Politechnika Świętokrzyska w Kielcach
Wydział Elektrotechniki, Automatyki i Informatyki

Programowanie w Języku C 2 - Projekt
Informatyka - II rok, Rok akademicki 2022/2023

Temat projektu:

Gra Tetris

Wykonał:

Mrówka Kacper

Grupa **2IB14D**

Spis treści

Spis treści

Spis listingów

Cel i realizacja projektu

Celem projektu z Podstaw Programowania 2 było napisanie implementacji gry Tetris w języku C/C++, z wykorzystaniem dowolnych bibliotek i środowisk programistycznych.

Jako język wybrałem **C++**, a do obsługi trybu graficznego wykorzystałem bibliotekę **SFML**. Projekt został napisany w środowisku programistycznym **Microsoft Visual Studio 2017**. Utworzony program działa na komputerach osobistych z systemem operacyjnym **Microsoft Windows** w wersjach **10** i **11**, ale powinien się też uruchamiać na niektórych z wcześniejszych wersji.

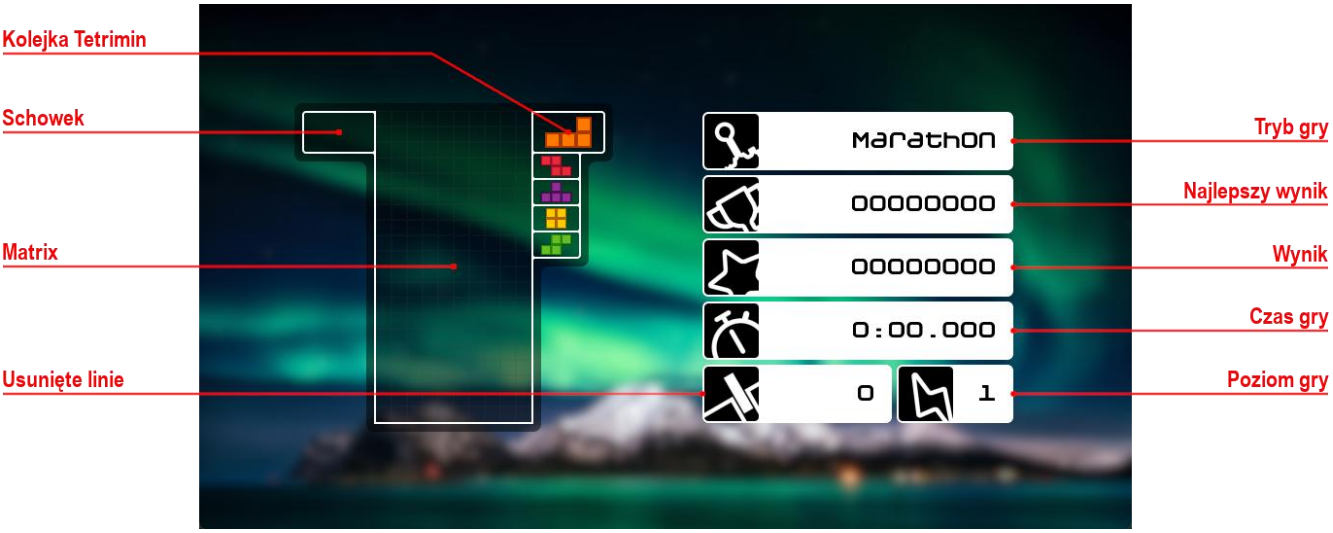
Grę można włączyć poprzez skompilowanie i uruchomienie rozwiązania w środowisku Microsoft Visual Studio 2017, albo z pliku wykonywalnego **TetrisPSk.exe**, znajdującego się w ścieżce **x64\Release**.

Czym jest Tetris?

Tetris to zręcznościowa gra logiczna stworzona przez programistę Aleksieja Pażytnowa w 1984 roku na komputery Elektronika 60. Od tamtej pory gra dostała liczne porty, edycje i nowe implementacje na wielu różnych platformach na całym świecie. Obecnie pieczę nad marką sprawuje założona przez Pażytnowa i Henka Rogersa firma The Tetris Company (skrótowo znane pod nazwą TTC).

Jak informuje samo TTC:

„Tetris to zwodniczo łatwa, całkowicie uzależniająca gra logiczna. Celem gry jest ustawianie spadających Tetrimin na dnie Matrixa. Tetrimina to kształty powstałe z połączenia czterech Min (bloków) ze sobą w jeden z siedmiu różnych wzorów. Tetrimina muszą być obracane podczas ich opadania i pozycjonowane na dnie Matrixa bez pozostawiania otwartych przestrzeni. Gdy cała pozioma linia zostanie wypełniona Minami, linia ta zostanie usunięta z Matrixa. Jeżeli linie nie będą w pełni uzupełnione Minami, nie będą one usuwane, a Tetrimina będą się układały w coraz wyższy stos. Jeżeli stos Tetrimin sięgnie wierzchu Matrixa, nastąpi koniec gry!”



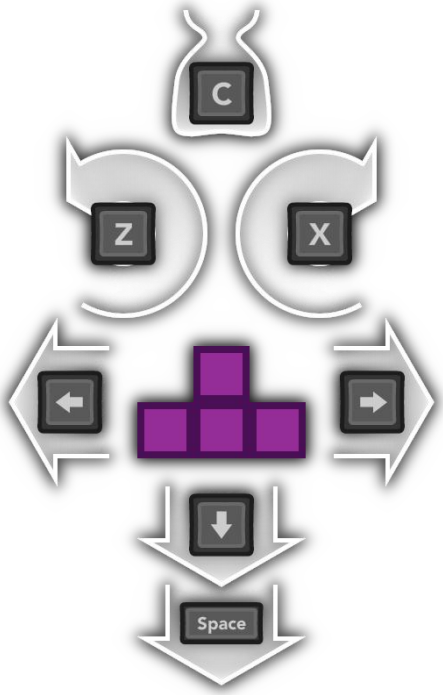
Ryc. 1 - Pierwsze spojrzenie na otwarty program z opisem elementów graficznych.

Opis funkcjonalności projektu

W momencie uruchomienia programu, ukaże nam się okno gry z pustym Matrixem i wyzerowanymi statystykami gry. Grę można rozpocząć, naciskając klawisz **Enter**. W tym momencie rozpocznie się odliczanie, po którego ukończeniu wygeneruje się pierwsze Tetrimino.

Sterowania aktywnym Tetriminem dokonuje się za pomocą poniższych klawiszy klawiatury:

Klawisz	Funkcja
Lewo	Przesunięcie Tetrimina w lewo
Prawo	Przesunięcie Tetrimina w dół
Z	Obrót Tetrimina przeciwnie do wskazówek zegara
X	Obrót Tetrimina zgodnie ze wskazówkami zegara
Dół	Soft Drop - opuszczanie Tetrimina w dół
Spacja	Hard Drop - zrzut Tetrimina w dół
C	Przechowanie Tetrimina w Schowku



Ryc. 2 - graficzne przedstawienie domyślnego sterowania w grze

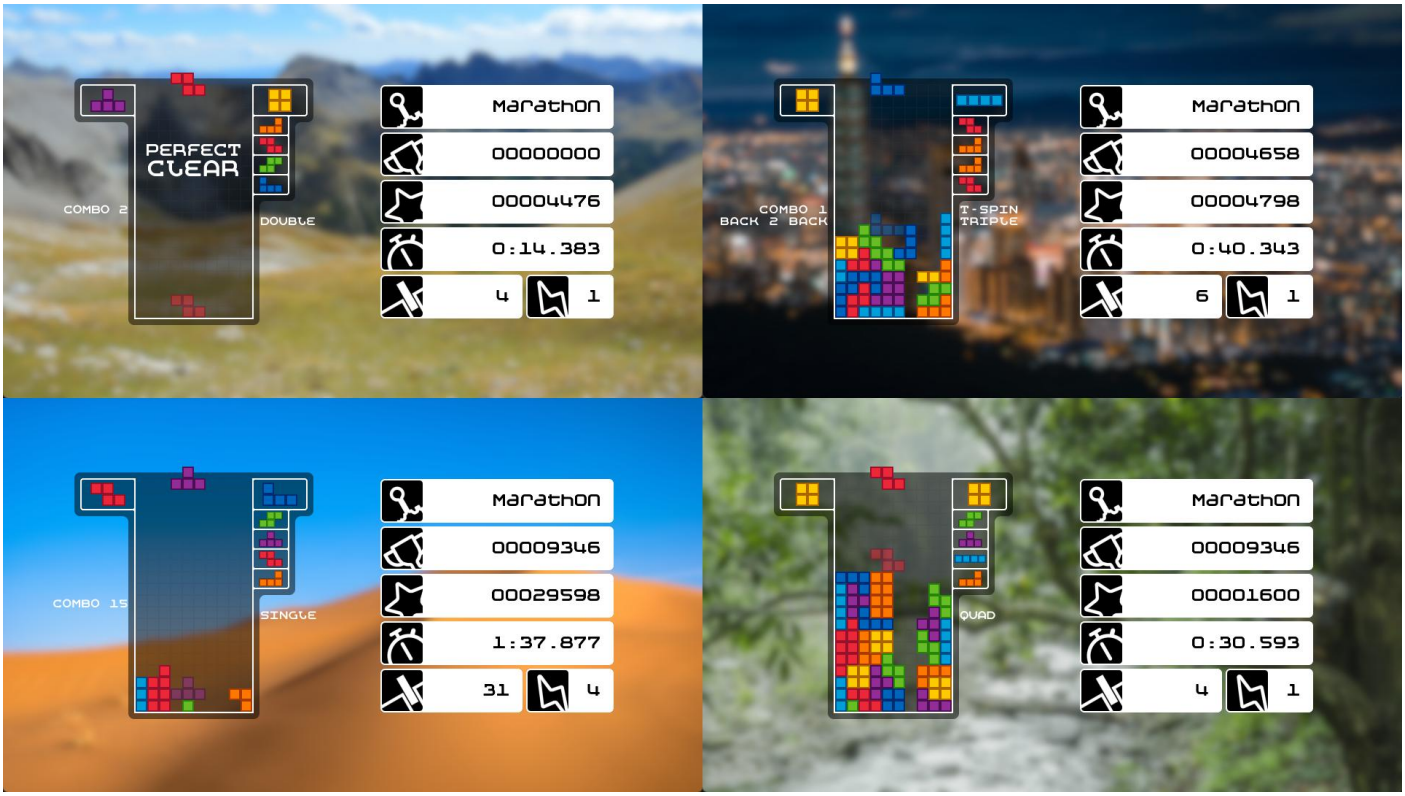
Gra rozpoczyna się z poziomu pierwszego (w pamięci gry notowany jako poziom zerowy), gdzie czas spadania Tetrimina wynosi 1 s/pole. Za każde 10 wyczyszczonych linii, gra zwiększa poziom o 1 do maksymalnej wartości 15 (w pamięci jako 14), jednocześnie skracając czas potrzebny na opuszczenie Tetrimina, do wyliczonego według wzoru:

$$(0{,}8 - (poziom \cdot 0{,}007))^{poziom}$$

gdzie numerem poziomu jest ten przechowywany w pamięci.
Przytrzymanie klawisza Soft Dropu powoduje tymczasowe skrócenie wymaganego czasu opadania 20-krotnie.

Podczas gry, gracz otrzymuje punkty za czyszczenie linii i wykonywanie specjalnych ruchów, według poniższej tabeli, gdzie numerem poziomu jest ten wyświetlany dla gracza:

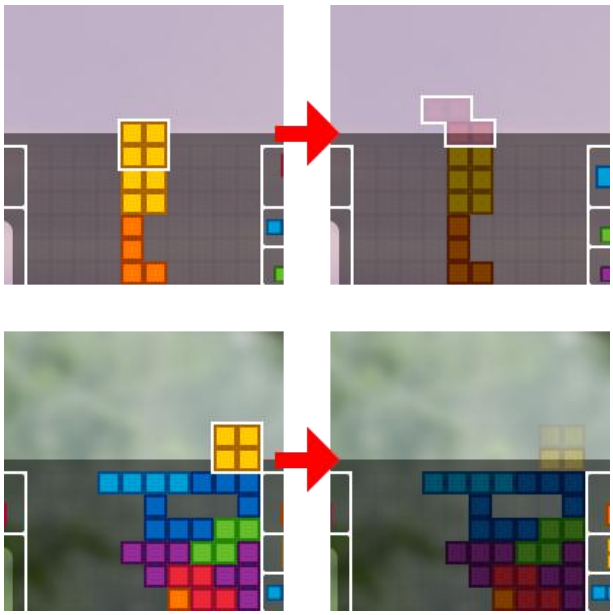
Nazwa	Punktacja	Opis	☆
Single	100 * poziom	Usunięcie 1 linii.	
Double	300 * poziom	Usunięcie jednocześnie 2 linii.	
Triple	500 * poziom	Usunięcie jednocześnie 3 linii.	
Quad	800 * poziom	Usunięcie jednocześnie 4 linii.	✓
Mini T-Spin	100 * poziom	Łatwiejszy rodzaj T-Spina, bez usunięcia linii.	✓
Mini T-Spin Single	200 * poziom	Łatwiejszy rodzaj T-Spina, usunięcie 1 linii.	✓
Mini T-Spin Double	1200 * poziom	Łatwiejszy rodzaj T-Spina, usunięcie 2 linii.	✓
T-Spin	400 * poziom	Wykonanie T-Spina, bez usunięcia linii.	✓
T-Spin Single	800 * poziom	Wykonanie T-Spina, usunięcie 1 linii.	✓
T-Spin Double	1200 * poziom	Wykonanie T-Spina, usunięcie 2 linii.	✓
T-Spin Triple	1600 * poziom	Wykonanie T-Spina, usunięcie 3 linii.	✓
Combo	50 * długość * poziom	Następujące natychmiast po sobie usuwanie linii.	
Bonus za **Back-2-Back**	0,5 * punkty za dany ruch	Następujące po sobie ruchy oznaczone w kolumnie ☆. Nie muszą być częścią Comba.	
Soft Drop	1 za każde pole w dół	Opuszczanie Tetrimina.	
Hard Drop	2 za każde pole w dół	Zrzut Tetrimina.	
Perfect Clear	3500 * poziom	Opróżnienie Matrixa.	



Ryc. 3 - prezentacja różnych aplauzów dla wykonywanych przez gracza ruchów

Gracz otrzymuje też gratulacje za każde usunięcie linii lub ruch specjalny poprzez animowane teksty po bokach Matrixa, informują o osiągnięciu.

Gra jest rozgrywana do osiągnięcia warunku końca gry. W jednoosobowej rozgrywce Tetrisa istnieją dwa takie warunki: **Block Out** i **Lock Out**. Lock Out polega na umieszczeniu Tetrimina całkowicie poza widzialną przestrzenią Matrixa. Block Out z kolei to zablokowanie miejsca, na którym powinno wygenerować się nowe Tetrimino. Spełnienie jednego z tych warunków kończy grę, a jeśli obecny wynik jest wyższy od poprzedniego najlepszego wyniku, jest on zapisywany na komputerze.



Ryc. 4 - porównanie ze sobą warunków końca gry; na górze Block Out, na dole Lock Out aktywne Tetrimino zaznaczone jest białą obwódką

Opis klas i ich najważniejszych metod

Na kod gry składa się kilka klas, reprezentujących różne elementy gry. Są to:

- Klasa **Mino** - reprezentująca pojedynczy blok na planszy lub w Tetriminie,
- Klasa **MinoAnchor** - reprezentująca najważniejsze Mino, które jest osią obrotu dla całego Tetrimina i wyznacza jego pozycję w Matrixie,
- Klasa **Player** - reprezentująca elementy gry należące do gracza, w tym Matrix, osobistą kolejkę, Schowek i statystyki,
- Klasa **Applause** - reprezentująca graficzne gratulacje dla gracza za jego działania,
- Klasa **pQueue** - reprezentująca wskaźnik na ogólnodostępną kolejkę Tetrimin,
- Klasa **Queue** - reprezentująca elementy ogólnodostępnej kolejki Tetrimin.

Większość funkcji używanych przez grę to metody zdefiniowane w klasie **Player**, ale najpierw omówmy funkcje z innych plików. Dokładniejsza dokumentacja dostępna jest w pliku *index.html* w folderze *html* w plikach źródłowych.

common.h / common.cpp

Pliki *common.h* i *common.cpp* nie reprezentują żadnej klasy, ale zawierają globalne zmienne, takie jak rozmiar widoku gry, wartości okresów ruchu Tetrimina, domyślne sterowanie i tablica przesunięć obrotu. Ostatni z tych elementów musi być najpierw zainicjowany, a dokonuje się tego za pomocą dostępnej w tych plikach funkcji *defineOffset(t)*. Funkcja zapisuje potrzebne wartości przesunięcia Tetrimina w momencie obrotu. Wizualnie tabela przedstawia się w ten sposób:

Przesunięcia obrotów dla Tetrimin J, L, S, T oraz Z					
	Test 1	Test 2	Test 3	Test 4	Test 5
0	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
R	[0, 0]	[+1, 0]	[+1, -1]	[0, +2]	[+1, +2]
2	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
L	[0, 0]	[-1, 0]	[-1, -1]	[0, +2]	[-1, +2]

Przesunięcia obrotów dla Tetrimina I					
	Test 1	Test 2	Test 3	Test 4	Test 5
0	[0, 0]	[-1, 0]	[+2, 0]	[-1, 0]	[+2, 0]
R	[-1, 0]	[0, 0]	[0, 0]	[0, +1]	[0, -2]
2	[-1, +1]	[+1, +1]	[-2, +1]	[+1, 0]	[-2, 0]
L	[0, +1]	[0, +1]	[0, +1]	[0, -1]	[0, +2]
Przesunięcia obrotów dla Tetrimina O					
	Test 1	Test 2	Test 3	Test 4	Test 5
0	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
R	[0, -1]	[0, -1]	[0, -1]	[0, -1]	[0, -1]
2	[-1, -1]	[-1, -1]	[-1, -1]	[-1, -1]	[-1, -1]
L	[-1, 0]	[-1, 0]	[-1, 0]	[-1, 0]	[-1, 0]

Znaki 0, R, 2, L symbolizują orientację Tetrimina: startowa, obrócone w prawo, obrócone dwukrotnie, obrócone w lewo. To, jak wyliczanie jest przesunięcie na podstawie tej tabeli omówię w dalszej części dokumentacji.

Queue

Klasa **Queue** reprezentuje elementy kolejki Tetrimin.

Klasa zawiera tylko pola z wartościami numeru ID obiektu, kształtu Tetrimina i wskaźnikiem na następny element.

pQueue

Klasa **pQueue** służy do stworzenia obiektu, który wskazuje na przód kolejki Tetrimin.

Klasa zawiera pola będące wskaźnikami na przód i tył kolejki, oraz metody służące d zakolejkowania obiektu **Queue**, odkolejkowania obiektu i dodania nowej torby Tetrimin do kolejki.

void enqueue(int d)

```
13  void pQueue::enqueue(int d)     // przykladowe wywolanie funkcji: enqueue(5);
14  {
15        Queue* newNode = (struct Queue*)malloc(sizeof(struct Queue));
16
17        if (newNode)
18        {
19              newNode->color = d;
20              newNode->next = NULL;
21
22              if (head == NULL)
23              {
24                    head = newNode;
25                    tail = newNode;
26                    newNode->id = 1;
27              }
28              else
29              {
30                    newNode->id = tail->id + 1;
31                    tail->next = newNode;
32                    tail = newNode;
33              }
34        }
35        else
36        {
37              std::cout << "Nowy element nie zostal utworzony!\n";
38        }
39  }
```

Listing 1 - wycinek kodu z pliku pQueue.cpp - enqueue(**int** d)

void *dequeue()*

```
41  void pQueue::dequeue()
42  {
43        if (head)
44        {
45              Queue* tmp = head->next;
46              free(head);
47              head = tmp;
48
49              if (tmp == NULL)
50              {
51                    tail = NULL;
52              }
53        }
54  }
```

Listing 2 - wycinek kodu z pliku pQueue - *dequeue()*

void *addBag()*

```
56  void pQueue::addBag()
57  {
58        /*
59        kolory i kształty Mino:
60              0 - Z - czerwony
61              1 - L - pomarańczowy
62              2 - O - żółty
63              3 - S - zielony
64              4 - I - błękitny
65              5 - J - niebieski
66              6 - T - fioletowy
67              7 - śmieciomino (obrażenia) - szary
68              8 - śmieciomino (limit czasu) - czarny
69        */
70
71        int bag[7] = { 0, 1, 2, 3, 4, 5, 6 };
72
73        for (int i = 0; i < 7; i++)
74        {
75              int rndPc = rand() % 7;
76              int swap = bag[rndPc];
77              bag[rndPc] = bag[i];
78              bag[i] = swap;
79        }
80
81        for (int i = 0; i < 7; i++)
82              enqueue(bag[i]);
83  }
```

Listing 3 - wycinek kodu z pliku pQueue.cpp - addBag()

Współczesne gry Tetris podążają za specjalnymi wytycznymi, które definiują jak ma wyglądać randomizacja kolejki. Zamiast losowo dobierać nowy kształt za każdym razem, gdy istnieje taka potrzeba, gra ma stworzyć wirtualny worek, w której znajduje się po jednym z każdego rodzaju Tetrimin. Gdy potrzeba dodać nowe Tetrimino do kolejki, jeden z kształtów jest losowo wyciągany z worka. W momencie opróżnienia worka, tworzona jest nowy. W ten sposób, gra zapewnia równą ilość każdego rodzaju Tetrimin oraz unika „susz” (okresów braku danego kształtu) i „powodzi” (okresów nadmiaru danego kształtu).

Metoda *addBag()* działa na tej zasadzie, ale gra, zamiast wyciągać co chwila Tetrimino z worka, randomizuje kolejność Tetrimin w worku i dodaje je wszystkie do kolejki poprzez metodę *enqueue(t)*.

Mino

Klasa **Mino** reprezentuje pojedynczy blok w grze. Dziedziczy z klasy **Sprite** należącej do biblioteki SFML.

Klasa składa się z pól opisujących pozycję obiektu, jego kolor i gotowość do wyświetlenia jako część Matrixa. Warto zwrócić uwagę na pola pozycji Mina. Mogą one być używane na dwa sposoby - jako punkt (0, 0) mogą być obrane Mino-Kotwica (jeżeli Mino jest aktualnie częścią sterowalnego Tetrimina) lub dolny lewy róg Matrixa (jeżeli Mino jest aktualnie częścią Matrixa).

Jedyną metodą jest *rotate(cw)*, które służy do obrócenia Mina wokół Mina-Kotwicy. Nową pozycję Mina można obliczyć poprzez zamienienie ze sobą wartości pozycji X i Y, oraz zmiana znaku na przeciwny jednej z tych wartości, w zależności od kierunku obrotu (Y, jeżeli w kierunku wskazówek zegara, X, jeżeli w przeciwnym).

void rotate(bool cw)

```
 8  void Mino::rotate(bool cw)
 9  {
10        if (cw) // ClockWise
11        {
12              int swap = posX;
13              posX = posY;
14              posY = 0 - swap;
15        }
16        else // Counter ClockWise
17        {
18              int swap = posX;
19              posX = 0 - posY;
20              posY = swap;
21        }
21  }
```

Listing 4 - wycinek kodu z pliku Mino.cpp - rotate(**bool** cw)

MinoAnchor

Klasa **MinoAnchor** reprezentuje Mino-Kotwicę - najważniejsze Mino w sterowalnym Tetriminie. Stanowi ono oś obrotu i punkt zerowy układu współrzędnych dla pozostałych Min. Klasa dziedziczy z klasy **Mino**.

W klasie znajdują się pola reprezentujące pozycję w Matrixie, orientację, kształt Tetrimina, gotowość do zalokowania i wskaźniki na Mina potomne oraz obiekt gracza.

Klasa zawiera destruktor i jedną metodę, która służy aktualizacji parametrów klasy **sf::Sprite** u siebie i u potomnych Min.

~MinoAnchor()

```
108  MinoAnchor::~MinoAnchor()
109  {
110        if (readyToLock)
111        {
112
113              // utworzenie min w matrixie
114              matrix->matrix[posX][posY] = new Mino();
115              matrix->matrix[posX + children[0]->posX][posY + children[0]->posY] = new
     Mino();
116              matrix->matrix[posX + children[1]->posX][posY + children[1]->posY] = new
     Mino();
117              matrix->matrix[posX + children[2]->posX][posY + children[2]->posY] = new
     Mino();
118
119              // zastosowanie pozycji min na ekranie
120              matrix->matrix[posX][posY]->setPosition(getPosition());
121              matrix->matrix[posX + children[0]->posX][posY + children[0]->posY]-
     >setPosition(children[0]->getPosition());
122              matrix->matrix[posX + children[1]->posX][posY + children[1]->posY]-
     >setPosition(children[1]->getPosition());
123              matrix->matrix[posX + children[2]->posX][posY + children[2]->posY]-
     >setPosition(children[2]->getPosition());
```

```
124
125             // ustawienie kolorów i duszków min
126             matrix->matrix[posX][posY]->colour = colour;
127             matrix->matrix[posX + children[0]->posX][posY + children[0]->posY]->colour =
    colour;
128             matrix->matrix[posX + children[1]->posX][posY + children[1]->posY]->colour =
    colour;
129             matrix->matrix[posX + children[2]->posX][posY + children[2]->posY]->colour =
    colour;
130
131             // usunięcie min sterowalnych
132
133       }
134       delete children[0];
135       delete children[1];
136       delete children[2];
137       matrix->activeAnchor = NULL;
138 }
```

Listing 5 - wycinek kodu z pliku MinoAnchor.cpp - *destruktor*

Destruktor ma dwa różne zachowania, zależnie od wartości zmiennej *readyToLock*. Jeżeli jest ona ustawiona na **true**, to oznacza, że Tetrimino jest gotowe do zalokowania się w Matrixie. W tym przypadku tworzy nowe obiekty **Mino** w Matrixie przechowywanym w obiekcie gracza, które dostają przeliczone na nowy układ współrzędnych pozycje XY i skopiowany numer koloru. W przeciwnym wypadku, kopiowanie nie jest dokonane, gdyż uznaje się, że destruktor został wywołany w celu schowkowania Tetrimina.

Na koniec, usuwane są oryginalne Mina, a wskaźnik na Mino-Kotwicę w obiekcie gracza jest ustawiony na wartość **NULL**.

Applause

Klasa **Applause** reprezentuje graficzne teksty gratulujące graczowi za wykonywane ruchy. W zależności od ich typu wyświetlają się po bokach Matrixa lub bezpośrednio na nim.

W klasie znajdują się pola reprezentujące tekst aplauzu (poprzez tablicę obiektów **Text** biblioteki SFML), długość tekstu, typ aplauzu, czas trwania, szybkości i opóźnienia animacji i widoczność.

Klasa zawiera trzy metody: czyszczącą obiekt do stanu z utworzenia, inicjującą potrzebny tekst i aktualizującą reprezentację graficzną.

***void** updatVis(**int** t)*

```
184 void Applause::updateVis(int t)
185 {
186       if (lifeTime <= 3000)
187       {
188             if (aType <= 1)
189             {
190                   for (int i = 0; i < len; i++)
191                   {
192                         txt[i].move(sf::Vector2f((0.0 - spd * (len - 1 - i)) * (t /
    1000.0), 0.0));
193                         if (lifeTime >= 1500)
194                         {
195                               txt[i].setFillColor(sf::Color(255, 255, 255, vis));
196                         }
197                   }
198             }
199             else if (aType <= 7)
200             {
201                   for (int i = 0; i < len; i++)
202                   {
203                         txt[i].move(sf::Vector2f(spd * i * (t / 1000.0), 0.0));
204                         if (lifeTime >= 1500)
```

```
205                         {
206                               txt[i].setFillColor(sf::Color(255, 255, 255, vis));
207                         }
208                   }
209             }
210             else if (aType <= 10)
211             {
212                   for (int i = 4; i >= 0; i--)
213                   {
214                         txt[i].move(sf::Vector2f(0.0 - spd * (5 - i) * (t / 1000.0),
    0.0));
215                         if (lifeTime >= 1500)
216                         {
217                               txt[i].setFillColor(sf::Color(255, 255, 255, vis));
218                         }
219                   }
220
221                   if (lifeTime >= 1500)
222                   {
223                         txt[5].setFillColor(sf::Color(255, 255, 255, vis));
224                   }
225
226                   for (int i = 6; i < 11; i++)
227                   {
228                         txt[i].move(sf::Vector2f(spd * (i - 5) * (t / 1000.0), 0.0));
229                         if (lifeTime >= 1500)
230                         {
231                               txt[i].setFillColor(sf::Color(255, 255, 255, vis));
232                         }
233                   }
234             }
235
236             if (lifeTime >= 1500)
237             {
238                   double temp = vis - (spdV * (t / 1000.0));
239                   if (temp <= 0)
240                         vis = 0;
241                   else
242                         vis = temp;
243                   spdV -= ((t / 1000.0) * accV);
244             }
245
246             spd -= ((t / 1000.0) * acc);
247
248             lifeTime += t;
249       }
250       else
251             clear();
252 }
```

Listing 6 - wycinek kodu z pliku Applause.cpp - updateVis(**int** t)

Jeżeli aplauz od momentu inicjalizacji przez *init(type, f, p)* był wyświetlany przez mniej niż 3 sekundy, podlega on animacji. Typy 0 - 1 rozsuwają litery w lewą stronę, typy 2 - 7 rozsuwają litery w prawą stronę, typy 8 - 9 rozsuwają litery od środka na boki. Wszystkie typy zaczynają zmniejszać swoją przezroczystość od 1,5 sekundy życia. Szybkość ruchu i zanikania jest aktualizowana co klatkę po animacji, obliczając nową prędkość korzystając do tego z wartości opóźnienia i podanego w parametrze czasu klatki.

Gdy upłyną 3 sekundy, obiekt jest resetowany przez metodę *clear()*.

Player

Klasa **Player** reprezentuje obiekt gracza i większość związanych z nim elementów gry. Jest to najobszerniejsza klasa z ponad 30 polami, zawierającymi m.in. Matrix będący tablicą Min, statystyki gry, czasomierze ruchu Tetrimina i tablicę aplauzów; oraz metodami opisującymi większość logiki gry.

Poniżej umieszczone są jedne z ważniejszych metod tej klasy.

***void** rotate(**bool** cw)*

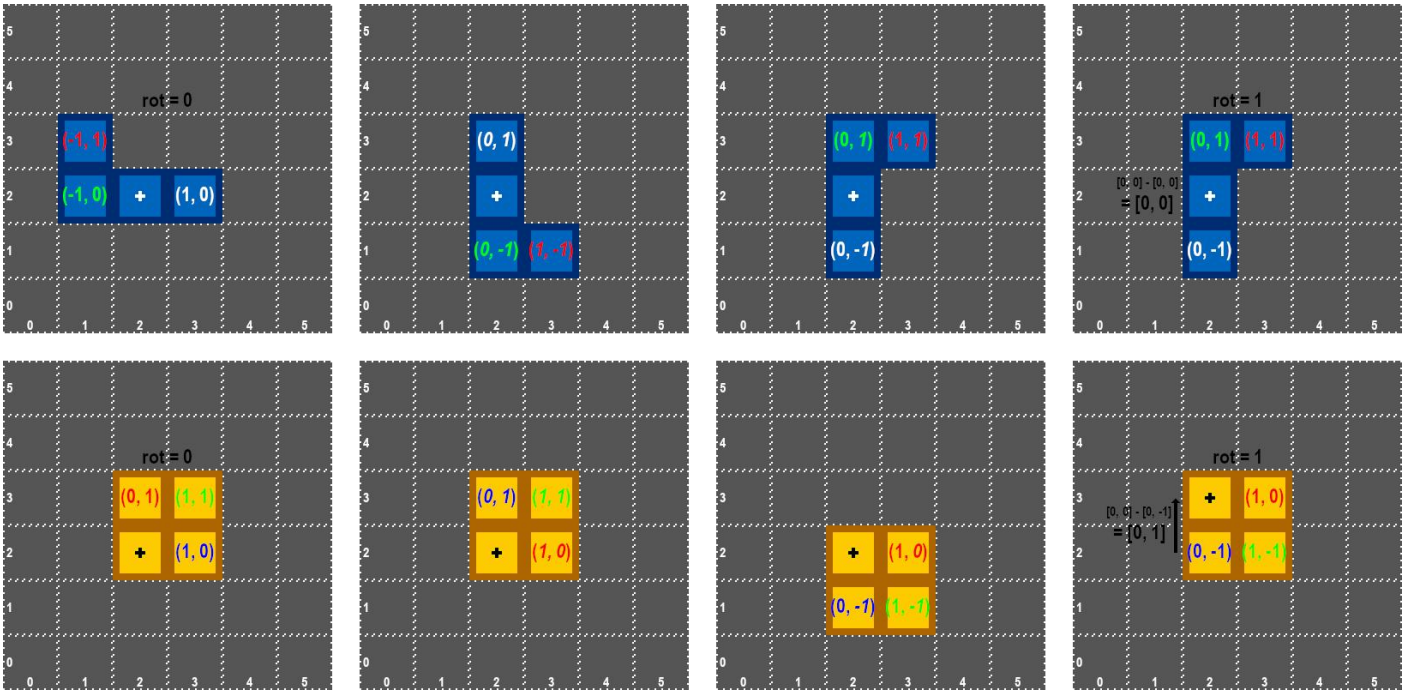
```
168 void Player::rotate(bool cw)
169 {
170 	int pom;
171 	if (cw == true)
172 		pom = 1;
173 	else
174 		pom = 0;
175 
176 	bool rotSuccess = false;
177 
178 	activeAnchor->children[0]->rotate(pom);
179 	activeAnchor->children[1]->rotate(pom);
180 	activeAnchor->children[2]->rotate(pom);
181 
182 	int xShift;
183 	int yShift;
184 
185 	for (int i = 0; i < 5; i++)
186 	{
187 
188 		if (activeAnchor->colour == 2)
189 		{
190 			xShift = offsetChart[2][activeAnchor->rot][i][0] -
    offsetChart[2][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][0];
191 			yShift = offsetChart[2][activeAnchor->rot][i][1] -
    offsetChart[2][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][1];
192 		}
193 		else if (activeAnchor->colour == 4)
194 		{
195 			xShift = offsetChart[1][activeAnchor->rot][i][0] -
    offsetChart[1][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][0];
196 			yShift = offsetChart[1][activeAnchor->rot][i][1] -
    offsetChart[1][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][1];
197 		}
198 		else
199 		{
200 			xShift = offsetChart[0][activeAnchor->rot][i][0] -
    offsetChart[0][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][0];
201 			yShift = offsetChart[0][activeAnchor->rot][i][1] -
    offsetChart[0][(activeAnchor->rot + 3 - (pom * 2)) % 4][i][1];
202 		}
203 
204 		if (checkPos(xShift, yShift))
205 		{
206 			activeAnchor->posX += xShift;
207 			activeAnchor->posY += yShift;
208 			activeAnchor->rot = (activeAnchor->rot + 3 - (pom * 2)) % 4;
209 			rotSuccess = true;
210 			tsMiniEligible = true;
211 			if (lockdownTime != -1) //resetowanie lock-ina
212 			{
213 				softDropTime = 0;
214 				lockdownReset++;
215 				lockdownTime = -1;
```

```
216                     }
217                     break;
218             }
219     }
220
221     if (!(rotSuccess))
222     {
223             activeAnchor->children[0]->rotate(1 - pom);
224             activeAnchor->children[1]->rotate(1 - pom);
225             activeAnchor->children[2]->rotate(1 - pom);
226     }
227 }
```

Listing 7 - wycinek kodu z pliku Player.cpp - rotate(**bool** cw)

Metoda *rotate(cw)* służy do obrócenia całego Tetrimina i umieszczenia go w poprawnym miejscu, bądź odmowy obrotu, gdy nie można znaleźć miejsca do legalnego umieszczenia Tetrimina.

Kod zaczyna się od zmienienia pozycji Min składowych Tetrimina, tak by układały się w obrócony kształt. Do tego służy metoda o tej samej nazwie będąca częścią klasy **Mino**. W zależności od tego, który kształt ma Tetrimino, metoda oblicza przesunięcie na podstawie tabeli przesunięć *offsetChart* (która jest definiowana na początku programu za pomocą wcześniej omówionej funkcji *defineOffset()*). Przesunięcie oblicza się jako różnica przesunięcia orientacji wejściowej i przesunięcia orientacji wyjściowej. Następnie wywoływana jest metoda *checkPos(xSh, ySh)*, która sprawdza czy dane przesunięcie pozwala na ustawienie Tetrimina tak, by nie kolidowało z istniejącymi na Matrixie Minami. Przy porażce, przesunięcie jest liczona ponownie z nowymi wartościami tabeli. Przy sukcesie nowa pozycja jest zapisana w Tetriminie i czasomierz lokowania zostaje zresetowany, jeśli wcześniej miał wartość nierówną -1, zwiększając jednocześnie licznik resetów. Przy 5 porażkach, pozycje Mino są przywracane do pozycji wejściowych.



Ryc. 5 - przykład działania obracania dwóch Tetrimin - J oraz O. Warto zauważyć, że Tetrimino O także się obraca, mimo, że wizualnie pozostaje niezmienne.

void** ctrlDo(**int** t,**sf::Texture&** ttr, **sf::Sound** s[], **sf::SoundBuffer sb[])*

```
229 // Poniżej tylko fragmenty warte uwagi, całość kodu dostępna w dokumentacji HTML
230
231 void Player::ctrlDo(int t, sf::Texture &ttr, sf::Sound s[], sf::SoundBuffer* sb[])
232 {
233         bool lockReset = false;
234
235         // Wykonaj funkcje przycisków
236         if (sf::Keyboard::isKeyPressed(ctrlKey["HLD"]))
237         {
238                 if (!ctrlState[6])
239                 {
```

```
240                     if (holdActive)
241                     {
242                             if (hold == 7)
243                             {
...                                     // (...) /* zmienienie wartości Hold na numer kształtu,
    usunięcie Tetrimina, dezaktywowanie Schowka, reset czasomierzy */
255                             }
256                             else
257                             {
...                                     // (...) /* zmienienie wartości Hold na numer kształtu,
    usunięcie Tetrimina, stworzenie nowego z poprzedniej wartości Hold, dezaktywowanie
    Schowka, reset czasomierzy */
275                             }
276
...                             // (...)
281                     }
282             }
283
284             ctrlState[6] = true;
285     }
286     else
287             ctrlState[6] = false;
288
289     if (sf::Keyboard::isKeyPressed(ctrlKey["HDR"]))
290     {
291             if (!ctrlState[5])
292             {
293                     int hdrDrop = 0;
294                     while (checkPos(0, -1))
295                     {
...                             //(...) /* przesunięcie w dół, dodanie do wyniku */
308                     }
309
310                     if (hdrDrop > 0)
311                             tsMiniEligible = false;
312
313                     lockdownReset = INT_MAX - LOCK_RESET;
314                     s[3].setBuffer(*sb[3]);
315                     s[3].play();
316                     activeAnchor->updateScreenXY();
317                     return;
318             }
319
320             ctrlState[5] = true;
321     }
322     else
323             ctrlState[5] = false;
324
325     if (sf::Keyboard::isKeyPressed(ctrlKey["LMV"]) ||
    sf::Keyboard::isKeyPressed(ctrlKey["RMV"]))  // jest wcisniety klawisz ruchu w bok
326     {
327             if (sf::Keyboard::isKeyPressed(ctrlKey["LMV"]) &&
    sf::Keyboard::isKeyPressed(ctrlKey["RMV"])) // oba przyciski wciśnięte
328             {
329                     if (ctrlState[0] && ctrlState[1])      // poprzednio oba
330                     {
331                             sideMoveTime += t;
332
333                             if (domDir == -1) // prawy guzik został dociśnięty do lewego
334                             {
335                                     if (sideMoveTime >= DAS_DEFAULT)
336                                     {
337                                             sideMoveTime -= ARR_DEFAULT;
338                                             if (checkPos(1, 0))
339                                             {
```

```
340                                             tsMiniEligible = false;
341                                             activeAnchor->posX += 1;
342                                             if (lockdownTime != -1) //resetowanie lock-
    ina
343                                             {
344                                                     softDropTime = 0;
345                                                     lockdownReset++;
346                                                     lockdownTime = -1;
347                                             }
348                                             s[0].setBuffer(*sb[8]);
349                                             s[0].play();
350                                     }
351                             }
352                     }
353                     else // lewy guzik dociśnięty do prawego
354                     {
...                             //  (...) /* jak wyżej, ale z przesunięciem w lewo */
372                     }
373             }
374             else if (ctrlState[0]) // poprzednio lewo (teraz +prawo)
375             {
376                     sideMoveTime = 0;
377                     if (checkPos(1, 0))
378                     {
379                             tsMiniEligible = false;
380                             activeAnchor->posX += 1;
381                             if (lockdownTime != -1) //resetowanie lock-ina
382                             {
383                                     softDropTime = 0;
384                                     lockdownReset++;
385                                     lockdownTime = -1;
386                             }
387                             s[0].setBuffer(*sb[8]);
388                             s[0].play();
389                     }
390             }
391             else if (ctrlState[1]) // poprzednio praw (teraz +lewo)
392             {
...                     // (...) /* jak wyżej, ale z przesunięciem w lewo */
407             }
408             else // poprzednio żaden
409             {
410                     sideMoveTime = 0;
411             }
412
413             ctrlState[0] = true;
414             ctrlState[1] = true;
415     }
416     else if (sf::Keyboard::isKeyPressed(ctrlKey["LMV"])) // tylko lewo
417     {
418             sideMoveTime += t;
419
420             if ((ctrlState[0] && ctrlState[1]) || ctrlState[1]) // poprzednio oba
    lub tylko prawo
421             {
422                     domDir = -1;
423             }
424             else if (ctrlState[0])
425             {
426                     // nic się nie dzieje
427             }
428             else // poprzednio żaden
429             {
430                     domDir = -1;
431                     if (checkPos(-1, 0))
```

```
432                         {
433                               tsMiniEligible = false;
434                               activeAnchor->posX -= 1;
435                               if (lockdownTime != -1) //resetowanie lock-ina
436                               {
437                                     softDropTime = 0;
438                                     lockdownReset++;
439                                     lockdownTime = -1;
440                               }
441                               s[0].setBuffer(*sb[8]);
442                               s[0].play();
443                         }
444                   }
445
446                   if (sideMoveTime >= DAS_DEFAULT)
447                   {
448                         sideMoveTime -= ARR_DEFAULT;
449                         if (checkPos(-1, 0))
450                         {
451                               tsMiniEligible = false;
452                               activeAnchor->posX -= 1;
453                               if (lockdownTime != -1) //resetowanie lock-ina
454                               {
455                                     softDropTime = 0;
456                                     lockdownReset++;
457                                     lockdownTime = -1;
458                               }
459                               s[0].setBuffer(*sb[8]);
460                               s[0].play();
461                         }
462                   }
463
464                   ctrlState[0] = true;
465                   ctrlState[1] = false;
466             }
467             else if (sf::Keyboard::isKeyPressed(ctrlKey["RMV"])) // tylko prawo
468             {
...                   // (...) /* jak wyżej, ale z przesunięciem w prawo */
517             }
518       }
519       else // żaden klawisz ruchu w bok nie jest wciśnięty
520       {
521             sideMoveTime = 0;
522             domDir = 0;
523
524             ctrlState[0] = false;
525             ctrlState[1] = false;
526       }
527
528       if (sf::Keyboard::isKeyPressed(ctrlKey["LSP"]) &&
    sf::Keyboard::isKeyPressed(ctrlKey["RSP"])) // oba przyciski obrotu wciśnięte
529       {
530             if (ctrlState[2] && ctrlState[3]) // poprzednio oba
531             {
532                   // nie dzieje sie nic
533             }
534             else if (ctrlState[2])    // poprzednio tylko lewo (teraz +prawo)
535             {
536                   rotate(true);
537                   if (activeAnchor->colour == 6)
538                   {
539                         if (checkTSpin() >= 0)
540                         {
541                               s[1].setBuffer(*sb[10]);
542                               s[1].play();
```

```
543                         }
544                         else
545                         {
546                               s[1].setBuffer(*sb[9]);
547                               s[1].play();
548                         }
549                   }
550                   else
551                   {
552                         s[1].setBuffer(*sb[9]);
553                         s[1].play();
554                   }
555             }
556             else if (ctrlState[3])    // poprzednio tylko prawo (teraz +lewo)
557             {
...                   // (...) /* jak wyżej, ale rotate(false);
577             }
578
579             ctrlState[2] = true;
580             ctrlState[3] = true;
581       }
582       else if (sf::Keyboard::isKeyPressed(ctrlKey["LSP"])) // lewy obrót
583       {
584             if (!ctrlState[2]) // poprzednio nie był trzymany lewy
585             {
...                   // (...) /* jak wyżej przy obrocie w lewo */
605             }
606
607             ctrlState[2] = true;
608             ctrlState[3] = false;
609       }
610       else if (sf::Keyboard::isKeyPressed(ctrlKey["RSP"])) // prawy obrót
611       {
612             if (!ctrlState[3]) // poprzednio nie był trzymany prawy
613             {
...                   // (...) /* jak wyżej przy obrocie w prawo */
633             }
634
635             ctrlState[2] = false;
636             ctrlState[3] = true;
637       }
638       else // żaden
639       {
640             ctrlState[2] = false;
641             ctrlState[3] = false;
642       }
643
...       // (...)
655
656       // Zaktualizuj grawitację
657
658       if (sf::Keyboard::isKeyPressed(ctrlKey["SDR"]))
659       {
660             softDropTime += (SDF_DEFAULT * t);
661             ctrlState[4] = true;
662       }
663       else
664       {
665             softDropTime += t;
666             ctrlState[4] = false;
667       }
668
669       if (softDropTime >= softDropUpdate)
670       {
671             if (checkPos(0, -1))
```

```
672             {
673                     tsMiniEligible = false;
674                     softDropTime -= softDropUpdate;
675                     activeAnchor->posY -= 1;
676                     if (sf::Keyboard::isKeyPressed(ctrlKey["SDR"]))
677                     {
678                             score += SDRP;
679
680                             txt_stats[2].setString("");
681                             int len = std::to_string(score).length();
682                             for (int i = 0; i < (8 - len); i++)
683                                     txt_stats[2].setString(txt_stats[2].getString() + "0");
684                             txt_stats[2].setString(txt_stats[2].getString() +
    std::to_string(score));
685                             sf::FloatRect bounds = txt_stats[2].getLocalBounds();
686                             txt_stats[2].setOrigin(sf::Vector2f(bounds.width,
    bounds.height));
687                             txt_stats[2].setPosition(sf::Vector2f(1632.0, 540.0));
688                     }
689
690                     if (softDropTime >= softDropUpdate)
691                     {
692                             ctrlDo(0, ttr, s, sb);
693                             return;
694                     }
695
696                     s[3].setBuffer(*sb[2]);
697                     s[3].play();
698             }
699             else
700             {
701                     softDropTime = softDropUpdate - 1;
702                     if (lockdownTime == -1)
703                     {
704                             lockdownTime = 0;
705                     }
706                     else
707                     {
708                             lockdownTime += t;
709
710                             if (lockdownTime >= LOCK_TIME)
711                                     lockdownReset = LOCK_RESET + 1;
712                             // TODO - jeżeli większe niż czas na lockdown, zalokuj
713                     }
714             }
715     }
716
717     activeAnchor->updateScreenXY();
718 }
```

Listing 8 - wycinek kodu z pliku Player.cpp - *fragmenty* ctrlDo(**int** t,**sf::Texture&** ttr, **sf::Sound** s[], **sf::SoundBuffer*** sb[])

Metoda *ctrlDo(t, ttr, s, sb)* jest najbardziej rozbudowana ze wszystkich. Jej zadaniem jest przetwarzanie wciśnięć klawiszy sterowania na ruch Tetrimina w grze.

Jako pierwsze, testowane są wciśnięcia klawisza Schowka, a potem klawisza Hard Dropu. Przy każdym z tych testów, jeżeli test wciśnięcia kończy się sukcesem, kończy to wykonywanie metody. Klawisz Schowka kopiuje wartość kształtu aktywnego Tetrimina, niszcząc go, i tworzy nowe Tetrimino ze Schowka, jeśli takie istniało. Hard Drop nieustannie przesuwa Tetrimino w dół, aż do napotkania przeszkody. Oba z tych działań nie mogą być wykonane, jeżeli w poprzedniej klatce trzymany był ten sam przycisk. Ochrania to grę przed automatycznym zrzucaniem i schowkowaniem nowo wygenerowanych Tetrimin, jeśli dany klawisz był przytrzymany.

Potem sprawdzane są wciśnięcia klawiszy ruchu w bok. Jednoczesne wciśnięcie obu klawiszy nie daje żadnego efektu. Wciśnięcie przycisku, który nie był wciśnięty w poprzedniej klatce powoduje natychmiastowe przesunięcie Tetrimina o 1 pole w tym kierunku (o ile nie ma tam przeszkody).

Przytrzymanie klawisza powoduje sumowanie czasów klatek, w którym klawisz był trzymany, do otrzymania liczby większej niż wartość DAS (Delayed Auto Shift - czasu minimalnego trzymania przycisku, by zacząć ciągły ruch), co powoduje kolejny ruch w bok o 1 pole (o ile nie ma tam przeszkody). Dalsze trzymanie klawisza sumuje czasy klatek do otrzymania liczby większej niż wartość ARR (Auto-Repeat Rate - odstępu pomiędzy kolejnymi przesunięciami w ruchu ciągłym). Dociśnięcie drugiego przycisku tymczasowo zmienia kierunek ruchu.

Kolejno są sprawdzane wciśnięcia klawiszy obrotu. Jeżeli dany klawisz nie był wciśnięty w poprzedniej klatce, wykonywany jest test obrotu. Dodatkowo, jeśli obrót jest T-Spinem, odgrywany jest specjalny efekt dźwiękowy.

Dalej do czasomierza opadania dodawany jest czas klatki, pomnożony przez 20, jeżeli przytrzymany jest klawisz Soft Dropu. Ostatecznie, wykonywany jest test grawitacji - jeżeli czasomierz opadania przekracza wymaganą przez poziom ilość czasu, Tetrimino jest opuszczane w dół o 1 pole (o ile nie ma tam przeszkód, wtedy rozpoczęty jest czasomierz lokowania). Wymagany czas opadania jest odejmowany od czasomierza, a jeśli ten nadal przekracza wymaganą wartość, metoda jest wywołana rekurencyjnie, ze zmianą czasu klatki na 0. W ten sposób podczas jednej klatki Tetrimino może przesunąć się wiele pól w dół (co jest częste na wyższych poziomach gry), a jednocześnie można je swobodnie kontrolować na każdej wysokości (na przykład, jeśli celujemy w mały otwór schowany pod Minami, trzymając klawisz w bok, test ruchu zostanie wykonany na każdej wysokości, bo Tetrimino nie przeskoczy automatycznie w dół o kilka pól).

Każdy z tych testów kończy się zapisaniem stanów wciśnięcia klawiszy do tabeli, tak, żeby można było porównać je z nowymi stanami w kolejnej klatce. Na koniec metody, wywoływana jest metoda aktualizacji reprezentacji graficznej aktywnego Tetrimina.

void checkTSpin()

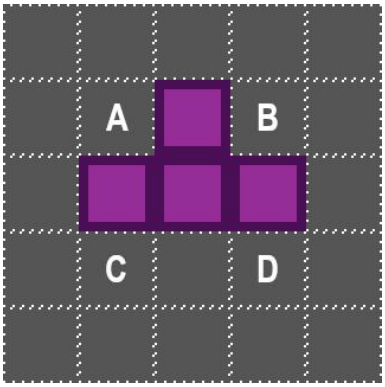
```
740 int Player::checkTSpin()
741 {
742         int tX = activeAnchor->posX;
743         int tY = activeAnchor->posY;
744 
745         switch (activeAnchor->rot)
746         {
747         case 0:
748         {
749               if ((matrix[tX - 1][tY + 1] != NULL) && (matrix[tX + 1][tY + 1] != NULL))
          // check for T-Spin
750               {
751                     if ((matrix[tX - 1][tY - 1] != NULL) || (matrix[tX + 1][tY - 1] !=
    NULL))
752                     {
753                           return 1;
754                     }
755               }
756               else if (tY == 0) // check for T-Spin Mini with wall
757               {
                     if ((matrix[tX - 1][tY + 1] != NULL) || (matrix[tX + 1][tY + 1] !=
758 NULL))
759                     {
760                           return 0;
761                     }
762               }
                     else if ((matrix[tX - 1][tY - 1] != NULL) && (matrix[tX + 1][tY - 1] !=
763 NULL)) // check for T-Spin Mini with minos
764               {
765                     if ((matrix[tX - 1][tY + 1] != NULL) || (matrix[tX + 1][tY + 1] !=
    NULL))
766                     {
767                           return 0;
768                     }
769               }
770               break;
771         }
```

```
772         case 1:
773         {
...                 // (...) /* jak wyżej, ale sprawdza pola z prawej i pola/ścianę z lewej */
796         }
797         case 2:
798         {
...                 // (...) /* jak wyżej, ale sprawdza pola z dołu i pola/ścianę z góry */
821         }
822         case 3:
823         {
...                 // (...) /* jak wyżej, ale sprawdza pola z lewej i pola/ścianę z prawej */
846         }
847         }
848
849         return -1;
850 }
```

Listing 9 - wycinek kodu z pliku Player.cpp - *fragmenty* checkTSpin()

Metoda *checkTSpin()* pozwala na sprawdzenie czy dany obrót Tetrimina T jest T-Spinem. Dana pozycja musi spełniać warunki:

- oba pola oznaczone literami A, B, muszą być zajęte przez istniejące Mina,
- przynajmniej jedno z pól C, D musi być zajęte przez istniejące Mino,
- ostatnie wykonane udane przesunięcie Tetrimina T musi być obrotem.



Ryc. 6 - narożniki wokół Tetrimina T oznaczone literami A, B, C, D w celu łatwiejszego rozpoznania

Ostatni warunek jest niemalże wymuszony przez pozostałe dwa, więc nie musi być koniecznie sprawdzany. W przypadku niepowodzenia, mogą być sprawdzone warunki wykonania T-Spina Mini, łatwiejszego wariantu tego ruchu:

- oba pola oznaczone literami C, D, muszą być zajęte przez istniejące Mina lub ścianę Matrixa,
- przynajmniej jedno z pól A, B musi być zajęte przez istniejące Mino,
- ostatnie wykonane udane przesunięcie Tetrimina T musi być obrotem.

Tu ostatni warunek musi być sprawdzony, ale jest to wykonywane poprzez pole *tsMiniEligible* - w momencie wykonania T-Spina Mini jest ustawiane na *true*, a w momencie dowolnego przesunięcia ustawione na *false*.

Źródło.cpp

Główny plik rozwiązania, zawierający pętlę *main()*. W niej zadeklarowane są wykorzystywane biblioteki, zdefiniowane tekstury, efekty dźwiękowe, muzyka i czcionka, a także zadeklarowane niektóre zmienne. Tu też znajduje się główna pętla programu, gdzie wywoływane są potrzebne metody odpowiedzialne za logikę gry, testowane są warunki końca gry, kontrolowane są zdarzenia bibliotek SFML i ostatecznie rysowane elementy gry.

Podsumowanie

Projekt został ukończony pomyślnie - kod się kompiluje, program się uruchamia i działa zgodnie z założeniami. Gra spełnia zdecydowaną większość znanych wytycznych gier Tetris. Mimo, że osobiście planowałem dodać więcej funkcjonalności, takie jak restartowanie rozgrywki, więcej trybów (np. Sprint, Ultra, tryb Klasyczny, Pojedynek) czy menu, nie udało mi się tego zrobić ze względu na zbyt małe doświadczenie z bibliotekami SFML i poświęcenie zbyt małej ilości czasu na wykonanie projektu.

Mimo to uważam projekt za zakończony. Podczas jego wykonywania, pogłębiłem moją wiedzę z korzystania z języka C++ i bibliotek SFML.