, 0.73f}, {0.52f, 0.66f}, {0.52f, 0.59f},
        {0.59f, 0.53f}, {0.66f, 0.53f}, {0.72f, 0.53f}, {0.79f, 0.53f}, {0.86f, 0.53f}, {0.91f, 0.53f},
        {0.91f, 0.46f}, {0.91f, 0.4f}, {0.85f, 0.4f}, {0.78f, 0.4f}, {0.71f, 0.4f}, {0.64f, 0.4f},
        {0.58f, 0.4f}, {0.52f, 0.33f}, {0.52f, 0.26f}, {0.52f, 0.19f}, {0.52f, 0.12f}, {0.52f, 0.05f},
        {0.52f, 0.00f}, {0.45f, 0.00f}, {0.45f, 0.07f}, {0.45f, 0.14f}, {0.45f, 0.21f}, {0.45f, 0.28f},
        {0.45f, 0.35f}, {0.45f, 0.42f}
    };
}
```

Listing 5. Funkcja getRedHouse zwraca współrzędne czterech pól w tzw. „domku” czerwonego gracza, czyli miejsca, z którego pionki startują grę.

Zwracane wartości to pary liczb zmiennoprzecinkowych (x, y), które określają położenie każdego z czterech pionów przed ich wyjściem na główną trasę. Funkcja getRedPath zwraca wektor współrzędnych określających trasę ruchu pionka czerwonego gracza po planszy. Trasa rozpoczyna się na jego polu startowym i prowadzi przez całą planszę – wokół pola gry, zgodnie z ruchem wskazówek zegara – aż do końcowego odcinka znajdującego się w centrum planszy (tzw. „meta”).

Współrzędne w tym wektorze odpowiadają kolejnym polom, które pionek ma pokonywać w trakcie gry. Są one zgodne z układem współrzędnych ortograficznych używanym w renderowaniu planszy (od 0.0 do 1.0). Tak samo działa to dla żółtego, zielonego i niebieskiego pionków

```
void PrimitiveDrawer::drawCubeWithTexture(float scale, float offsetX, float offsetY, BitmapHandler& bitmapHandler) {
    glEnable(GL_TEXTURE_2D);
```

```

switch (textureSet) {
    case 1: // Zestaw tekstur 1
        textures[0] = bitmapHandler.texture1; // Przednia
        textures[1] = bitmapHandler.texture2; // Tylna
        textures[2] = bitmapHandler.texture3; // Lewa
        textures[3] = bitmapHandler.texture4; // Prawa
        textures[4] = bitmapHandler.texture5; // Góra
        textures[5] = bitmapHandler.texture6; // Dolna
        break;
    case 2: // Zestaw tekstur 2
        textures[0] = bitmapHandler.texture2; // Przednia
        textures[1] = bitmapHandler.texture3; // Tylna
        textures[2] = bitmapHandler.texture4; // Lewa
        textures[3] = bitmapHandler.texture5; // Prawa
        textures[4] = bitmapHandler.texture6; // Góra
        textures[5] = bitmapHandler.texture1; // Dolna
        break;
    case 3: // Zestaw tekstur 2
        textures[0] = bitmapHandler.texture3; // Przednia
        textures[1] = bitmapHandler.texture2; // Tylna
        textures[2] = bitmapHandler.texture4; // Lewa
        textures[3] = bitmapHandler.texture5; // Prawa
        textures[4] = bitmapHandler.texture6; // Góra
        textures[5] = bitmapHandler.texture1; // Dolna
        break;
    case 4: // Zestaw tekstur 2
        textures[0] = bitmapHandler.texture4; // Przednia
        textures[1] = bitmapHandler.texture3; // Tylna
        textures[2] = bitmapHandler.texture2; // Lewa
        textures[3] = bitmapHandler.texture5; // Prawa
        textures[4] = bitmapHandler.texture6; // Góra
        textures[5] = bitmapHandler.texture1; // Dolna
        break;
    case 5: // Zestaw tekstur 2
        textures[0] = bitmapHandler.texture5; // Przednia
        textures[1] = bitmapHandler.texture2; // Tylna
        textures[2] = bitmapHandler.texture4; // Lewa
        textures[3] = bitmapHandler.texture3; // Prawa
        textures[4] = bitmapHandler.texture6; // Góra
        textures[5] = bitmapHandler.texture1; // Dolna
        break;
    case 6: // Zestaw tekstur 2
        textures[0] = bitmapHandler.texture6; // Przednia
        textures[1] = bitmapHandler.texture2; // Tylna
        textures[2] = bitmapHandler.texture4; // Lewa
        textures[3] = bitmapHandler.texture5; // Prawa
        textures[4] = bitmapHandler.texture3; // Góra
        textures[5] = bitmapHandler.texture1; // Dolna
        break;
    default: // Domyślny zestaw tekstur
        textures[0] = bitmapHandler.texture1;
        textures[1] = bitmapHandler.texture2;
        textures[2] = bitmapHandler.texture3;
}

```

```

textures[3] = bitmapHandler.texture4;
textures[4] = bitmapHandler.texture5;
textures[5] = bitmapHandler.texture6;
break;
}

glPushMatrix();
glTranslatef(offsetX, offsetY, 0.0f);
glScalef(scale, scale, scale);

// Przednia ściana
glBindTexture(GL_TEXTURE_2D, textures[0]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glEnd();

// Tylna ściana
glBindTexture(GL_TEXTURE_2D, textures[1]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();

// Lewa ściana
glBindTexture(GL_TEXTURE_2D, textures[2]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();

// Prawa ściana
glBindTexture(GL_TEXTURE_2D, textures[3]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f);
glEnd();

// Góra ściana
glBindTexture(GL_TEXTURE_2D, textures[4]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f);
glEnd();

```

```

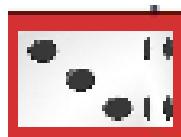
// Dolna ściana
glBindTexture(GL_TEXTURE_2D, textures[5]);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, -1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, -1.0f, -1.0f);
glEnd();

glPopMatrix();
glBindTexture(GL_TEXTURE_2D, 0);
glDisable(GL_TEXTURE_2D);
}

```

Listing 6. Funkcja drawCubeWithTexture odpowiadająca za rysowanie sześciangu (kostki) 3D z teksturami na wszystkich sześciu ścianach, przy użyciu OpenGL.

Każda ściana kostki otrzymuje osobną teksturę, którą można wybrać dynamicznie dzięki zmiennej textureSet. Ona robi: Włącza tryb teksturowania. Wybiera zestaw tekstur. Ustawia transformacje. Rysuje każdą ścianę kostki. Przywraca stan OpenGL.



Rys. 15. Narysowana kostka do gry za pomocą tej funkcji

```

void Engine::updatePawnPosition(const std::string& id) {
// funkcja jest zbyt dłużna, całość można zobaczyć tutaj:
https://github.com/olekpopina/Silnik\_3D/blob/main/Silnik\_3D/Engine.cpp
}

```

Listing 6. Funkcja updatePawnPosition.

Funkcja odpowiadająca za logikę chodzenia pionkami po polu, za zbijanie swojego przeciwnika oraz wyświetlenia okna Winner, jeżeli któryś z graczy doszedł 4 pionkami do środka pola, a także za powrócenia zditego pionka do domku

```

void Engine::onMouse(int button, int state, int x, int y) {
// funkcja jest zbyt dłużna, całość można zobaczyć tutaj:
https://github.com/olekpopina/Silnik\_3D/blob/main/Silnik\_3D/Engine.cpp
}

```

Listing 6. Funkcja odpowiada za obsługę kliknięć myszy w grze Chińczyk (Ludo).

Reaguje na kliknięcie lewym przyciskiem myszy i realizuje trzy główne zadania:

**Kliknięcie w pionek w domku** – jeśli gracz wyrzucił 6 i ma pionki w domku, może kliknąć wybrany pionek, aby wyciągnąć go na planszę.

**Kliknięcie w pionek na planszy** – jeśli gracz ma pionki na planszy i zakończył rzut kostką, może kliknąć w jednego z nich, aby poruszyć go o wylosowaną liczbę pól.

**Kliknięcie w kostkę** – jeżeli kostka nie jest w trakcie animacji i żaden pionek nie wykonuje ruchu, kliknięcie w kostkę rozpoczyna animację obrotu i losuje wynik rzutu.

Funkcja obsługuje czterech graczy (czerwony, niebieski, żółty, zielony) oraz cztery pionki dla każdego z nich. Uwzględnia także specjalne zasady gry, takie jak możliwość ponownego rzutu po wyrzuceniu 6 oraz zakończenie tury po wyrzuceniu trzech szóstek z rzędu.

```
void Engine::render() {  
    // funkcja jest zbyt d³uga, ca³o¶ mo¿na zobaczyæ tutaj:  
    // https://github.com/olekpopina/Silnik\_3D/blob/main/Silnik\_3D/Engine.cpp  
}
```

Listing 7. Funkcja odpowiada za wyświetlanie jednej klatki gry oraz aktualizację jej stanu wizualnego.

Jest ona wywoływana cyklicznie i realizuje następujące zadania: Kontrola liczby klatek na sekundę (frame rate). Czyszczenie buforów i inicjalizacja sceny. Animacja i zakoñczenie obrotu kostki. Ruch pionków na planszy. Rysowanie pionków 3D. Rysowanie kostki w widoku 2D. Wyświetlanie nazw graczy. Wymiana buforów.

### 3. Instrukcja dla dewelopera

#### 3.1. Instalacja narzędzi, bibliotek:

Aby uruchomić projekt „Chińczyk 3D” oraz własny silnik graficzny, należy skonfigurować środowisko programistyczne w **Visual Studio 2022 Community Edition** i zainstalować odpowiednie biblioteki za pomocą **NuGet**.

**Kroki instalacji:**

- **Otwórz projekt w Visual Studio 2022 Community.**

Upewnij się, że konfiguracja projektu ustawiona jest na x64 oraz używa kompilatora MSVC.

- **Zainstaluj wymagane biblioteki przez NuGet:**

Kliknij prawym przyciskiem myszy na projekt w **Eksploratorze rozwiązań** i wybierz „**Zarządzaj pakietami NuGet**”.

Przejdź do zakładki „**Przeglądaj**” i wyszukaj oraz zainstaluj następujące pakiety:

- **assimp** – biblioteka do wczytywania modeli 3D
- **glew** – zarządzanie rozszerzeniami OpenGL
- **sFML** – obsługa okien, zdarzeń, tekstu
- **freeglut** – niezbędna do poprawnego działania silnika graficznego i obsługi kontekstu OpenGL

- **Sprawdź poprawno¶ konfiguracji:**

Visual Studio automatycznie doda ścieżki do nagłówków i bibliotek z NuGet, ale w razie

potrzeby możesz je ręcznie sprawdzić w:

**Właściwości projektu → VC++ Directories → Include/Library Directories**

- **Zbuduj projekt i uruchom aplikację.**

Po pomyślnej instalacji bibliotek projekt powinien kompilować się bez błędów.

3.2. Załadowanie projektu w IDE/silniku, komplikacja:

Po zainstalowaniu wszystkich wymaganych bibliotek, projekt „Chińczyk 3D” można z łatwością załadować i uruchomić w środowisku **Visual Studio 2022 Community**.

#### **Instrukcja:**

- **Otwórz plik .sln projektu** w Visual Studio – jest to gotowe rozwiązanie zawierające cały kod źródłowy gry oraz silnika 3D.
- **Zbuduj projekt:**
  - W menu wybierz „**Kompiluj → Kompiluj rozwiązanie**”
- **Uruchom grę** za pomocą przycisku **Start (F5)** lub skrótu **Ctrl + F5** (bez debugowania).

Po poprawnym zbudowaniu aplikacja uruchomi się i otworzy menu startowe gry Chińczyk.

## **4. Instrukcja obsługi**

### **4.1. Interfejs użytkownika**

Gra „Chińczyk 3D” to cyfrowa wersja klasycznej gry planszowej dla 2–4 graczy. Celem gry jest przeprowadzenie wszystkich swoich pionków z domku do pola końcowego

#### **Aby rozpocząć grę:**

1. Uruchom Visual Studio Community i skompiluj kod.
2. Wprowadź nicki graczy w menu startowym.
3. Kliknij „Start gry”, aby przejść do planszy.
4. Kliknij kostkę, aby wykonać rzut.
5. Kliknij wybrany pionek (jeżeli wylosowano 6), aby go przesunąć o wskazaną liczbę oczek.

#### **Interfejs użytkownika składa się z trzech głównych ekranów:**

- **Menu startowe:**

- Przyciski „Zaloguj się” umożliwiają wpisanie nicków graczy.
- Przycisk „Start gry” – aktywny po wpisaniu co najmniej dwóch nicków.
- W razie braku nicków – komunikat: „Podaj nick'i przynajmniej 2 graczy!”

- **Plansza gry:**

- Widok planszy z pionkami 3D.
- Interaktywna kostka do rzutu.

- Nicki graczy widoczne pod ich domkami.
- Komunikaty tekstowe w konsoli (np. „Ruch pionku: Czerwony”).
- **Okno zwycięzcy:**
  - Po zakończeniu gry – ekran informujący o wygranej konkretnego gracza.

#### 4.2. Schemat sterowania

##### Mysz:

- Lewy przycisk – kliknięcie na kostkę, aby wykonać rzut.
- Lewy przycisk – kliknięcie na pionek, aby go przesunąć.

##### Klawiatura:

- Wprowadzanie liczby oczek kostki (funkcja pomocnicza – do testów/debugowania).

## 5. Literatura, źródła, wykorzystane zasoby zewnętrzne

- Dokumentacja OpenGL: <https://www.khronos.org/registry/OpenGL-Refpages/>
- Dokumentacja SFML: <https://www.sfml-dev.org/documentation/>
- Modele 3D pionków i stworzone samodzielnie w Blenderze.
- Font: Arial z Windows
- Tekstura planszy wzięta z gry „Ludo Club”
- Grafiki: własne + tło *winner.png*
- Tekstury kostki stworzona samodzielnie w Photoshop.
- Repozytorium projektu: [github.com/olekpopina/Silnik\\_3D](https://github.com/olekpopina/Silnik_3D)
- Dokumentacja techniczna wygenerowana za pomocą narzędzia \*\*Doxygen\*\*
  - Plik konfiguracyjny: ‘Doxyfile.txt’

### 5.1 Dokumentacja techniczna

Kod źródłowy projektu został udokumentowany za pomocą systemu Doxygen.

Wygenerowana dokumentacja znajduje się w katalogu [docs/html](#) i zawiera szczegółowe opisy klas, funkcji oraz zależności między modułami silnika 3D.

## 6. Wnioski

Projekt gry Ludo w 3D pozwolił nam zrealizować pełny cykl tworzenia aplikacji graficznej: od podstawowego silnika renderującego po działającą logikę planszówki. Udało się wdrożyć dynamiczne oświetlenie, teksturowanie, kolizje oraz interfejs użytkownika.

Największym wyzwaniem było połączenie warstwy logiki z grafiką oraz obsługa wielu pionków i graczy w czasie rzeczywistym. Projekt dał nam solidne podstawy do tworzenia bardziej zaawansowanych gier 3D i rozszerzania własnego silnika o nowe funkcje.