Zajęcia V – sterowanie obiektem głównym (łazikiem)

1. Wymagania

Ocena	Kryteria
3.0	Realizacja prostego sterowanie przód-tył i obrót wokół własnej osi.
4.0	Implementacja prostej fizyki sterowania (w przypadku łazika różnica prędkości na gąsienicach lub oś skrętna)
5.0	Jak na ocenę 4 oraz implementacja podstawowych zagadnień fizycznych np. pęd ciała

2. Wyjaśnienie pomocne w realizacji:

W celu przemieszczenia łazika lub jego obrotu należy wykorzystać odpowiednio funkcje *glTranslatef()/glTranslated()* lub *glRotatef()/glRotated()*. Ponieważ chcemy poruszać łazikiem, a podłoże pozostawić bez żadnych przemieszczeń, niezbędne będą jeszcze funkcje *glPushMatrix()* oraz *glPopMatrix()*. Pierwsza z nich odkłada aktualnie zbudowaną macierz na stos. W uproszczeniu można mówić, że chodzi u o macierz punktów łączonych przez prymitywy. Jej odłożenie na stos powoduje, że wszystko co do tej pory było utworzone nie podlega przekształceniom wykonywanym między innymi przez funkcje *glTranslate* i *glRotate*. Aby zostały one wyrysowane bez zmian, należy je ściągnąć ze stosu za pomocą funkcji *glPopMatrix()*.

Aby lepiej to wyjaśnić zmodyfikowałem w przykładowym kodzie rysującym sześcian funkcję *RenderScene()*. Ma ona za zadanie wstawić walec, którego podstawy będą w płaszczyznach XY, na wysokości *Z=-10* oraz *Z=40*. Walec ten będzie cały czas stał nieruchomo. Obok niego znajduje się sześcian, który będzie przemieszczał się równolegle do osi Z w zakresie odpowiadającym wysokości walca. Utworzenie takiej animacji zostało przedstawione na poniższym fragmencie kodu.

<u>Uwaga:</u> funkcja *walec()* została w tym programie zmodyfikowana względem funkcji o takiej samej nazwie w udostępnionym projekcie. Pierwsza zmiana dotyczy uwzględnienia w parametrach współrzędnych drugiej podstawy (pierwotnie na sztywno przyjęta wartość 0). Druga miana, to przesunięcie rysowania walca w powierzchni XY, tak aby znajdował się obok sześcianu, a nie w tych samych współrzędnych.

W celu wykonania animacji ruchu sześcianu wzdłuż walca utworzono zmienną *pos_z*, która przechowuje aktualną wartość o ile sześcian jest przesunięty względem pierwotnego kodu w funkcji *szescian()*. Dodatkowo zmienna *phrase* przechowuje informację o ile wartość ta ma się zmienić w każdej klatce animacji. Zmienne te są globalne. Zmienna *pos_z* musi przechowywać aktualną pozycję sześcianu nawet po zakończeniu działania funkcji *RenderScene()*. Zmienna *phrase* nie musiałaby być zmienną globalną, ale postanowiłem obydwa te parametry ruchu mieć zapisane w jednym miejscu kodu, ponieważ w przyszłości może być konieczność zwiększenia zmiennej *phrase*. Działoby się tak, gdybym w innej funkcji chciał określić prędkość ruchu (regulacja przyciskiem klawiatury prędkości sześcianu lub łazika).

```
1  float pos_z = 0.1; //pozycja początkowa sześcianu float
2  phrase = 0.1; //wartość skoku sześcianu
3  void RenderScene(void)
4  {
```

5	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
	
	// Save the matrix state and do the rotations
	glPushMatrix();//początkowe warunki okna na stos glRotatef(xRot,
	1.0f, 0.0f, 0.0f);//obsługa kamery
	glRotatef(yRot, 0.0f, 1.0f, 0.0f);
	// MIEJSCE NA KOD OPENGL DO TWORZENIA WLASNYCH SCEN:
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	
	walec(10.0, -10.0, 40.0); //utworzenie walca glPushMatrix();
	//odłożenie walca na stos
	
	if (pos_z >= 40.0 || pos_z <= 0.0)//skrajne wartości ruchu
	{
	phrase = phrase * (-1);
	}
	pos_z += phrase;//przesunęcie sześcianu o wartość skoku
	
	glTranslatef(0.0, 0.0, pos_z);//przemieszczenie sześcianu szescian();
	
	glPopMatrix();//ściągnięcie
	//KONIEC MIEJSCA NA TWORZENIE WŁASNYCH SCEN
	
	glPopMatrix();//ściągnięcie początkowych warunków okna ze stosu
	glMatrixMode(GL_MODELVIEW); // Flush drawing commands
	glFlush();
	}

W linii 14 powyższego kodu wywołano funkcję *walec()*, która wstawia odpowiadnią figurę geometryczną. Będzie ona podlegała przekształceniom obrotu realizowanym w liniach 9 oraz 10. Natomiast następnie ten walec jest odłożony na stos funkcją *glPushMatrix()* z linii 15. Stąd też walec nie będzie podlegał żadnym przekształceniom występującym w kodzie, aż do linii 26, gdy zostaje zdjęty ze stosu. Oczywiście tłumaczenie to jest uproszczeniem, ponieważ nie sam walec jest odkładany na stos, tylko macierz punktów odpowiadająca za jego realizację.

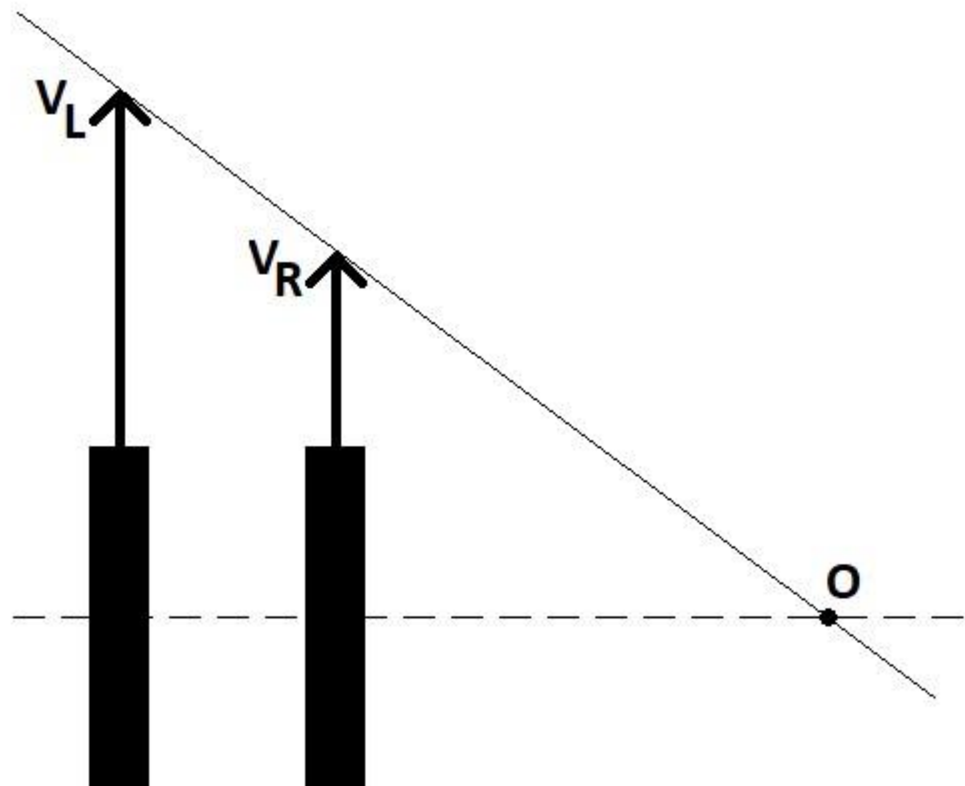
Warunek z linii 17 sprawdza, czy aktualne położenie sześcianu nie osiągnęło wartości skrajnych. Jeżeli je osiągnęło, zmienia znak liczby przechowywanej w zmiennej *phase* na przeciwny, co powoduje zmianę kierunku ruchu sześcianu.

Uwaga: powyższy kod będzie realizował ruch sześcianu tylko w przypadku gdy do programu będą przychodziły komunikaty WinApi zdefiniowane w funkcji *WndProc()*. Czyli np. przy obracaniu widoku przy wykorzystaniu klawiszy strzałek na klawiaturze lub naciśnięciu innego klawisza z klawiatury. Nawet takiego, na którego nie jest zdefiniowana reakcja. W takim przypadku wystarczy trzymać np. wciśnięty dowolny klawisz, aby obserwować animację. Taka

obsługa wystarczy na najniższe oceny. Natomiast nie da się w ten sposób zrealizować uwzględnienia pędu łazika. Ta sytuacja została omówiona niżej.

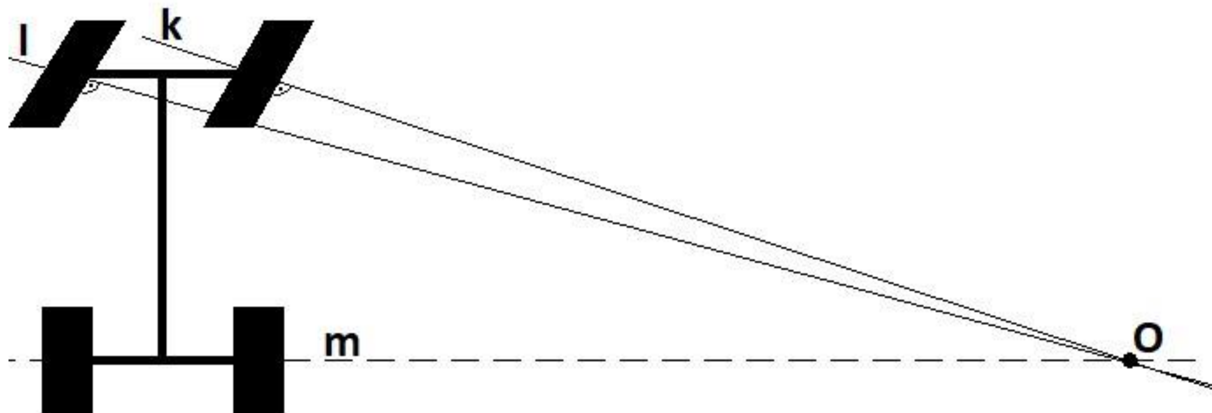
W celu realizacji sterowania łazikiem polecam wykorzystać mechanizm obsługi przycisków zastosowany w zadaniu 6 z zajęć „Wstęp do tworzenia prymitywów cz. 1”. Na ocenę 3.0 wystarczy zaimplementować funkcje ruchu do przodu i do tyłu przypisaną do dwóch przycisków np. „W” i „S” oraz dwóch kolejnych przycisków odpowiadających za obroty w prawo i w lewo (przykładowo „A” i „D”).

W realizacji na ocenę 4.0 należy zaimplementować fizykę, która będzie wyznaczała odpowiedni punkt wokół którego będzie obracał się łazik gdy będzie jednocześnie jechał do przodu lub tyłu i skręcał. Dla łazików gąsienicowych w tym również takich, które posiadają dwa rzędy kół (po dwa, trzy lub więcej z obu stron) i żadne z nich nie skręca się – środek obrotu jest zależny od prędkości gąsienic. Na rysunku 1 prędkości gąsienic oznaczono przez VL oraz VR. Jeżeli wektor prędkości zaczepimy w środku gąsienicy i poprowadzimy prostą przechodzącą przez końce tych wektorów, to środek obrotu jest punktem przecięcia tej prostej oraz prostej przechodzącej przez środki gąsienic. Tylko w przypadku, gdy wektory prędkości gąsienic będą miały te same wartości, ale przeciwne zwroty, to środek obrotu znajdzie się idealnie w środku odległości pomiędzy środkami gąsienic. Wyznaczenie wzorów, które należy zaimplementować pozostawiam jako część zadania.



Rysunek 1 Wyznaczenie środka obrotu dla pojazdów gąsienicowych

W przypadku pojazdów z osią skrętną – taką jak w samochodach - wyznaczenie środka obrotu jest zależne od kąta o jaki skręcone są koła tej osi. Jeżeli ze środka każdego z tych kół wyznaczymy prostą prostopadłą do ich boków (proste k i l) oraz poprowadzimy prostą przechodzącą przez środki kół osi nieskrętnej (prosta m), to wszystkie proste powinny przeciąć się w jednym punkcie, który będzie środkiem obrotu pojazdu – tak jak przedstawiono to na rysunku 2.



Rysunek 2 Wyznaczenie środka obrotu dla pojazdów z osią skrętną

W celu realizacji zadania na ocenę 5.0, należy uwzględnić pęd łazika. To znaczy, że po nadaniu mu pewnej prędkości i puszczeniu klawiszy za to odpowiedzialnych łazik nie zatrzyma się natychmiast, ale będzie prędkość tracił stopniowo. Można to zrealizować poprzez zastosowanie współczynnika tarcia µ, który może być równy np. µ=0,1 lub µ=0,01. Wartość prędkości wyznaczyć można wtedy ze wzoru:

$$V_{new} = \frac{1-\mu}{1} V_{old}$$

Aby animacja była realizowana również wtedy, gdy nie będzie wciśnięty żaden przycisk na klawiaturze, należy dodać timer, który co określony czas będzie przysłał komunikat. Komunikat ten będzie odpowiednio obsłużony w celu realizacji animacji.

Rozpocznijmy od utworzenia timera. W tym celu w funkcji *WinMain()*, przed pętlą *while* należy wstawić następujący kod:

```
// Set first timer
const WORD ID_TIMER =
1;
SetTimer(hWnd, ID_TIMER, 100, NULL);
```

Zmienna *ID_TIMER* oznacza wyłącznie numer timera. Stosując ten numer można go zatrzymać i jest niezbędny w przypadku zastosowania wielu timerów, ale w opracowywanej aplikacji raczej nie będzie potrzeby realizowania różnych animacji z różną częstotliwością. Trzeci parametr funkcji *SetTimer()* określa co ile milisekund nastąpi wysłanie komunikatu do tego timera. Aby obsłużyć komunikat musimy dodać modyfikację w funkcji *WndProc()*. W niej instrukcja warunkowa *switch()* opisuje sposób reagowania na przychodzące komunikaty. W niej należy utworzyć *case*, który będzie obsługiwał komunikaty wysyłane przez uruchomiony timer. W tym celu wystarczy dodać następujący fragment kodu:

```
case WM_TIMER:
{
        RenderScene();
        SwapBuffers(hDC);
        ValidateRect(hWnd, NULL);
        break;
}
```