Zajęcia VI – Wykrywanie kolizji

1. Wymagania

Ocena	Kryteria
3.0	Brak wymagań w zakresie wykrywania kolizji
4.0	Implementacja wykrycia kolizji z jedną przeszkodą
5.0	Implementacja wykrycia kolizji z co najmniej dwiema przeszkodami znajdującymi się w dwóch różnych obszarach sprawdzania odległości (grupowanie przeszkód w mniejsze obszary wykrywania kolizji)

2. Wyjaśnienie pomocne w realizacji:

Wykonując na wcześniejszych zajęciach sterowanie obiektem w przestrzeni trójwymiarowej nie trudno zauważyć, że może on przenikać przez inne obiekty. W celu poprawy stworzonego świata tak, aby bardziej odpowiadał rzeczywistemu należy wyeliminować zjawisko wzajemnego przenikania się. Służy do tego wykrywanie kolizji obiektów. W silnikach graficznych takich jak Unity lub Unreal algorytmy wykrywające kolizje są już zaimplementowane i zwalnia to programistę z tworzenia za każdym razem bardzo pracochłonnego procesu.

Przedmiotem tych zajęć jest zaznajomienie z zasadą działania takich algorytmów. Stąd w instrukcji zostały opisane podstawowe zasady tworzenia kodu mającego na celu wykrycie kolizji obiektów przy wykorzystaniu biblioteki OpenGL.

W prezentowanym przykładzie rozważymy środowisko, w którym znajduje się sześcian i walec. Pierwszy z nich jest obiektem, którym możemy sterować kombinacją klawiszy WASD dokonując jego przemieszczenia w osiach XY. Natomiast walec jest obiektem nieruchomym, z którym może nastąpić kolizja podczas ruchu sześcianem.

Aby zrealizować wykrywanie kolizji opisaną metodą należy tak zbudować obiekty, aby znany był ich środek. W przypadku sześcianu będzie to punkt przecięcia się jego przekątnych, w przypadku walca będzie to połowa wysokości. W tym celu w przykładzie „szescian" została zmodyfikowana funkcja szescian() (**linia 368**). W pierwotnej jej postaci w środku globalnego układu współrzędnych znajdował się jeden z wierzchołków sześcianu. W omawianym kodzie tak zmieniono wartości wierzchołków, aby każdy z nich znajdował się w jednakowej odległości od początku układu współrzędnych. Ich nowe wartości prezentuje poniższy fragment kodu:

```
GLfloat sa[3] = { -10.0f,-10.0f,10.0f };
GLfloat sb[3] = { 10.0f,-10.0f,10.0f }; GLfloat sc[3] = { 10.0f,10.0f,10.0f };
GLfloat sd[3] = { -10.0f,10.0f,10.0f };
GLfloat se[3] = { -10.0f,-10.0f,-10.0f };
GLfloat sf[3] = { 10.0f,-10.0f,-10.0f };
GLfloat sg[3] = { 10.0f,10.0f,-10.0f };
GLfloat sh[3] = { -10.0f,10.0f,-10.0f };
```

Zmodyfikowano również funkcję walec (**linia 434**) w taki sposób, że rysuje ona walec, którego wysokość jest równoległa do osi Z oraz na sztywno wewnątrz funkcji określono, że wysokość ta rzutowana na

płaszczyznę XY przetnie ją w punkcie o współrzędnych (20, 0) i są to zarazem współrzędne środka naszego walca.

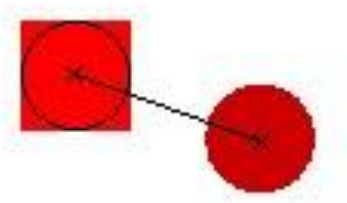
Następnym krokiem było utworzenie struktury (**linia 536**), która będzie przechowywała trzy wartości współrzędnych dowolnego punktu oraz wykorzystanie jej do zapamiętania w zmiennej środka sterowanego obiektu:

```
typedef struct wspolrzedne
{
	float x;
	float y;
	float z;
}Wspolrzedne;

Wspolrzedne polozenie_szescianu = { -2.1, 0.0, 0.0
}; float promien_szescianu = 10; float phrase = 0.1;
```

W powyższym fragmencie kodu poza określeniem środka położenia sześcianu w momencie uruchomienia programu (zostanie on o podane współrzędne przesunięty), utworzono również zmienną typu *float*, która przechowuje promień okręgu, który w płaszczyźnie XY będzie przybliżał obszar sześcianu.

Rzutując wszystkie obiekty na płaszczyznę XY, w której będzie poruszał się sześcian (nie uwzględniono ruchu w osi Z, co odpowiada podstawowym ruchom łazika) naszym zadaniem jest sprawdzenie, czy przemieszczający się kwadrat (rzut sześcianu) nie dotyka okręgu (rzut walca). W tym celu najłatwiej sprawdzić odpowiednią odległość pomiędzy ich środkami.



Rysunek 1 Rzut na płaszczyznę XY obiektów z zaznaczoną odległością pomiędzy ich środkami oraz przyjętym promieniem opisującym kwadrat jako okrąg wpisany w jego środek

W najprostszy sposób kolizję można wykryć poprzez obliczenie odległości pomiędzy środkami badanych obiektów i porównanie tej odległości z sumą promieni przybliżających obszar danego obiektu tak jak jest to pokazane na rysunku 1.

W tym celu proponuje się utworzenie funkcji obliczającej odległość pomiędzy dwoma punktami, które będą środkami obiektów. Zadanie to realizuje funkcja dystans_xy() (**linia 547**), w której do obliczenia odległości zastosowane miarę euklidesową:

```
float dystans_xy(Wspolrzedne object_a, Wspolrzedne object_b)
{

	return sqrt(pow((object_a.x - object_b.x), 2) + pow((object_a.y -
object_b.y), 2));
}
```

Następnie utworzono funkcję detekcja_kolizji() (**linia 553**), która przyjmuje jako parametr współrzędne środka sześcianu i sprawdza, czy nie koliduje on z żadnym innym obiektem. Jeżeli nie wykryto kolizji, to funkcja zwraca 0, natomiast w przypadku, gdy wykryto kolizję – zwraca 1.

Wewnątrz funkcji zdefiniowane są dwie tablice. Jedna przechowuje zmienne typu strukturalnego *Wspolrzedne* przechowujące informację o środkach poszczególnych obiektów, druga – zmienne typu *float* informujące o promieniu, jaki w przybliżeniu opisuje daną przeszkodę.

Prezentowany przykład został przygotowany w celach edukacyjnych. Zaleca się zastosowanie klasy opisującej całą przeszkodę (współrzędne oraz promień), a także utworzenie globalnej tablicy przeszkód tak, aby pamięć na nią nie była alokowana za każdym razem, gdy będziemy sprawdzać występowanie kolizji, ale tylko raz na początku działania programu, co zmniejszy zapotrzebowanie na moc obliczeniową.

```
int detekcj_kolizji(Wspolrzedne szescian)
{
	Wspolrzedne
polozenia_przeszkod[1];	float
promienie_przeszkod[1];	int
ilosc_przeszkod = 1;

	polozenia_przeszkod[0].x = 20.0;
	polozenia_przeszkod[0].y = 0.0;
	polozenia_przeszkod[0].z = 0.0;
	promienie_przeszkod[0] = 10;

	for (int i = 0; i < ilosc_przeszkod; i++)
	{
		if (dystans_xy(szescian, polozenia_przeszkod[i]) <=
(promien_szescianu + promienie_przeszkod[i]))
			return 1;
	}
	return 0;
}
```

Następnie funkcja w pętli for sprawdza, czy odległość pomiędzy sterowanym obiektem a kolejnymi przeszkodami nie jest mniejsza od sumy opisujących ich promieni, co oznaczałoby wystąpienie kolizji. Jeżeli takie zjawisko zostanie wykryte, funkcja zostaje zakończona (nie ma potrzeby sprawdzania kolizji z pozostałymi obiektami) oraz zwraca 1.

W prezentowanym prostym przykładzie występuje wyłącznie jedna przeszkoda, stąd rozmiar tablic *polozenia_przeszkod* oraz *promienie_przeszkod* wynoszą wyłącznie 1. W przypadku, gdy w przygotowywanych projektach również będzie występowała tylko jedna przeszkoda, można zastosować zmienne zamiast tablic.

Mając tak przygotowane rozpoznawanie można dodać obsługę wykrywania kolizji. W prezentowanym przypadku sterowanie sześcianem odbywa się przy wykorzystaniu klawiszy WASD (**linia 970**). W przypadku każdego z klawiszy procedura jest dokładnie taka sama:

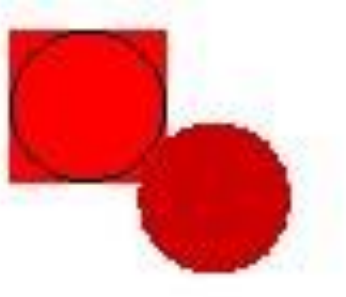
```
if (wParam == 0x41)//A
{
	Wspolrzedne nowe_polozenie_szescianu = polozenie_szescianu;
	nowe_polozenie_szescianu.x -= phrase;
	if(detekcj_kolizji(nowe_polozenie_szescianu) == 0)
	polozenie_szescianu.x -= phrase;
```

```
}
```

W pierwszej kolejności tworzona jest zmienna tymczasowa, która będzie przechowywała pozycję sześcianu jeżeli zostanie wykonane odpowiednie przemieszczenie zadanie przez użytkownika. Następnie sprawdzamy, czy dla takiego przemieszczenia nie zachodzą kolizje. Jeżeli nie, to w zmiennej globalnej *polozenie_szescianu* zostaje zmieniona pozycja sterowanego obiektu, a jeżeli zachodzą kolizję, to przemieszczenie nie ulega zmianie.

Uwaga dotycząca promienia przybliżającego kształt obiektu

W omawianym przykładzie kształt kwadratu został przybliżony przez okrąg wpisany w ten kwadrat. Skutkuje to tym, że kolizje z rogami kwadratu zostają wykryte późno i może dojść do sytuacji, w której ewidentnie kwadrat wnika w przeszkodę, a mimo to kolizja jeszcze nie zostaje wykryta. Sytuację tą przedstawiono na rysunku 2.



Rysunek 2 Zbyt mały promień przybliżający opisaną figurę co powoduje niewykrycie kolizji

Rozwiązaniem takiego problemu mogłoby być przybliżenie kształtu okręgiem opisanym na figurze, jednak to prowadzi do sytuacji, gdy kolizja zostaje wykryta, gdy obiekt tylko zbliży się bardzo blisko do przeszkody, ale nawet jej nie dotyka. Kompromisowym przypadkiem jest przyjęcie promienia o wartości średniej pomiędzy jednym, a drugim. Jeżeli długość boku kwadratu przyjmiemy jako *a*, to promień będzie miał wartość:

- Dla okręgu wpisanego w kwadrat:
$$\frac{a}{2}$$
- Dla okręgu opisanego na kwadracie:
$$\frac{a\sqrt{2}}{2}$$
- Dla średniej pomiędzy okręgiem wpisanym, a opisanym:
$$\frac{a + a\sqrt{2}}{4}$$

Zachęcam do sprawdzenia jaki wpływ na działanie wykrycia kolizji ma zastosowanie różnych promieni. Jest to szczególnie istotne dla łazików, które w rzucie na płaszczyznę XY będą miały kształt prostokąta.

Opisane przybliżenia w celu wykrycia kolizji były stosowane w grach z lat 90'. Obecnie przy dostępnej dużo większej mocy obliczeniowej oraz bardziej wyrafinowanych kształtach obiektów stosuje się metodę, w której obiekt przybliża się kształtem kuli, w której zawiera się cały (może to być to kula opisana na obiekcie lub jeszcze większa). Następnie sprawdza się, czy występują kolizje obiektów z tą kulą i dopiero po wykryciu takiej kolizji już tylko dla obiektów kolidujących z taką kulą sprawdza się dokładniej, czy nie występują kolizje z konkretnymi skrajnymi punktami obiektu sterowanego. W naszym przypadku można by to porównać do sprawdzenia w pierwszym kroku, czy występuje kolizja jakiejkolwiek przeszkody z okręgiem opisanym na kwadracie i w momencie jej wykrycia kolejne sprawdzenia już tylko dla tej przeszkody lub przeszkód z wierzchołkami kwadratu. Metoda ta sprawdziłaby się w przypadku, gdyby łazik nie wykonywał driftów. Dla bardziej zaawansowanego sterowania łazikiem należałoby sprawdzać, czy nie występują przecięcia charakterystycznych prostych lub krzywych dających się opisać matematycznie w układzie współrzędnych. Rozwiązania takie wymagają szerszej wiedzy teoretycznej.

Uwaga dotycząca wielu obiektów, z którymi może nastąpić kolizja (rozszerzenie do zadania na 5.0)

W przypadku bardzo rozbudowanych map, w których występują tysiące obiektów, z którymi może wystąpić kolizja byłoby bardzo duże obciążenie komputera sprawdzającego w każdym kroku możliwość wystąpienia kolizji z każdą nawet bardzo odległą przeszkodą. Aby temu zapobiec można podzielić mapę na mniejsze przestrzenie np. na ćwiartki układu współrzędnych i w zależności od miejsca w którym znajduje się łazik sprawdzać, czy nie występuje kolizja z inną tabelą przeszkód. W takim przypadku dla każdej ćwiartki układu współrzędnych istniałaby odrębna tabela przechowująca współrzędne środków przeszkód. Przy czym wybrane przeszkody znajdujące się na granicy tych przeszkód będą opisane przez więcej niż jedną tabelę.

Innym sposobem zapewniającym większy porządek w kodzie i tworzącym go bardziej uniwersalnym jest ustawienie w jednej tabeli przeszkód tak, aby były zgrupowane w zależności od obszarów w których się znajdują i będą w nich liczone odległości. Przykładowo dla 40 przeszkód oraz podziału wykrywania kolizji na 4 ćwiartki układu współrzędnych, gdy robot będzie znajdował się w pierwszej ćwiartce mogą być sprawdzane kolizje z przeszkodami od numeru 0 do numeru 11. Gdy robot będzie w drugiej ćwiartce – z przeszkodami o numerach 9-22. Dla trzeciej ćwiartki mogą to być przeszkody numer 23-35, a dla czwartej ćwiartki od numeru 33 do 4 (również z początku tablicy). Metoda ta sprawdza się bardziej dla przeszkód statycznych, natomiast dla obiektów dynamicznych (poruszających się) należałoby stworzyć drugą metodę wykrywania kolizji.

W zadaniu na ocenę 5.0 oczekuję utworzenia co najmniej dwóch obszarów na których będą wykrywane kolizje. Przykładowo gdy środek łazika będzie miał współrzędną $x \geq 0$ oraz drugi obszar dla $x < 0$ (w kwestii ustalenia obszarów jest pełna dowolność)

Jak wspomniałem na samym początku instrukcji w silnikach graficznych są zaimplementowane algorytmy odpowiedzialne za wykrywanie kolizji i we współczesnej pracy programistów nikt takich funkcji nie tworzy od nowa (bez szczególnych i bardzo specyficznych przyczyn). Wytłumaczenie w instrukcji ma na celu przybliżenie sposobu ich działania w celach edukacyjnych.