Sprawozdanie

Projekt – Szkielet gry w OpenGL

1. Cel projektu

Należy stworzyć szkielet gry, o temacie wcześniej skonsultowanym z prowadzącym. Wykorzystać zdobytą wiedzę z modelowania, interakcji, oświetlenia i teksturowania. W przypadku tego projektu tematem jest tetris, zamodelowany w trójwymiarze.

2. Implementacja

Pierwszą dużą zmianą w porównaniu z programami pisanymi w trakcie zajęć, jest zmiana języka programowania z C++ na C#. Aby łatwiej korzystało się z OpenGl'a, pracę oparto na bibliotece OpenTK.

Kluczowym punktem implementacji jest klasa **Game**, która dziedziczy po klasie GameWindow. Zaimplementowano w niej metody **OnLoad** odpowiednik MyInit()**,** **OnUpdateFrame** oraz **OnRenderFrame** odpowiednik RenderScene(). W metodzie OnUpdateFrame odbywa się cała logika, pozwala to na uczciwą rozgrywkę, ponieważ efektywność w grze nie będzie zależeć od liczby klatek na sekundę, czyli wydajności komputera. Maksymalna częstotliwość odświeżania ekranu to 60 Hz.

Listing 1 Metoda OnLoad, inicjująca potrzebne właściwości (głębię obrazu i światło)

```
protected override void OnLoad(EventArgs e)
{
        base.OnLoad(e);

        GL.ClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        GL.Enable(EnableCap.DepthTest);

        //Light
        GL.Enable(EnableCap.Lighting);

        float[] light_position = new float[] { 0.0f, 0.0f, 1.5f };
        GL.Light(LightName.Light0, LightParameter.Position, light_position);
        GL.Enable(EnableCap.Light0);
}
```

Utworzona została klasa dostarczających statycznych metod, którymi można sprawdzić, czy dany klawisz został wciśnięty, przytrzymany lub puszczony, choć w tym projekcie skorzystano tylko z dwóch pierwszych. Nie licząc metody inicjującej, która tworzy potrzebne struktury i procedury obsługi zdarzeń oraz metody aktualizującej dane o klawiszach.

Listing 2 Metoda odpowiedzialna za całą logikę, wykonująca się przed wyświetleniem obrazu

```
protected override void OnUpdateFrame(FrameEventArgs e)
{
    base.OnUpdateFrame(e);
    frameCounter++;

    if(newBlockNeeded)
    {
        gameBoard.MakeNewBlock();
        newBlockNeeded = false;
        gameBoard.ActualBlock.Move(MoveDirection.Down);
    }

    if (Input.KeyPress(OpenTK.Input.Key.Left))
        gameBoard.ActualBlock.Move(MoveDirection.Left);
    if (Input.KeyPress(OpenTK.Input.Key.Right))
        gameBoard.ActualBlock.Move(MoveDirection.Right);
    if (Input.KeyPress(OpenTK.Input.Key.Up))
        gameBoard.ActualBlock.Rotate();

    if (Input.KeyDown(OpenTK.Input.Key.Down))
        gameBoard.ActualBlock.Move(MoveDirection.Down);

    if (frameCounter >= 60)
    {
        newBlockNeeded = !gameBoard.ActualBlock.Move(MoveDirection.Down);
        frameCounter = level;
    }

    if (newBlockNeeded)
    {
        gameBoard.LockBlock();
        if(gameBoard.DeleteFullLine())
        {
            if (level < 45)
                level++;
            gameBoard.ActualBlock.Move(MoveDirection.Down);
        }

    }

    Input.Update();
}
```

Na listingu 2 zawarta została cała logika gry. Na samym starcie należy sprawdzić czy trzeba utworzyć nowy klocek, który domyślnie pojawi się mniej więcej w połowie szerokości planszy i zacznie spadać. Poza obsługą klawiszy strzałek, które sterują ruchomym elementem, pojawia się warunek, który powoduje automatyczne opuszczenie klocka o jeden wiersz w dół, gdy licznik klatek osiągnie wartość 60. W ciągu sekundy zawsze będzie ich 60, jednak jak można zauważyć, po utworzeniu nowego klocka, licznik klatek staruje od wartości zmiennej level, która początkowo jest równa 0. Gdy metoda ActualBlock.Move() zwróci fałsz, oznacza to, że aktualny element wylądował. Wtedy nastąpi zajęcie przez niego komórek i sprawdzenie czy powstał zapełniony wiersz do wyczyszczenia. Za każdym razem, gdy czyści się wiersze, zwiększa się zmienna level. Im większa wartość level, tym mniej czasu pomiędzy automatycznym spadaniem klocka, a więc większa trudność gry.

Metoda renderująca obraz jest maksymalnie uproszczona, najpierw tworzona jest macierz perspektywy, ustawiane są odpowiednie parametry i ładowana jest ona do pamięci. Następnie przełącza się na macierz widoku modelu, ładuje się macierz opisującą punkt kamery, cel kamery (Matrix4.LookAt - ulepszona wersja metody gluLookAt()). Po tych zabiegach wystarczy określić Viewport i wywołać metodę, która narysuje obiekty gry.

Listing 3 Metoda rysująca grę na ekranie

```
protected override void OnRenderFrame(FrameEventArgs e)
{
    base.OnRenderFrame(e);

    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

    Matrix4 perspective = Matrix4.CreatePerspectiveFieldOfView(0.7f, this.Width
/ this.Height, near, far);

    Matrix4 lookAt = Matrix4.LookAt(new Vector3(0, 2*gameBoardSize,
3*gameBoardSize), modelPosition, new Vector3(0, 1, 0));
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    GL.LoadMatrix(ref perspective);


    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    GL.LoadMatrix(ref lookAt);

    GL.Viewport(0, 0, this.Width, this.Height);

    gameBoard.Render();

    this.SwapBuffers();
}
```

Struktury gry

Całą planszę i wszystkie jej obiekty zamknięto w obiekcie klasy **GameBoard**. Zadaniem tego obiektu jest panowanie nad dwuwymiarową tablicą sześcianów, z których składają się klocki TETRIS'a (dokładnie 4 sześciany na każdy klocek). Gdyby jednak przechowywać jedynie sześciany, problem wykrywania kolizji i stwierdzania, czy spadający element może się w jakiś sposób poruszyć, niepotrzebnie się komplikuje. Zatem jest to dwuwymiarowa tablica obiektów **GameBoardCell**, których zadaniem jest pamiętanie informacji jaki sześcian powinien znaleźć się w danej komórce oraz czy dana komórka jest już zajęta, tymczasowo zajęta (spadający element, jeszcze nie zablokowany), czy pusta.

Obiekt klasy GameBoard pamięta także odwołanie do aktualnego elementu, czyli tego, który się porusza. Gdy element spadnie, ten sam obiekt wypełni na stałe komórki zajmowane przez aktualny element.

Listing 4 Metoda blokująca element na miejscu jego lądowania

```
public void LockBlock()
{
    Dictionary<int, Tuple<int, int>> cells = actualBlock.UsedCells();
    int i = -1, j = -1;
    for (int k = 0; k < 4; k++)
    {
        i = cells[k].Item1;
        j = cells[k].Item2;
        board[i, j].CellStatus = CellStatus.Filled;
    }
}
```

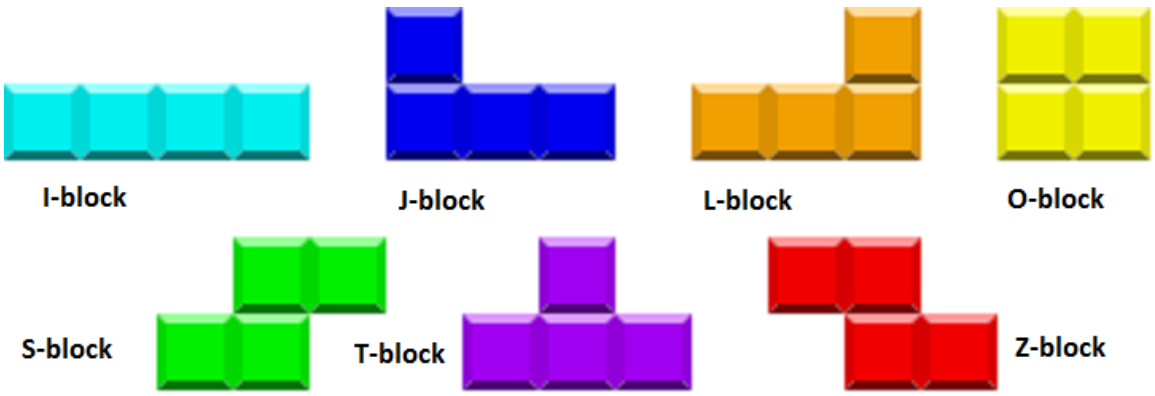
Po każdym „lądowaniu" aktualnego elementu, wywoływana jest metoda, która sprawdzi, czy któryś wiersz planszy nie został całkowicie zapełniony. Jeśli taka sytuacja się wydarzy, zapełniony wiersz zostanie wyczyszczony, a wszystkie wyższe rzędy zostaną przeniesione o jeden w dół. Jest to wykonywane zanim pojawi się nowy element.

Jeśli chodzi o tworzenie nowego elementu, jest to wylosowanie obiektu którejś z klas opisujących spadający element. W celu wyjaśnień należy zaznaczyć, że w implementacji obiekt klasy GameBoard jest trochę bardziej skomplikowany, ponieważ „tablica gry" po której poruszają się elementy jest otoczona po bokach i z dołu pojedynczą warstwą bloków neutralnych. Ich zadaniem jest tylko i wyłącznie zaznaczenie granic ruchów, aby nie mylić gracza. Dodatkowo, wspomniana tablica ma kilka wierszy zapasu „z góry", aby ułatwić implementację obrotów elementów tuż na samym szczycie planszy.

Listing 5 Metoda usuwająca zapełnione wiersze i przesuwająca pozostałe

```
public bool DeleteFullLine()
{
    bool lineDeleted = false;
    for (int j = 1; j < boardHeight + 1; j++)
    {
        bool rowFilled = true;
        for (int i = 1; i < boardAbsoluteWidth - 1; i++)
        {
            if(board[i, j].CellStatus == CellStatus.Free)
            {
                rowFilled = false;
                break;
            }
        }
        if(rowFilled)
        {
            lineDeleted = true;
            int k = j;
            for (k = j; k < boardHeight - 1; k++)
            {
                for (int l = 1; l < boardAbsoluteWidth - 1; l++)
                {
                    board[l, k] = board[l, k + 1];
                }
            }
            for (int l = 1; l < boardAbsoluteWidth - 1; l++)
            {
                board[l, k] = new GameBoardCell();
            }
            j--;
        }
    }
    return lineDeleted;
}
```

Klocki w grze



Rysunek 1 Rodzaje klocków występujące w grze

Pokazane na rysunku 1 elementy zostały zaimplementowane w klasach, które nazwane zostały XBlock, gdzie X to litera odpowiadająca kształtowi. Każdy klocek składa się z dokładnie 4 sześcianów, każdy może się ruszać na boki, obracać i przyspieszać spadanie. Każdy klocek jest także odpowiedzialny za sprawdzenie, czy pod wpływem zmiany ruchu

nie nastąpi kolizja. Jeśli nastąpi albo nie wykona ruchu, albo zmieni położenie (obracanie elementu przy krawędzi planszy). Dodatkowo zgodnie z założeniami obiektu klasy GameBoard, każdy klocek może zwalniać zajmowane przez siebie komórki (metoda **FreeCells()**) oraz zajmować tymczasowo przyjęte w wyniku wykonania ruchu nowe komórki (metoda **TakeCells()**). Pomocniczo implementują także metody Render() i UsedCells(). Pierwsza w celu łatwiejszego rysowania, druga aby zwrócić zajęte komórki do metody **LockBlock()**. Ponieważ każdy klocek zachowuje się dokładnie tak samo poza momentem wykonywania obrotu, zaimplementowano klasę bazową wszystkich klocków **BaseBlock**, dzięki której w klasach odpowiadającym poszczególnym klockom wystarczy zaimplementować konstruktor z przypisaniem początkowych komórek oraz metodę obrotu.

Obrót w grze, wykonywany jest po wciśnięciu klawisza strzałki w górę. Ponieważ jest tylko jeden klawisz, obrót następuje zawsze w pewnej ustalonej kolejności, a klocek przyjmuje zawsze któryś z określonych mu stanów (4, 2 lub 1 dla OBlock). Przykład metody wykonującej obrót klocka typu SBlock został pokazany na listingu 6.

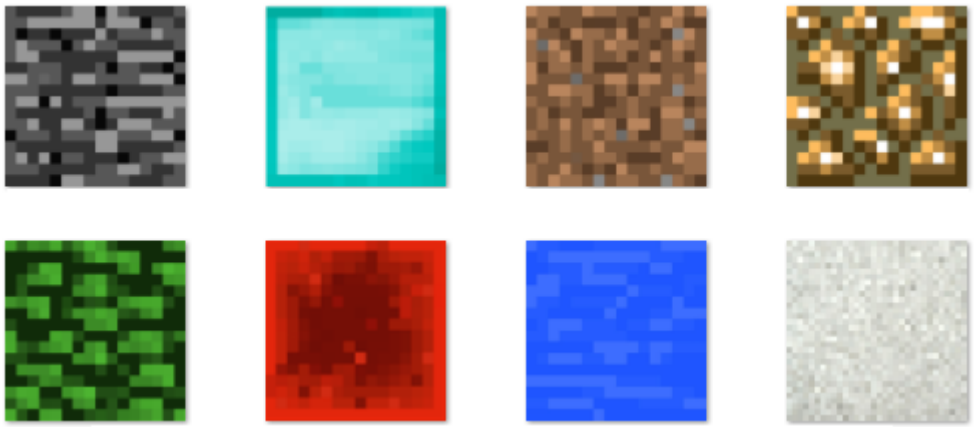
Listing 6 Metoda wykonującej obrót dla SBlock

```
private enum Position
{
    Horizontal,     //   Horizontal    Vertical
    Vertical        //                   0
                    //        1 3        1 2
                    //      0 2            3
}
public override void Rotate()
{
    int i = -1, j = -1;
    switch (actualPosition)
    {
        case Position.Horizontal:           // GOTO -> Position.Horizontal
            i = usedCells[1].Item1;
            j = usedCells[1].Item2;
            if (i + 1 == boardHeight - 1)
                i--;

            if (j - 1 >= 0)
            {
                if ((gameBoard[i, j + 1].CellStatus != CellStatus.Filled)
                   && (gameBoard[i + 1, j].CellStatus != CellStatus.Filled)
                   && (gameBoard[i + 1, j - 1].CellStatus != CellStatus.Filled))
                {
                    FreeCells();
                    usedCells[0] = new Tuple<int, int>(i, j + 1);
                    usedCells[1] = new Tuple<int, int>(i, j);
                    usedCells[2] = new Tuple<int, int>(i + 1, j);
                    usedCells[3] = new Tuple<int, int>(i + 1, j - 1);
                    actualPosition = Position.Vertical;
                    TakeCells();
                }
            } break;
        case Position.Vertical:           // GOTO -> Position.Vertical
            i = usedCells[1].Item1;
            j = usedCells[1].Item2;
            if (i - 1 < 1)
                i++;
            else if (i + 1 == boardWidth - 1)
                i--;

            if (j - 1 >= 0)
            {
                if ((gameBoard[i - 1, j - 1].CellStatus != CellStatus.Filled)
                   && (gameBoard[i, j - 1].CellStatus != CellStatus.Filled)
                   && (gameBoard[i + 1, j + 1].CellStatus != CellStatus.Filled))
                {
                    FreeCells();
                    usedCells[0] = new Tuple<int, int>(i - 1, j - 1);
                    usedCells[1] = new Tuple<int, int>(i, j);
                    usedCells[2] = new Tuple<int, int>(i, j - 1);
                    usedCells[3] = new Tuple<int, int>(i + 1, j);
                    actualPosition = Position.Horizontal;
                    TakeCells();
                }
            } break;
    }
}
```

Wracając do obiektów klasy **GameBoardCell**, każdy z nich przechowuje informację o zajętości komórki i sześcian. Każdy sześcian, to obiekt klasy **MyCube**. Podczas jego tworzenia podaje się jego rozmiar, jednak w tym projekcie wszystkie są rozmiaru 1.0. Aby uatrakcyjnić grę wizualnie, klocki powinny różnić się od siebie nie tylko kształtem, ale też kolorem. Postanowiono więc tworzyć sześciany, których ściany pokryte są teksturami. W tym celu wykorzystano tekstury pokazane na rysunku 2.



Rysunek 2 Tekstury wykorzystane w implementacji

Tekstury przygotowano za pomocą programu GIMPPortable. Każdą przeskalowano do rozmiaru 512x512 pikseli oraz wyeksportowano do formatu 24 bitowego formatu .bmp. W programie zostały one wczytane za pomocą statycznej (i jedynej) metody klasy **ContentPipe**, która została pokazana na listingu 7.

Podsumowując, każdy **BaseBlock** składa się z 4 obiektów **MyCube**, a zarazem mieści się w 4 obiektach **GameBoardCell**. Obiekty **GameBoardCell** tworzą obiekt klasy **GameBoard**, który jest jeden w klasie **Game**, która jest oknem stworzonej gry.

Listing 7 Metoda ładująca teksturę do pamięci OpenGl'a

```
class ContentPipe
{
    public static int LoadTexture(string path)
    {
        if (!File.Exists("Textures/" + path))
            throw new FileNotFoundException("File not found at 'Textures/'" +
path);

        int id = GL.GenTexture();
        GL.BindTexture(TextureTarget.Texture2D, id);

        Bitmap bmp = new Bitmap("Textures/" + path);
        BitmapData data = bmp.LockBits(new Rectangle(0, 0, bmp.Width,
bmp.Height), ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format24bppRgb);

        GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgb,
data.Width, data.Height, 0, OpenTK.Graphics.OpenGL.PixelFormat.Bgr,
PixelType.UnsignedByte, data.Scan0);

        bmp.UnlockBits(data);

        GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureWrapS, (int)TextureWrapMode.Clamp);
        GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureWrapT, (int)TextureWrapMode.Clamp);

        GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
        GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);

        return id;
    }
}
```

3. Efekt końcowy

Rysunek 3 Zrzut ekranu z gry z planszą o szerokości 10 bloków

Rysunek 4 Zrzut ekranu z gry z planszą o szerokości 20 bloków

4. Wnioski

Wbrew początkowym założeniom o implementacji wykorzystującej OpenGL'a w wersji 4, zdecydowałem się na powrót do starszej wersji, która dzięki bibliotece OpenTK była przyjemniejsza w użytkowaniu. Wersja 4 choć nowsza i zapewne szybsza, wymaga bardzo dużej wiedzy, na zdobycie której brakło czasu. Stworzony projekt jednak działa, brak mu tylko warunku przegrania (bloki w najwyższym rzędzie będą się nakładać. Jednak dzięki możliwości regulowania wielkości planszy, całość dobrze nadaje się na zaimplementowanie rozgrywki wieloosobowej (im większa plansza, tym więcej graczy). Dzięki temu projektowi poznałem podstawowe zagadnienia z tworzenia gier komputerowych oraz przypomniałem sobie programowanie obiektowe, które podczas implementacji gry, pokazuje swoje możliwości, przez co znacznie ułatwia tworzenie gier.