J2ME. Tworzenie gier

Autor: Janusz Grzyb
ISBN: 978-83-246-1263-5

Stwórz własną grę do telefonu komórkowego

- Poznaj technologię J2ME
- Zaprojektuj interfejs użytkownika dla gry
- Zaimplementuj mechanizmy wyświetlania grafiki 3D

Współczesne telefony komórkowe przestały być urządzeniami wykorzystywanymi wyłącznie do prowadzenia rozmów i wysyłania wiadomości SMS. Można je dziś stosować do wielu innych celów – pełnią rolę notatników, dyktafonów, aparatów fotograficznych, odtwarzaczy plików MP3 i przenośnych konsoli do gier. Jednak największe możliwości telefony komórkowe uzyskały dzięki zaimplementowaniu w nich języka Java. Mobilna Java pozwala nie tylko na tworzenie dodatkowych narzędzi, ale także gier – zarówno prostych platformówek, jak i skomplikowanych gier wykorzystujących złożone algorytmy wyświetlania grafiki trójwymiarowej.

Książka „J2ME. Tworzenie gier" to podręcznik, dzięki któremu opanujesz technologię J2ME i wykorzystasz ją do napisania własnej gry do telefonu komórkowego. Czytając ją, poznasz podstawy mobilnej Javy, zainstalujesz narzędzia niezbędne do pracy i dowiesz się, jak zbudowana jest aplikacja przeznaczona do wykorzystania w tym urządzeniu. Zaprojektujesz interfejs użytkownika i zastosujesz podstawowe metody generowania grafiki 2D. Poznasz także bibliotekę Java Mobile 3D i wykorzystasz ją do stworzenia prawdziwej gry z trójwymiarową grafiką, uwzględniającą tekstury obiektów, oświetlenie i cieniowanie. Dowiesz się również, jak budować gry dla wielu graczy łączących się ze sobą poprzez interfejs Bluetooth.

- Instalacja i konfiguracja narzędzi Wireless Toolkit, Ant i Antenna
- Tworzenie interfejsu użytkownika
- Rysowanie podstawowych obiektów graficznych
- Algorytm śledzenia promieni światła
- Reprezentacja obiektów 3D w Java Mobile 3D
- Modelowanie oświetlenia i cieniowania
- Animowanie obiektów
- Przygotowanie sceny gry w programie Blender
- Komunikacja poprzez interfejs Bluetooth
- Tworzenie gry typu „multiplayer"

Nie szukaj w sieci gier do swojej komórki – napisz własną!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Gry 3D w 2D — Wolfenstein

Jeszcze kilka lat temu, grając na swojej Nokii 3310 w kultowego już „węża", za fantazję uznałbym, gdyby ktoś powiedział mi wtedy, że za kilka lat na telefonach komórkowych królować będą gry 3D. A jednak! Najnowsze telefony komórkowe mają już tak duże moce obliczeniowe, że świat gier 3D stanął przed nimi otworem. Co więcej, niektóre urządzenia mobilne mają już wbudowane sprzętowe akceleratory grafiki 3D. Najnowsze telefony komórkowe udostępniają programistom interfejs Java Mobile 3D, ułatwiający w zasadniczym stopniu proces tworzenia aplikacji korzystających z grafiki trójwymiarowej.

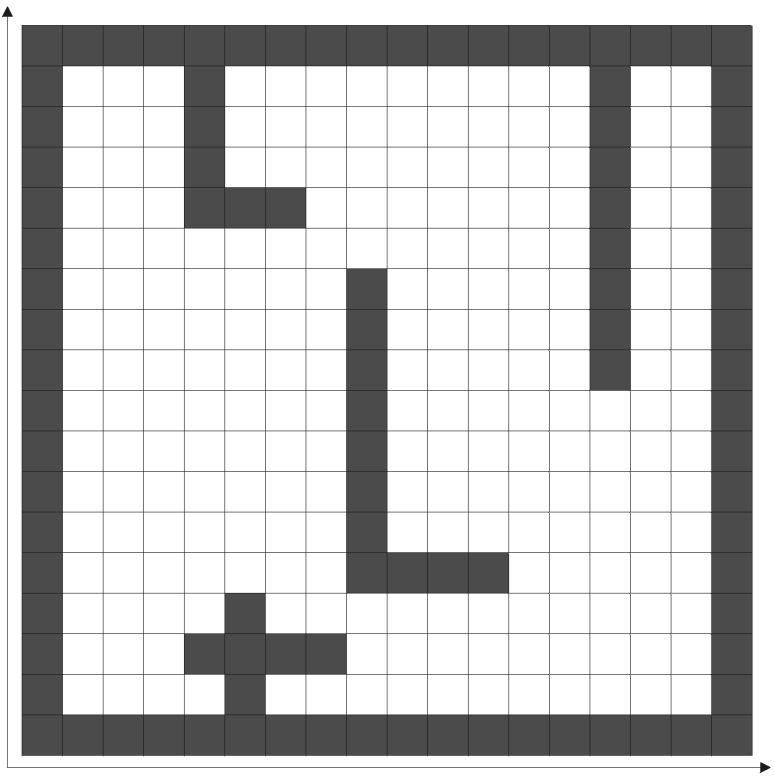
Czy można jednak stworzyć grę 3D na popularnych (czytaj: tańszych) modelach telefonów? Okazuje się, że przy zastosowaniu pewnych trików można osiągnąć efekt trójwymiarowości na telefonach o niskich mocach obliczeniowych nieposiadających dedykowanych ku temu API czy też sprzętowej akceleracji. Z pomocą przychodzi nam metoda raycastingu, znana z takich gier jak m.in. *Wolfenstein3D*, *Doom* i wiele innych tytułów z początku lat 90.

W rozdziale tym:

- dowiemy się, czym jest metoda raycastingu,
- poznamy zasady działania algorytmu *DDA* rzucania promieni,
- stworzymy symulację „spaceru" po trójwymiarowym labiryncie, takim jak znany z gier *Wolfenstein* czy *Doom*.

Raycasting

Świat gry w metodzie raycastingu reprezentowany jest przez dwuwymiarową siatkę, taką jak ta pokazana na rysunku 3.1. Siatka ta przedstawia rzut „z góry" labiryntu, po którym poruszają się bohaterowie gry. W pamięci komputera siatka ta może być reprezentowana



Rysunek 3.1. *Reprezentacja świata gry w metodzie raycastingu*

przez dwuwymiarową tablicę bajtów. Każde oczko siatki (element tablicy) określa, czy dany sektor (np. o rozmiarze 1×1 metr) jest pusty, czy stanowi ścianę labiryntu (a właściwie cztery ściany, gdyż sektor ten przylega ściankami do czterech sąsiednich sektorów).

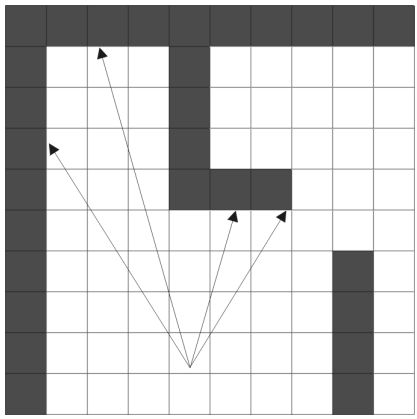
A oto przykład reprezentacji takiego świata w pamięci komputera:

```
int worldMap[][] = new int[][] {
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1},
        {1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1},
        {1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1},
        {1,0,0,0,1,1,1,0,0,0,0,0,0,1,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1,1,1,1,0,0,0,1},
        {1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,1,1,1,1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}};
```

Dzięki sprytnemu trikowi możemy na płaszczyźnie ekranu wygenerować trójwymiarowy obraz tego, co widzi gracz umieszczony wewnątrz labiryntu.

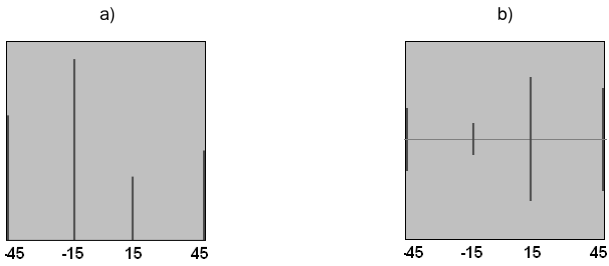
Dla uproszczenia przyjmijmy, że kąt rozwarcia stożka kamery, przez którą patrzymy, jest równy 90 stopni. Z punktu, w którym stoimy, prowadzimy 90 promieni (co 1 stopień), tak jak to jest pokazane na rysunku 3.2. (Dla czytelności na rysunku pokazano tylko kilka takich promieni).

Rysunek 3.2. *Promienie „rzucane" z punktu, w którym znajduje się obserwator*



Dla każdego z tych promieni szukamy punktu, w którym promień po raz pierwszy uderzy w ścianę labiryntu. Po znalezieniu każdego takiego punktu obliczamy jego odległość od punktu, w którym znajduje się obserwator. Odległość ta określa, jaką wysokość na ekranie monitora będzie mieć fragment ścianki, w którą uderzył dany promień. Im dalej nastąpiła kolizja ze ścianką labiryntu, tym mniejszy powinien być pionowy pasek przedstawiający ten fragment ścianki na wyświetlaczu telefonu. Wygenerowany na ekranie obraz będzie miał szerokość 90 pikseli (każda kolumna odpowiada jednemu z dziewięćdziesięciu promieni) i wysokość zależną od przyjętego przez nas współczynnika proporcjonalności pomiędzy wysokością ścianki a odległością od punktu kolizji z promieniem. Rysunek 3.3 przedstawia opisany sposób generacji obrazu na przykładzie kilku promieni z rysunku 3.2.

Rysunek 3.3.

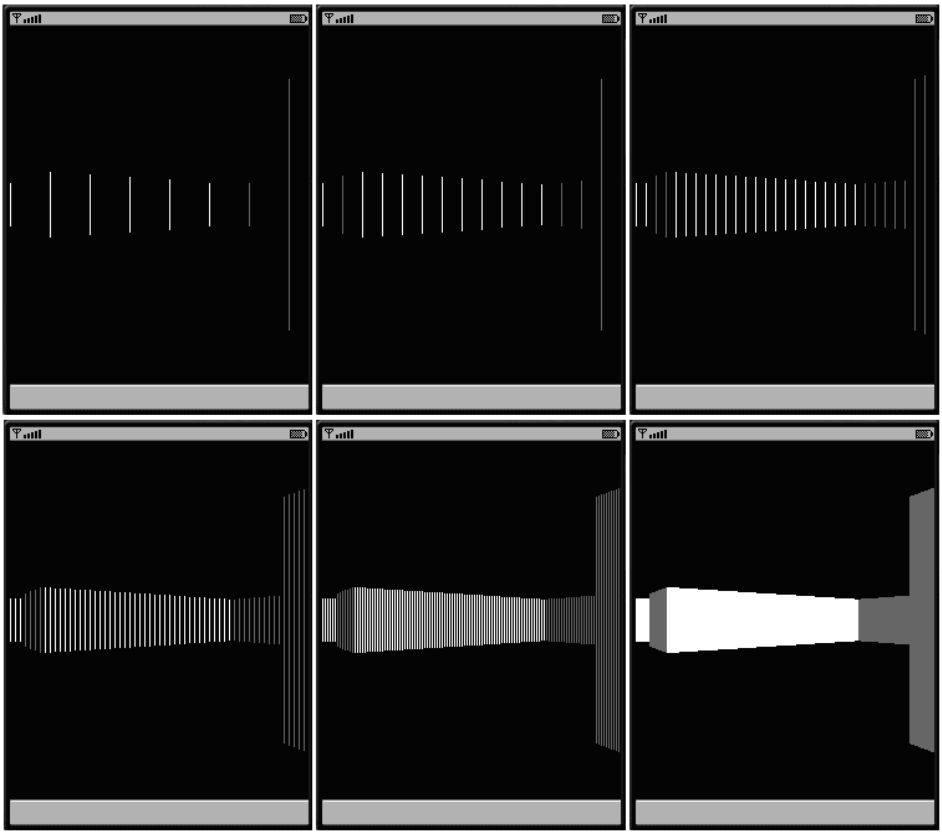


Na rysunku widzimy 4 przykładowe promienie. Pierwszy z nich biegnie pod kątem -45 stopni względem kierunku, w którym patrzy gracz. Kolejne trzy promienie puszczane są pod kątami odpowiednio: -45, -15, 15 i 45 stopni względem obserwatora. Promienie te uderzają w ściany labiryntu w czterech różnych punktach. Odległości, w jakich uderzyły one w ściany labiryntu, pokazane są na rysunku 3.3a w postaci wykresu słupkowego.

Im dalej nastąpiło uderzenie, tym wyższy będzie słupek odpowiadający promieniowi wypuszczonemu pod określonym kątem. Widzimy, że najwyższy jest słupek odpowiadający promieniowi wypuszczonemu pod kątem -15 stopni. Pokrywa się to z rzeczywistością (popatrzmy na rysunek 3.3a); promień wypuszczony pod kątem -15 stopni dotarł do najodleglejszego zaułka labiryntu. Słupki dla kątów 15 i 45 stopni są najniższe, gdyż, jak widzimy na rysunku 3.3a, promienie te natrafiły na przeszkodę najszybciej.

Teoria perspektywy mówi, że obiekty położone dalej są widziane jako mniejsze niż te, które znajdują się tuż przy obserwatorze. Musimy więc odpowiednio przeskalować nasz wykres słupkowy, a ponadto wyśrodkować paski względem linii horyzontu. Przedstawia to rysunek 3.3b. Paski najdłuższe stają się najkrótsze i podobnie — najkrótsze są odpowiednio dłuższe.

Przy odrobinie wyobraźni można dostrzec, że rysunek 3.3b stanowi fragment obrazu, który widzi gracz z rysunku 3.2. Wygenerowaliśmy dla przykładu tylko 4 promienie. Popatrzmy, co będzie się działo, gdy użyjemy 8, 16, 32, 64, 128 i wszystkich promieni. Dla ułatwienia percepcji użyjemy różnych kolorów dla ścianek labiryntu wzdłuż osi X i osi Y (rysunek 3.4).



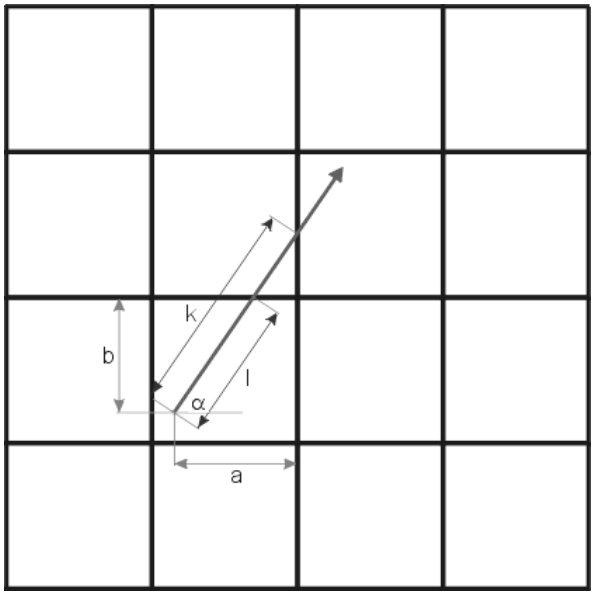
Rysunek 3.4. *Widok labiryntu przy 8, 16, 32, 64 i 128 promieniach rzuconych z punktu obserwacji*

Wszystkie obliczenia odbywają się w przestrzeni 2D i polegają głównie na znalezieniu punktu przecięcia się prostej z pierwszą napotkaną ścianką labiryntu. Ponieważ obliczenia takie musimy przeprowadzić tylko raz dla każdej kolumny ekranu, to przy ekranie o szerokości 90 pikseli musimy obliczenia wykonać jedynie 90 razy na klatkę animacji sceny.

Algorytm DDA śledzenia promienia

Na podstawie ogólnego opisu raycastingu widzimy, że najważniejszym elementem metody jest algorytm znajdujący te sektory mapy, przez które przebiegnie promień.

Algorytm, który zastosowaliśmy w naszym przykładzie, rozwiązuje problem w sposób iteracyjny. W każdym kroku iteracji analizowany jest jeden sektor mapy. Obliczenia rozpoczynają się w sektorze, w którym znajduje się obserwator — patrz rysunek 3.5.



Rysunek 3.5. *Szukanie drugiego sektora, który odwiedzi rzucany promień*

Danymi wejściowymi do obliczeń są:

- Pozycja obserwatora względem lewego dolnego rogu sektora
- Kierunek, w którym spogląda obserwator
- Kierunek (kąt α), pod jakim wyrusza promień z punktu obserwacji

Zadaniem algorytmu jest znalezienie następnego w kolei sektora, znajdującego się na drodze promienia. Aby to stwierdzić, algorytm bada, przez którą ściankę promień opuszcza bieżący sektor.

Rysunek 3.5 pokazuje, w jaki sposób możemy obliczyć dwie wielkości:

- Odległość *k* punktu, w którym promień przetnie pierwszy raz pionową linię wirtualnej siatki oddzielającej sektory
- Odległość *l* od punktu, w którym promień przetnie po raz pierwszy poziomą linię wspomnianej siatki

Te dwie wartości pozwalają nam stwierdzić, czy promień przebije ścianki w kierunkach poziomych (*W* lub *E*) czy pionowych (ścianki *N* lub *S*). Jeśli wartość *k* jest mniejsza od *l*, wnioskujemy, że promień opuszcza sektor przechodząc przez ściankę *W* lub *E*. W przeciwnym razie oznacza to, że następnym sektorem odwiedzonym przez promień będzie sąsiad ścianki *N* lub *S*.

Aby uściślić obliczenia, badana jest jeszcze wartość kąta α, pod którym biegnie promień. Poniższa tabelka przedstawia reguły wnioskowania algorytmu na podstawie wartości *k*, *l* oraz wartości kąta α.

Kierunek ścianki, przez którą promień opuszcza sektor	Reguła wnioskowania
W	*K<=l* oraz α*>90* i α*<270*
E	*K<=l* oraz α*<90* i α*>-90*
N	*k>l* oraz α*>0 i* α*<180*
S	*k>l* oraz α*>180*

Teraz czas na odrobinę matematyki. Wartości *k* i *l* wyrażają się wzorami:

- $k = a/\cos(\alpha)$
- $l = b/\sin(\alpha)$

Wartości *a* oraz *b* możemy obliczyć z zależności:

- $a = 1 - X_l$
- $b = 1 - Y_l$

Gdzie (x_l, y_l) to współrzędne obserwatora w lokalnym układzie współrzędnych sektora — jak znaleźć te współrzędne, pokażemy w dalszej części rozdziału.

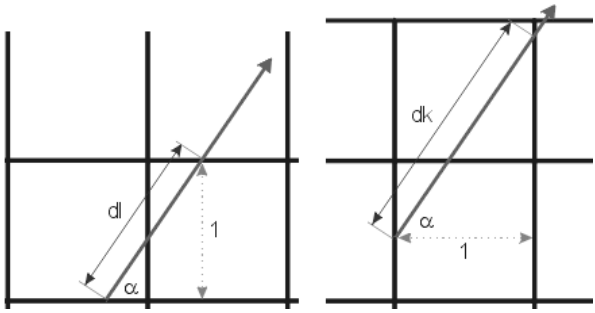
Kolejne iteracje

Wiemy już, jak wyznaczyć drugi z kolei sektor, który przetnie rzucany promień. A jak znaleźć kolejne sektory, aż do momentu, gdy promień natrafi na ścianę? Spójrzmy na rysunek 3.6.

Znając kąt, pod jakim wypuszczony został promień, możemy obliczyć odległości od kolejnych przecięć poziomych i pionowych linii wirtualnej siatki:

- $dk = 1/\cos(\alpha)$
- $dl = 1/\sin(\alpha)$

Informacje te wykorzystamy do znalezienia kolejnych sektorów na drodze promienia.



Rysunek 3.6. *Szukanie kolejnych sektorów odwiedzanych przez rzucany promień*

Na tym etapie znajdujemy się w drugim z kolei sektorze na trasie promienia. Zastanówmy się, jakie informacje posiadamy zaraz po wejściu do tego sektora. Po pierwsze, wiemy, przez którą ściankę promień do niego wkracza. Po drugie, wiemy, jaką drogę przebył promień do momentu przecięcia tej ścianki.

Gdybyśmy teraz znali odległość do kolejnego przecięcia się promienia z linią pionową i poziomą wirtualnej siatki, moglibyśmy wyznaczyć kolejny sektor na drodze promienia. Skorzystamy tutaj z wyprowadzonych wcześniej wzorów (*dk* oraz *dl*). Mamy do rozważenia dwa przypadki:

Jeśli do sektora promień wbiegł przecinając ściankę poziomą, to odległość od przecięcia ścianki pionowej znamy, gdyż wyliczyliśmy ją dla poprzedniego sektora. Odległość od następnego przecięcia z ścianką poziomą obliczymy, dodając wartość *dl* do odległości, w jakiej nastąpiło poprzednie przecięcie ścianki poziomej. Zmienne decyzyjne *l* oraz *k* przyjmą następujące wartości:

- $k = k$
- $l = l+dl$

Jeśli do sektora promień wbiegł przecinając ściankę pionową, to odległość od przecięcia ścianki poziomej również znamy, gdyż wyliczyliśmy ją dla poprzedniego sektora. Odległość od następnego przecięcia z ścianką pionową obliczymy, dodając wartość *dl* do odległości, w jakiej nastąpiło poprzednie przecięcie ścianki pionowej.

- $k = k+dk$
- $l = l$

Dalej postępujemy podobnie jak w przypadku sektora startowego:

- na podstawie dwóch odległości oraz kąta α wybieramy następny sektor na drodze promienia;
- gdy znamy już kolejny sektor, sprawdzamy, czy jest on pusty. Jeśli tak, wówczas przechodzimy do kolejnej iteracji. W przeciwnym razie rysujemy linię na ekranie, gdyż natrafiliśmy na przeszkodę.

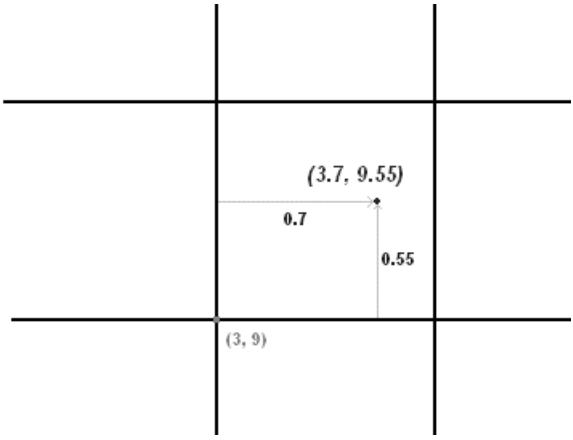
Współrzędne położenia obserwatora względem sektora

Każdy sektor jest kwadratem o długości boku równym 1. Znając współrzędne (X_g, Y_g) położenia obserwatora w układzie globalnym, możemy obliczyć:

- współrzędne sektora na wirtualnej siatce — poprzez odcięcie części ułamkowej współrzędnych globalnych,
- współrzędne lokalne (X_l, Y_l) względem tego sektora — poprzez odcięcie części całkowitej współrzędnych globalnych.

Rozważmy na przykład współrzędne globalne obserwatora o wartości (3.7, 9.55) — rysunek 3.7. Na ich podstawie obliczamy:

- współrzędne sektora na mapie — (3, 9)
- współrzędne lokalne — (0.7, 0.55)



Rysunek 3.7. *Obliczanie współrzędnych położenia względem sektora*

Implementacja — wersja uproszczona

W podrozdziale tym przytoczona jest uproszczona implementacja metody rzucania promieni. Sposób generowania obrazu przez metodę `Render()` klasy `Raycaster` (opisany w poprzednich akapitach) jest stosunkowo łatwy do zrozumienia, lecz zawiera pewne uproszczenia wprowadzone na potrzeby dydaktyczne. Na tym etapie najważniejsze jest zrozumienie samej idei raycastingu — na szczegóły przyjdzie pora później.

Klasa Raycaster

Klasa `Raycaster` zawiera główną część naszej aplikacji zajmującej się rysowaniem labiryntu. Oto jej kod:

```
package mypackage;

import javax.microedition.lcdui.*;
import java.util.*;
import java.lang.Math.*;

public class Raycaster extends Canvas {
```

```
public static final int screenHeight = 200;
public static final int screenWidth = 90;
public static final int screenHalfHeight = 50;
public static final int maxDistance = 40;
private static final int halfViewAngle = 45;

private Player player;

private  int worldMap[][] = new int[][] {
    {4,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
    {4,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,2}};

private int arrX[] = new int[] { 1,-1,-1, 1 };
private int arrY[] = new int[] { 1, 1,-1,-1 };

public Raycaster() {
    initialize();
}

private void initialize() {
    player = new Player(this);
    player.setPos(12.0f,12.0f);
}

public void paint(Graphics g)
{
    int playerDirection = player.getAngle()%360;
    if(playerDirection<0)
        playerDirection+=360;

    g.fillRect(0,0,screenWidth,screenHeight);

    Render(g,playerDirection, player.getPosX(), player.getPosY());
}

private double Cos(double angle) {
```

```
    return Math.cos(Math.toRadians(angle));
}
private double Sin(double angle) {
    return Math.sin(Math.toRadians(angle));
}

public void Render(Graphics g, int playerDirection,
                   double playerX, double playerY)
{
        int x=0;

        for(int rayAngle = playerDirection - halfViewAngle ;
                rayAngle < playerDirection + halfViewAngle ;
                rayAngle++)
        {
            int ceilPosX = (int)playerX;
            int ceilPosY = (int)playerY;

            double a = 1.0 - (playerX - ceilPosX);
            double b = 1.0 - (playerY - ceilPosY);

            double dk = Math.abs(1.0 / Cos(rayAngle));
            double dl = Math.abs(1.0 / Sin(rayAngle));

            double k = a * dk;
            double l = b * dl;

            int index = (rayAngle % 360) / 90;
            int stepX = arrX[index];
            int stepY = arrY[index];

            double distance = 0.f;
            boolean hit = false;
            int leftHit = 1;

            while(false == hit) {
                if(k>l) {
                    ceilPosY += stepY;
                    if(0 != worldMap[ceilPosX][ceilPosY]) {
                        hit = true;
                        distance = l;
                    } else {
                        l += dl;
                    }
                } else {
                    ceilPosX += stepX;
                    if(0 != worldMap[ceilPosX][ceilPosY]) {
                        hit = true;
                        distance = k;
                leftHit=2;
                    } else {
                        k += dk;
                    }
                }
            }

            g.setColor(160/leftHit,160/leftHit,255/leftHit);
```

```
                distance = maxDistance-distance%maxDistance;

                g.drawLine(x,screenHalfHeight - (int)(distance),
                     x,screenHalfHeight + (int)(distance));
                x++;
            }
        }

        public void keyPressed(int keyCode) {
            int action = getGameAction(keyCode);
            switch(action) {
                case Canvas.LEFT:
                    player.RotateLeft();
                    break;
                case Canvas.RIGHT:
                    player.RotateRight();
                    break;
                case Canvas.UP:
                    player.MoveForward();
                    break;
                case Canvas.DOWN:
                    player.MoveBackward();
                    break;
            }
            repaint();
            serviceRepaints();
        }
}
```

W klasie `Raycaster` na samym początku definiujemy kilka pożytecznych stałych wykorzystywanych w algorytmie rzucania promieni:

- `screenHeight` — wysokość ekranu wyrażona w pikselach,
- `screenWidth` — szerokość ekranu wyrażona w pikselach,
- `maxDistance` — maksymalna odległość, na jaką widzi obserwator,
- `halfViewAngle` — połowa kąta rozwarcia stożka kamer.

W dwuwymiarowej tablicy `worldMap` definiujemy wygląd naszego labiryntu. Wartości różne od zera oznaczają sektory stanowiące ściany labiryntu. Zera oznaczają przestrzenie otwarte labiryntu.

Tablice `arrX` i `arrY` służą do celów optymalizacyjnych procedury rozstrzygającej o tym, którą ścianką promień opuszcza bieżący sektor.

Metoda Render()

Omówiony wcześniej *algorytm DDA* realizowany jest w metodzie `Render()`. Parametrami wejściowymi do metody są:

- wektor kierunku, w którym spogląda gracz (0 – 360 stopni),
- współrzędne położenia gracza (współrzędne globalne),
- kontekst graficzny.

Pętla `for` obejmująca cały kod wewnątrz metody `render()` zapewnia, że wszystkie obliczenia powtarzane są dla każdego paska ekranu z osobna. W ten sposób realizowane jest rzucanie promienia dla każdego kąta z zakresu stożka widzenia kamery.

W pierwszym kroku znajdujemy współrzędne sektora, w którym znajduje się gracz. Przypomnijmy, że każdy sektor to kwadrat o długości boku równym 1 metr. Poprzez rzutowanie współrzędnych globalnych położenia gracza do wartości typu `int` pozbywamy się ich części ułamkowej. Wartości te (są to szukane współrzędne sektora) zapamiętujemy w zmiennych `ceilPosX` oraz `ceilPosY`.

Następnie obliczamy wartości następujących współczynników: *a*, *b*, *dk*, *dl*, *k*, *l*. Ich znaczenie oraz sposób obliczenia już omówiliśmy. Aby obliczyć współczynnik *a*, od jedynki odejmujemy wartość współrzędnej lokalnej gracza w sektorze (wzdłuż osi *x* układu współrzędnych — patrz rysunek 3.7). Podobnie postępujemy obliczając współczynnik *b*. Zestaw tych sześciu współczynników posłuży dalej do obliczania odległości od kolejnych punktów przecięcia się promienia z poziomymi i pionowymi liniami wirtualnej siatki wyznaczającej sektory.

W zależności od wartości kąta, pod jakim biegnie rzucany promień, inicjalizowane są dwie zmienne decyzyjne:

- `stepX` przyjmuje wartość dodatnią (promień biegnie w prawą stronę), dla wartości kąta z zakresu od -90 do 90 stopni. Dla pozostałych kątów zmienna przyjmuje wartość ujemną — promień biegnie w lewą stronę,
- `stepY` przyjmuje wartość dodatnią (promień biegnie w górę) dla wartości kąta z zakresu od 0 do 180 stopni. Dla pozostałych kątów zmienna przyjmuje wartość ujemną — promień biegnie w dół.

Najważniejsza część algorytmu `DDA` zawarta jest w pętli `while`. Oto pseudokod pętli `while`:

```
Dopóki(nie nastąpiła kolizja ze ścianką labiryntu)
{
  Jeśli(odległość od przecięcia z najbliższą pionową granicą kratki >
        Odległości od przecięcia z najbliższą poziomą granicą kratki)
  {
    Analizuj kratkę powyżej lub poniżej bieżącej

    Jeśli(nowa kratka jest ściana)
    {
      nastąpiła kolizja - zapamiętaj odległość
    }

    Ustal odległość od punktu kolizji z kolejną poziomą granicą kratek
  }
  w przeciwnym wypadku
  {
    Analizuj kratkę po prawej lub lewej stronie bieżącej

    Jeśli(nowa kratka jest ściana)
    {
      nastąpiła kolizja - zapamiętaj odległość
    }
```

```
        Ustal odległość od punktu kolizji z kolejną pionową granicą kratek
      }
    }
```

Kod w pętli powtarzany jest aż do momentu, gdy stwierdzona zostanie kolizja promienia ze ścianką labiryntu. Gdy stwierdzona zostanie kolizja, w zmiennej `distance` zapamiętywana jest odległość od punktu kolizji. W zmiennej `leftHit` zapamiętywana jest informacja pozwalająca zastosować proste cieniowanie ścianek labiryntu. Ścianki równoległe do osi *Y* układu współrzędnych będą jaśniejsze od tych równoległych do osi *X*.

Po wykryciu kolizji promienia ze ścianką wykonanie pętli `While` jest przerywane. Następnie obliczana jest długość pionowego paska na ekranie odpowiadającego promieniowi i obliczonej odległości od punktu kolizji. Pasek ten ostatecznie jest rysowany na ekranie przy użyciu metody `drawLine`.

Klasa Player

Obiekt tej klasy reprezentuje gracza poruszającego się po labiryncie. W prawdziwej grze klasa ta zawierałaby między innymi takie informacje, jak: ilość „żywotów", zdobyte punkty, ilość energii i wiele innych zależnych od typu gry. W naszym przykładzie gracz opisany jest najprostszymi parametrami. Są to:

- pozycja na mapie,
- orientacja (kąt obrotu),
- parametry ruchu — prędkość poruszania i prędkość rotacji.

Klasa `Player` udostępnia także metody służące do manipulacji tymi parametrami. Metody `moveForward` oraz `moveBackward` pozwalają poruszać się odpowiednio do przodu i w tył. Metody te wykrywają też ewentualne przeszkody (takie jak ściany labiryntu), stojące na drodze gracza, i odpowiednio modyfikują wektor ruchu. Funkcje `rotateRight` oraz `rotateLeft` pozwalają graczowi obracać się wokół własnej osi pionowej.

```
package hello;

public class Player
{
    private Raycaster r;

    private float posX;
    private float posY;
    private int angle = 90;

    /** Konstrukcja obiektu gracza */
    public Player(Raycaster _r) {
        r = _r;
    }

    private int getSpeed() {
        return 1;
    }
```

```
    public void setPos(float x,float y) {
        posX = x;
        posY = y;
    }

    public float getPosX()    { return posX; }
    public float getPosY()    { return posY; }

    public int getAngle() { return angle; }
    public void setAngle(int _angle) { angle = _angle; }

    public void MoveForward()
    {
        float newPosX = posX + 1.0f *
                        (float)Math.cos(Math.toRadians((double)angle));
        float newPosY = posY + 1.0f *
                        (float)Math.sin(Math.toRadians((double)angle));

        if(r.getMapPoint( (int)newPosX,(int)posY) == 0)
            posX = newPosX;
        if(r.getMapPoint( (int)posX,(int)newPosY) == 0)
            posY = newPosY;
    }

    public void MoveBackward()
    {
        float newPosX = posX - 1.0f *
                        (float)Math.cos(Math.toRadians((double)angle));
        float newPosY = posY - 1.0f *
                        (float)Math.sin(Math.toRadians((double)angle));

        if(r.getMapPoint( (int)newPosX,(int)posY) == 0)
            posX = newPosX;
        if(r.getMapPoint( (int)posX,(int)newPosY) == 0)
            posY = newPosY;
    }

    public void RotateLeft() {
        angle-=5;
    }

    public void RotateRight() {
        angle+=5;
    }
}
```

Klasa midletu

Ponieważ nasz przykład nie używa żadnych (!) zasobów (grafik, dźwięku itp.), kod klasy midletu wygląda następująco:

```
package mypackage;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

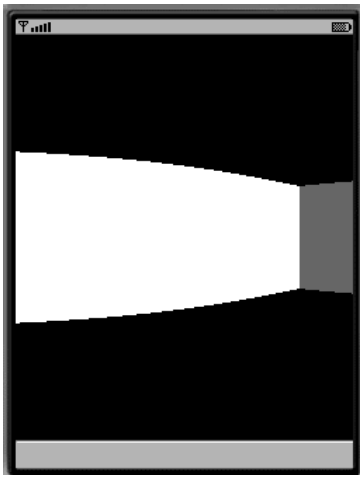
```
public class Welcome extends MIDlet {

    public Welcome() {}

    public void startApp() {
        Display.getDisplay(this).setCurrent(new Raycaster());
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

}
```

Efekt „rybiego oka”

Niestety, w naszym przykładzie napotykamy pewien problem. Do obliczenie wysokości paska na ekranie używamy wprost odległości od punktu, w którym promień uderzył w ściankę. Wynikiem takiego postępowania jest efekt „rybiego oka”.

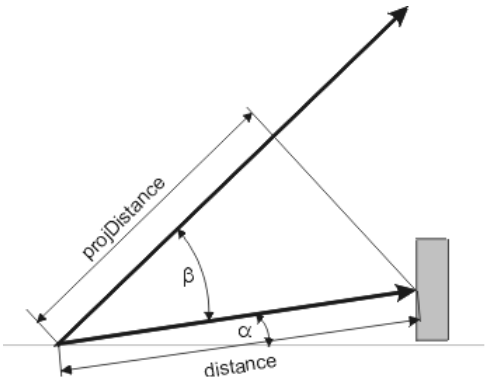
Wyobraźmy sobie sytuację, jaką przedstawia rysunek 3.8. Obserwator znajduje się dokładnie na wprost długiej ściany. Pomimo to obraz jest zdeformowany, co widać na rysunku. Aby efekt ten zniwelować, powinniśmy zrzutować wektor długości na wektor kierunku, w którym patrzy obserwator. Długość wynikowego wektora powinniśmy zastosować w rysowaniu paska dla tego promienia.

Rysunek 3.8.
Efekt rybiego oka



Odległość tę uzyskamy także, mnożąc odległość euklidesową przez kosinus kąta pomiędzy promieniem a wektorem kierunku obserwatora.

Kąta tego nie znamy, ale możemy go łatwo obliczyć. Na podstawie rysunku 3.9 widzimy, że odejmując od kąta β kąt α, uzyskamy szukany kąt pomiędzy promieniem a kierunkiem, w którym patrzy gracz. Kąt β to kąt pomiędzy rzucanym promieniem a osią



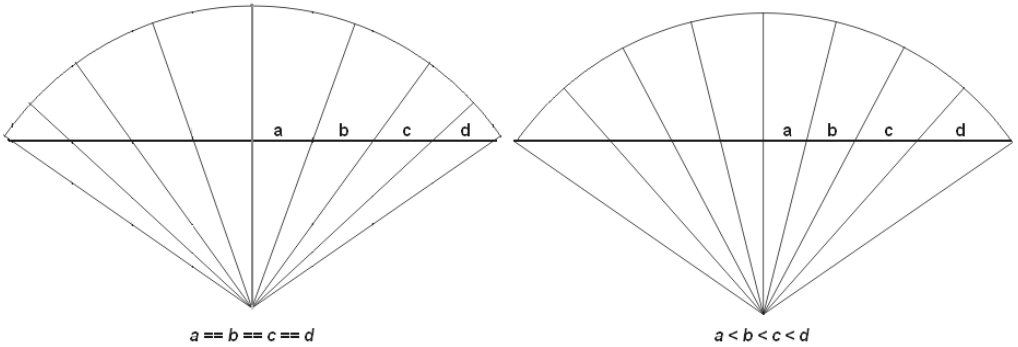
Rysunek 3.9. *Rzutowanie wektora odległości na kierunek obserwacji*

x układu współrzędnych. Kąt α to kąt pomiędzy kierunkiem, w którym patrzy gracz, a osią x układu. W ten sposób uzyskujemy następujący wzór na odległość punktu kolizji ze ścianą labiryntu od płaszczyzny ekranu:

$$projDistance = \cos(\beta - \alpha) * distance$$

Jeszcze jeden problem

Spójrzmy na rysunek 3.10. Widzimy na nim pęk promieni wychodzących z punktu *P*. Na granicznych promieniach (o kątach -45 oraz 45 stopni względem kierunku, w którym patrzymy) rozpięliśmy płaszczyznę ekranu. To na niej wygenerowany zostanie obraz naszej sceny.



Rysunek 3.10. *Błędny oraz poprawny podział płaszczyzny ekranu przez rzucane promienie*

Z lekcji matematyki wiemy, że promienie dzielą tę prostą na odcinki o różnej długości. W efekcie obraz, który uzyskamy, może być zdeformowany. Błędy będą widoczne głównie przy dużych zbliżeniach na krawędziach ekranu. Aby tego uniknąć, musimy promienie „rzucać" w taki sposób (pod takimi kątami), aby płaszczyznę ekranu (prostą na rysunku) dzieliły na równej długości odcinki.

Metoda dokładna

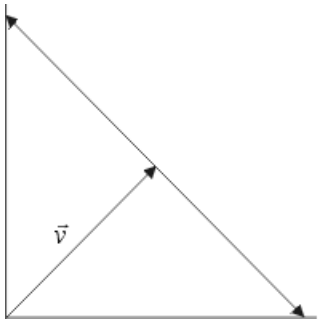
W dalszej części tego rozdziału zajmiemy się rozwiązaniem problemu prawidłowego podziału rzutni (ekranu) przez rzucane promienie.

W metodzie dokładnej obliczamy współrzędne wektora kierunku promienia, który biegnie przez określony pasek powierzchni ekranu. Zmiana ta pociąga za sobą modyfikacje obliczeń pozostałych wielkości używanych przez algorytm uproszczony.

Obliczamy współrzędne wektora promienia

Na końcu jednostkowego wektora kierunku obserwatora (rysunek 3.11) zaczepiamy dwa wektory jednostkowe o przeciwnych znakach, prostopadłe do wektora kierunku. W ten sposób tworzymy stożek kamery, przez którą obserwowana będzie scena.

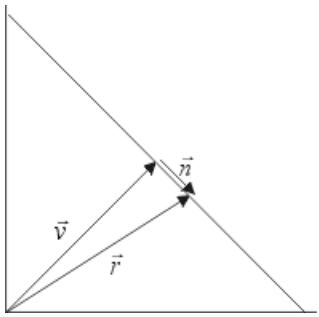
Rysunek 3.11. *Tworzenie stożka kamery na podstawie wektora kierunku obserwacji*



Prostą utworzoną przez dwa wektory prostopadłe do kierunku, w którym zwrócony jest gracz, dzielimy na N równych odcinków. Wartość N określa poziomą rozdzielczość ekranu, a tym samym liczbę promieni, które będziemy śledzić.

Obliczamy wektor prostopadły do wektora kierunku obserwacji. Jego współrzędne wyrażone są wzorem:

Rysunek 3.12. *Podział płaszczyzny ekranu oraz znajdowanie wektora kierunku rzucanego promienia*



$$Vperp = [Vy, -Vx]$$

W powyższym wzorze Vx i Vy to współrzędne wektora kierunku, w którym patrzy gracz.

Mnożąc ten wektor przez odpowiednią wielkość skalarną z zakresu [–1,1] uzyskamy wektor *n*, jak na powyższym rysunku. Wektor ten wyznacza punkt płaszczyzny ekranu, przez który przebiegać będzie rzucany promień. Wektor kierunku promienia obliczamy, dodając do siebie wektor *n* i prostopadły do niego wektor kierunku obserwacji *v*:

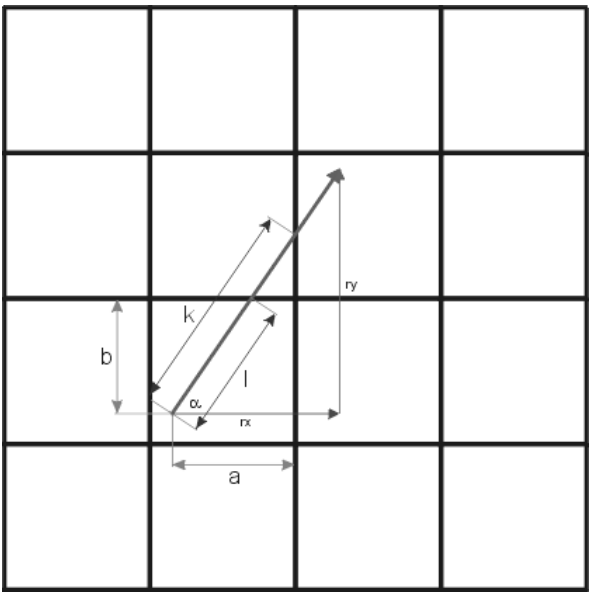
$$\vec{r}(r_x, r_y) = \vec{v} + \vec{n}$$

Algorytm DDA w metodzie dokładnych podziałów

Zmienne decyzyjne wyliczamy na podstawie znaku współrzędnych r_x i r_y:

- `stepX` przyjmuje wartość dodatnią (promień biegnie w prawą stronę), jeśli składowa r_x ma wartość dodatnią; w przeciwnym przypadku zmienna decyzyjna `stepX` przyjmuje wartość ujemną — promień biegnie w lewą stronę;
- `stepY` przyjmuje wartość dodatnią (promień biegnie w górę), jeśli składowa r_y ma wartość dodatnią; w przeciwnym przypadku zmienna decyzyjna `stepY` przyjmuje wartość ujemną — promień biegnie w dół.

Sposób obliczenia wartości współczynników *k*, *l* pokazany jest na rysunku 3.13.



Rysunek 3.13. *Obliczanie współczynników k, l w metodzie „dokładnej”*

Wartości tych współczynników wyrażone są wzorami:

$$\cos\alpha = \frac{r_x}{|\vec{r}|} = \frac{a}{K} \Rightarrow K = a\frac{|\vec{r}|}{r_x}$$

$$\sin\alpha = \frac{r_y}{|\vec{r}|} = \frac{b}{L} \Rightarrow L = b\frac{|\vec{r}|}{r_y}$$

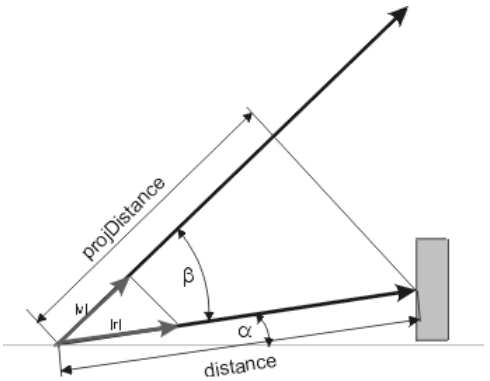
Modyfikacji ulega także sposób obliczenia rzutu odległości na wektor kierunku, w którym spogląda gracz. Kosinus kąta β potrzebny do znalezienia rzutu możemy obliczyć w dwojaki sposób:

- Po pierwsze, wyraża się on ilorazem dwóch wielkości: modułu rzutu wektora odległości na kierunek obserwacji oraz modułu wektora odległości euklidesowej,
- Po drugie, jest on równy ilorazowi długości wektora jednostkowego *v* i wektora *r* wyznaczającego promień.

Zależności te opisać możemy następującym wzorem:

$$\cos(\beta - \alpha) = \frac{dist_{\perp}}{dist} = \frac{|\hat{v}|}{|\vec{r}|}$$

Znaczenie poszczególnych składowych wzoru pokazane są na rysunku 3.14.



Rysunek 3.14. *Obliczanie rzutu odległości euklidesowej w metodzie dokładnej*

Przekształcając ten wzór, otrzymujemy zależność wartości rzutu wektora odległości od kierunku obserwacji:

$$dist_{\perp} = \frac{|\hat{v}|}{|\vec{r}|} dist = \frac{dist}{|\vec{r}|}$$

Poprawiona implementacja metody Render

Poniższa implementacja odzwierciedla wspomniane wcześniej poprawki:

```
public void Render(Graphics g, int playerDirection,
                   double playerX, double playerY)
{
    double playerDirX = Cos(playerDirection);
    double playerDirY = Sin(playerDirection);

    double playerPerpDirX = -playerDirY;
    double playerPerpDirY =  playerDirX;
```

```
double deltaOffset=2.0/screenWidth;
int x=0;

for(double offset = -1 ; offset < 1 ; offset += deltaOffset)
{
   int ceilPosX = (int)playerX;
   int ceilPosY = (int)playerY;
   double a = 1.0 - (playerX - ceilPosX);
   double b = 1.0 - (playerY - ceilPosY);

   double rx = playerDirX + playerPerpDirX * offset;
   double ry = playerDirY + playerPerpDirY * offset;

   double stepX,stepY;
   stepX = stepY = -1;

   if(rx>0) stepX = 1;
   if(ry>0) stepY = 1;

   double lengthR = Math.sqrt(rx*rx + ry*ry);

   double dl = Math.abs(lengthR /ry);
   double dk = Math.abs(lengthR /rx);

   double k = a * dk;
   double l = b * dl;

   boolean hit = false;     // nastąpiła kolizja ze ścianą?
   int leftSideHit = 1;     // czy to ściana płn. czy płd.?


   double distance = 0.f;

   while(false == hit) {
       if(k>l) {
            ceilPosY += stepY;
            if(0 != worldMap[ceilPosX][ceilPosY]) {
                  hit = true;
                  distance = l;
                  leftSideHit = 1;
            } else {
                  l += dl;
            }
        } else {
            ceilPosX += stepX;
            if(0 != worldMap[ceilPosX][ceilPosY]) {
                  hit = true;
                  distance = k;
                  leftSideHit = 2; //2 dodaje cieniowanie
            } else {
                  k += dk;
            }
        }
     }
     g.setColor(160/ leftSideHit,160/ leftSideHit,255/ leftSideHit);

     distance = distance / lengthR;
```

```
            distance = 30/distance;

            g.drawLine(x,screenHalfHeight - (int)distance,
             x,screenHalfHeight + (int)distance);
            x++;
        }
    }
```

Najpierw na podstawie kąta rotacji obliczamy jednostkowy wektor *v* orientacji gracza. Następnie na jego podstawie obliczamy współrzędne wektora prostopadłego do wektora kierunku. Współrzędne tych wektorów reprezentowane są przez pary zmiennych (`playerDirX`, `playerDirY`) i (`playerPerpDirX`, `playerPerpDirY`).

Następnie obliczamy długość wektora *n*, prostopadłego do wektora kierunku i odmierzającego na płaszczyźnie kolejne punkty, przez które przechodzić powinny rzucane promienie. Ponieważ płaszczyznę ekranu rozpostarliśmy na dwóch jednostkowych wektorach, długość wektora *n* obliczamy, dzieląc wartość 2 przez szerokość (w pikselach) generowanego obrazu. Zmienna wektora *n* zapamiętana zostaje w zmiennej `deltaOffset`.

Podobnie jak w pierwszym przykładzie główna praca wykonywana jest w pętli `for`. Tutaj, po obliczeniu wartości współczynników *a* oraz *b*, obliczamy współrzędne wektora ***r*** wyznaczającego rzucany aktualnie promień.

Wektor *r* jest wynikiem sumy dwóch wektorów:

- Wektora kierunku, w którym zwrócony jest gracz,
- Prostopadłego do niego wektora, o długości zależnej od piksela ekranu (linii), dla której jest on rzucany.

W kolejnym kroku znajdujemy wartość zmiennych decyzyjnych `stepX` i `stepY`. Tym razem obliczamy je na podstawie wartości współrzędnych wektora *r*.

Ostatnim krokiem przed rozpoczęciem śledzenia promienia jest obliczenie współczynników *k*, *l*, *dk* i *dl*. Sposób, w jaki jest to wykonywane, obrazuje rysunek 3.13.

Dalsza część kodu (właściwe śledzenie promienia) jest prawie identyczna z tą w poprzednim przykładzie i nie będziemy jej opisywać tutaj ponownie. Jedyna różnica to sposób obliczenia odległości od punktu kolizji. Korzystamy tutaj z rzutu odległości euklidesowej na wektor orientacji gracza. Sposób jego obliczenia już omawialiśmy, ponadto zobrazowany jest on także na rysunku 3.14.

Podsumowanie

Technika raycastingu, którą poznaliśmy w tym rozdziale, okazała się krokiem milowym w świecie gier komputerowych. Jest potwierdzeniem znanej tezy, że geniusz tkwi w prostocie. Wykorzystując tak prostą funkcjonalność jak rysowanie linii oraz proste obliczenia na płaszczyźnie, stworzyliśmy wrażenie trójwymiarowości świata opisanego przez dwuwymiarową tablicę znaków.

Technika ta, po raz pierwszy zastosowana przez Johna Carmacka w grze *Wolfenstein*, legła u podstaw takich gier jak *Doom* i przyczyniła się do błyskawicznego rozwoju gier 3D. Bez raycastingu gry komputerowe nie byłyby w miejscu, w którym znalazły się obecnie. Każdy programista gier komputerowych powinien więc znać tę technikę, bo to znaczący fragment historii gier komputerowych.

W rozdziale zaprezentowaliśmy jedynie podstawy techniki raycastingu. Zachęcamy Czytelnika do dalszego zgłębiania tajników tej metody. Przez dodanie tekstur, podłogi, sufitu i przeciwników możemy tworzyć przepiękne gry 3D, które w dodatku nie będą wymagać najnowszych technologicznie telefonów.

Pytania kontrolne

1. Czemu generowanie obrazu 3D metodą raycastingu zawdzięcza tak dużą efektywność?
2. Opisz zasadę działania algorytmu *DDA*.
3. Wyprowadź wzór na odległości od pierwszego przecięcia promienia z pionową i poziomą granicą sektora.
4. Jak oblicza się współrzędne gracza w sektorze na podstawie położenia globalnego?
5. Co to jest i dlaczego powstaje efekt „rybiego oka” w metodzie raycastingu?

Zadania

1. Zmodyfikuj przykład tak, aby ściany labiryntu mogły mieć dowolny kolor.
2. Wprowadź kolor podłogi oraz dodaj niebo (tło) nad labiryntem.
3. W rozdziale 5. omówiona została metoda billboardingu wyświetlania płaskich obrazów w trójwymiarowym świecie. Zastanów się, w jaki sposób metodologia ta mogłaby być wykorzystana do reprezentacji animowanych postaci w świecie generowanym metodą raycastingu.