Tetris

Dokumentacja techniczna

Autor: Wiktor Lechowicz

Akademia Górniczo-Hutnicza

Wydział Informatyki, Elektroniki i Telekomunikacji

Elektronika

Kraków 2019

1.Wstęp

Program realizuje kopię popularnej gry Tetris w jej podstawowej wersji w oparciu o język programowania C/C++ oraz bibliotekę graficzną SFML w wersji 2.5. Część projektu wykorzystująca programowanie obiektowe ogranicza się jedynie do wykorzystania klas dostarczonych przez bibliotekę SFML. Do stworzenia programu zostało wykorzystane środowisko Visual Studio 2018. Program został wykonany docelowo na komputery z systemem operacyjnym Windows 10 64-bit o rozdzielczości ekranu nie mniejszej niż 1024x786.

2.Instrukcja użytkownika

Rozgrywka polega na układaniu opadających bloków w taki sposób, aby zapełniały kolejne wiersze prostokątnej planszy 20x10. Po wypełnieniu znikają bloki, które wcześniej go zajmowały. Użytkownik może zmieniać pozycję oraz rotację klocków przy użyciu klawiatury:

- strzałka w lewo – przesuwa blok w lewo,

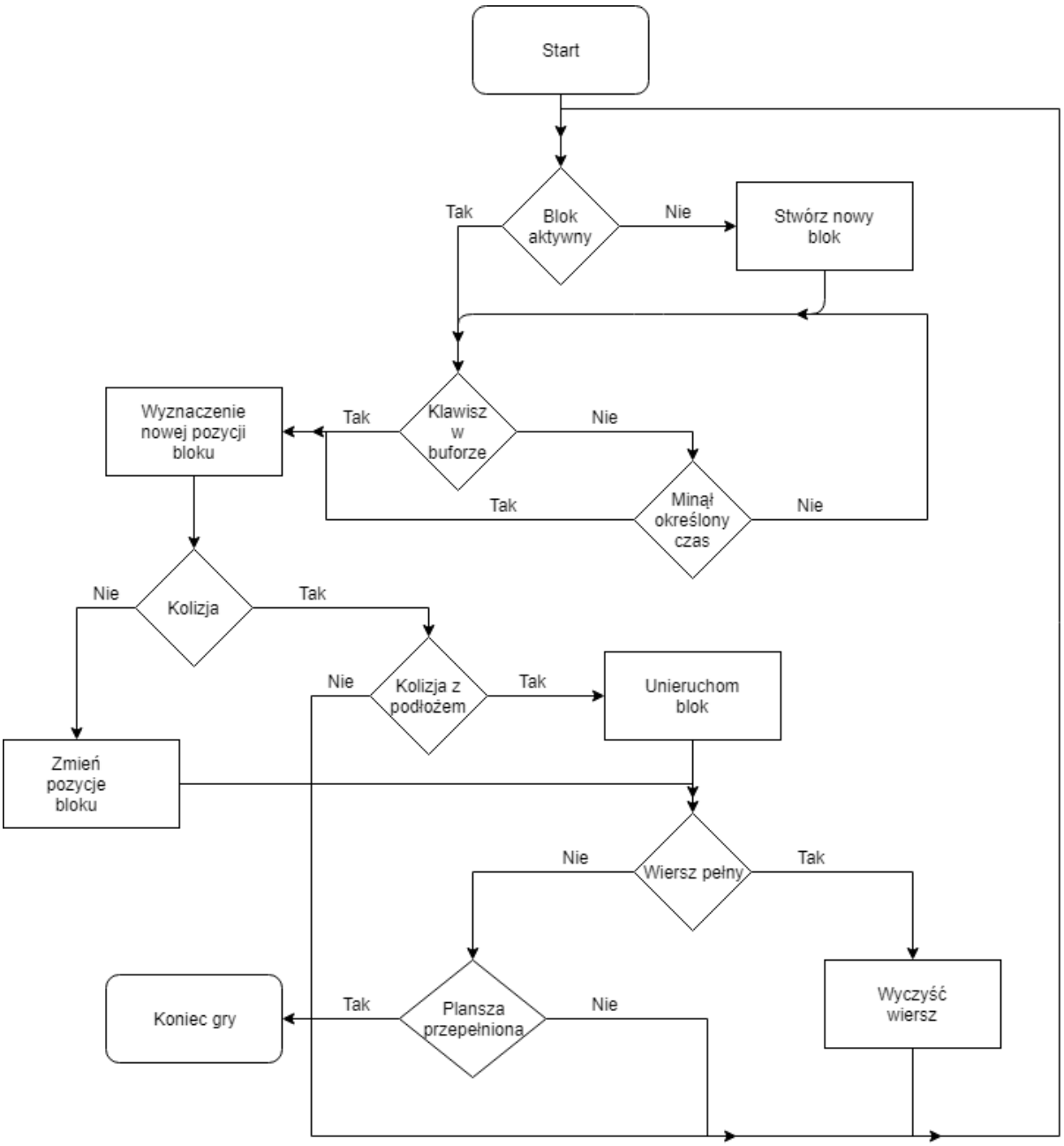
- strzałka w prawo – przesuwa blok w prawo,

- strzałka w dół – przyśpiesza opadanie bloku,

- strzałka w górę – obraca blok o 90 stopni

Zadaniem użytkownika jest nie dopuścić, aby klocki przepełniły planszę. Po przegranej należy nacisnąć dowolny klawisz aby opuścić grę, i uruchomić ją ponownie aby spróbować jeszcze raz.

3.Schemat blokowy działania



4.Opis realizacji

wersja 1.0 (26.06.2019)

Logika gry opiera się na tablicy o rozmiarach 22x10 reprezentującej planszę. Każda komórka zawiera dwucyfrową liczbę, gdzie liczba dziesiątek reprezentuje stan pola według zasady:

0 – wolna przestrzeń

1 – aktywny blok

2 – bloki unieruchomione

Liczba jedności reprezentuje natomiast kolor bloku.

Po wczytaniu i konfiguracji tekstur oraz czcionki program zaczyna działać w trybie nieskończonej pętli aż do zamknięcia okna lub zakończenia gry:

```
while (window.isOpen()) {

	//główny kod programu

}
```

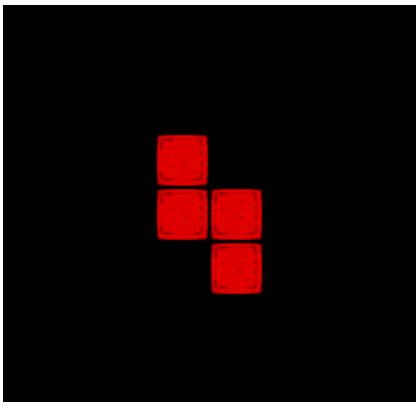
Na początku obiegu pętli, jeśli zachodzi taka potrzeba jest generwoany nowy blok:

```
if (!blockFalling) {
	blockFalling = true;
	n = std::rand() % 7 + 1;
	.
	.
	.
	}
	else if (n == 2) {  // S
		cords[0].x = cords[1].x = 4;
		cords[2].x = cords[3].x = 5;
		cords[0].y = 0;
		cords[1].y = cords[2].y = 1;
		cords[3].y = 2;
	}

	.

}
```

Kolor nowego bloku jest losowany jako liczba od 1 do 7, a jego współrzędne reprezentują zmienne klasy Vector. Powyższy fragment generuje klocek o kształcie „S”



Następnie program obsługuje klawisz z bufora wciśnięte przez użytkownika.

Przykładowo:

```
case sf::Event::KeyPressed:  // obsługa klawiatury

                            if (event.key.code == sf::Keyboard::Left)  // przesuwa blok w
lewo po nacisnieciu strzałki w lewo
                            {
                                    keyPressed = true;
                                    for (int i = 0; i < blockSize; i++)
                                    {
                                            if (cords[i].x == 0 ||
plane[cords[i].y][cords[i].x - 1] / 10 == 2)
                                                    noCollision = false;
                                    }
                                    if (noCollision)
                                    {
                                            for (int i = 0; i < blockSize; i++)
                                            {
                                                    cords[i].x = cords[i].x - 1;
                                            }
                                    }
                                    noCollision = true;
                            }
```

Powyższa procedura zmienia wartości zmiennych cords, czyli aktualne położenie bloku o ile na docelowym polu (tablciy „plane”)nie znajduje się przeszkoda.

Największym wyzwaniem okazało się obracanie klocków na planszy bez użycia zdefioniowanych wcześniej położeń bloków dla każdej rotacji. Zastosowane rozwiązanie wymagało stworzenia tymczasowych zmiennych do przechowywania poprzedniego ułożenia bloków na wypadek, gdyby po rotacji znajdowałyby się one poza planszą lub kolidowały z innymi blokami.

```
else if (event.key.code == sf::Keyboard::Up)
{
        keyPressed = true;
        for (int i = 0; i < blockSize; i++)
        {
                tempVec[i].x = cords[i].x;
                tempVec[i].y = cords[i].y;
        }
        rot = cords[2];
        for (int i = 0; i < blockSize; i++)
        {
                cords[i].x = -cords[i].y + rot.y + rot.x;
                cords[i].y = tempVec[i].x - rot.x + rot.y;
        }
        clearPlane(plane, pointsPointer);
        for (int i = 0; i < blockSize; i++) {
```

```
			if (cords[i].x < 0 || cords[i].x >= colNum ||
plane[cords[i].y][cords[i].x] / 10 != 0) {
				noCollision = false;
				break;
			}
		}
		if (!noCollision) {		// jeśli po obrocie blok kolidował by z innymi,
wraca do poprzedniej pozycji
			noCollision = true;
			for (int j = 0; j < blockSize; j++) {
				cords[j].x = tempVec[j].x;
				cords[j].y = tempVec[j].y;
			}
		}
}
```

Zmienna rot określa blok, wokół którego ma obracać się klocek. Pozycja każdego sześcianu zostaje obliczona za pomocą prostych przekształceń algebraicznych względem pozycji określonej w zmiennej rot. W przypadku kolizji, do współrzędnych bloków „cords” zostaje zwrócone zapisane wcześniej położenie.

Samoistne opadanie bloków zostało zaimplementowane przy użyciu obiektu klasy Clock dostarczonej wraz z biblioteką SFML.

```
if (clock.getElapsedTime() > fallingTime)
{
		clock.restart();
		for (int i = 0; i < blockSize; i++)
		{
			if (cords[i].y == verNum - 1 || plane[cords[i].y + 1][cords[i].x] / 10 ==
2)	//
				noCollision = false;
		}
		if (noCollision)
		{
			for (int i = 0; i < blockSize; i++)	// jeśli blok nie będzie
kolidował z innymi, zmienia jego pozycje
			{
				cords[i].y = cords[i].y + 1;
			}
		}
		else	// jeśli pod blokiem znajduje się kolejny, unieruchom go i odblokuj
generowanie kolejnego
		{
			noCollision = true;
			for (int i = 0; i < blockSize; i++)
			{
				plane[cords[i].y][cords[i].x] = 20 + n;
				cords[i].x = 0;
				cords[i].y = 0;
			}
			blockFalling = false;
		}
}
```

Ten fragment kodu z każdym obiegiem głównej pętli sprawdza czy upłynął określony czas, i jeśli tak to przesuwa klocek na doł, obsługując jednocześnie jego unieruchomienie jeśli pod nim znajdowała się przeszkoda.

Kolejnym etapem jest aktualizacja planszy, odpowiada za to funkcja clearPlane:

```
bool clearPlane(int plane[22][10], int *pts) {
	bool verseFilled{ true };
	for (int i = 0; i < 22; i++)
	{
		verseFilled = true;
		for (int j = 0; j < 10; j++) {
			if (plane[i][j] / 10 == 2 &&(i == 0 || i ==1)) {
				return true;
			}

			if (plane[i][j] / 10 != 2) {
				plane[i][j] = 0;
				verseFilled = false;
			}
		}
		if (verseFilled) {
			*pts = *pts + rand() % 100 + 100;
			for (int k = i; k > 0; k--) {
				for (int l = 0; l < 10; l++) {
					plane[k][l] = plane[k - 1][l];
				}
			}
		}
	}
	return false;
}
```

Usuwa ona wypełnione wiersze oraz przyznaje za to punkty. Dodatkowo jeśli plansza zostanie przepełniona (wiersz zerowy i pierwszy są ukryte przed użytkownikiem) funkcja zwraca wartość „true” co w konsekwencji wywołuję obsługę „końca gry”.

Po obsłudze mechaniki zostaje wyświetlony obraz odpowiadający stanowi planszy zapisanemu w tablciy zmiennych integer „plane”.

```
// czyszczenie ekranu
window.clear(sf::Color::Black);
// rysowanie do bufora
window.draw(spriteBack);
text.setString("WYNIK: \n" + std::to_string(points));
window.draw(text);
for (int i = 0; i < verNum; i++) {
	for (int j = 0; j < colNum; j++) {
		if (plane[i][j] / 10 == 1 || plane[i][j] / 10 == 2)
		{
			switch (plane[i][j] % 10) {
			case 1:
				sprite.setColor(sf::Color::Blue);
				break;
			case 2:
				sprite.setColor(sf::Color::Red);
				break;
			case 3:
				sprite.setColor(sf::Color::Yellow);
				break;
			case 4:
				sprite.setColor(sf::Color::Green);
				break;
			case 5:
				sprite.setColor(sf::Color::Magenta);
				break;
			case 6:
				sprite.setColor(sf::Color::Cyan);
				break;
			case 7:
				sprite.setColor(sf::Color::Color(122, 222, 50, 255));
				break;
			}
			sprite.setPosition(j * tileSize + planeOffset.x, i * tileSize +
planeOffset.y);
			window.draw(sprite);
		}
	}
}
window.draw(spriteMask);
//wyswietlanie obrazu
window.display();
```

Do wyświetlania obrazu klocków została użyta zaledwie jedna tekstura, na którą zostały nałożone maski o odpowiednich kolorach.

5.Wyniki testów programu.

Testy manualne nie wykazały żadnych nieprawidłowości w działaniu programu.

W przyszłości planuje się sprawdzić działanie na systemach operacyjnych innych niż Windows 10 64-bit.