Politechnika Łódzka

Instytut Informatyki

Instrukcja do laboratorium

Fotorealistyczna Grafika Komputerowa

Laboratorium I

Punkt, wektor, promień, prymityw, przecięcie.

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

dr inż. Piotr Napieralski
mgr inż. Krzysztof Guzek

Łódź, 28.02.2012

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: napieral@ics.p.lodz.pl

1. Wstęp

Dzisiejsze metody renderingu opierają się na rzeczywistych i fizycznych właściwościach światła. Ich twórcy starają się w jak najbardziej efektywny i przede wszystkim efektowny sposób wprowadzić do przestrzeni wirtualnej elementy fizyczne mające wpływ na ogólny realizm tworzonego komputerowo świata Symulacja zjawisk i fizycznych właściwości światła w syntetycznym modelu świata nosi nazwę global illumination (oświetlenia globalnego) nazywana jest ona w dalszej części GI. Metoda ta pozwala na znalezienie natężenia światła w każdym punkcie przestrzeni. W przypadku opisu sceny geometria obiektów i materiały, z których są zbudowane są równie ważne jak opis źródeł światła. Algorytm GI opisuje moment opuszczenia źródła przez strumień światła i późniejszą jego interakcję z innymi obiektami sceny. Algorytm ten pozwala na obliczenie pośredniego oświetlenia obiektów. Z takim właśnie oświetleniem mamy do czynienia w świecie rzeczywistym. Nieuwzględnienie tego typu oświetlenia przy tworzeniu sztucznego świata prowadzi do powstania obrazów mało realnych nie uwzględniających interakcji światła oraz odbić od wszystkich obiektów w scenie. Algorytmy GI oparte są na dwóch głównych technikach:

- Technice próbkowania punktowego (oparte na ray tracing)
- Technice elementów skończonych (oparte na radiosity)

2. Opis wirtualnego świata

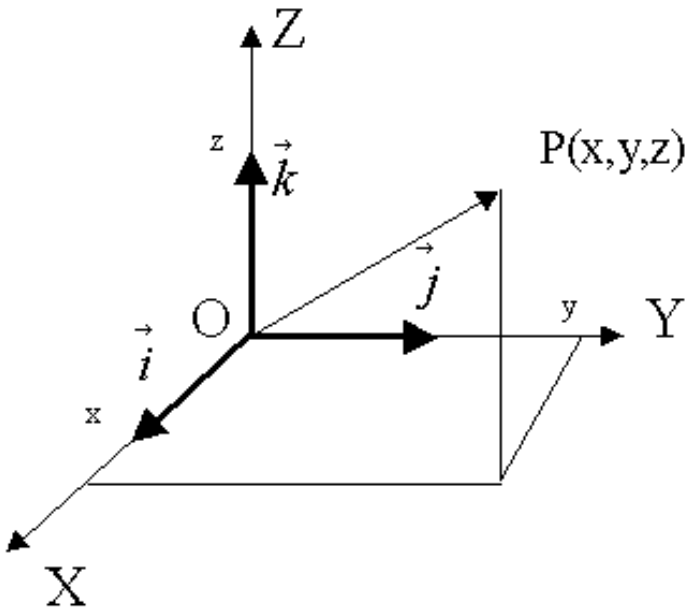
„Kto chce jednym spojrzeniem ująć wiele równocześnie przedmiotów, ten żadnego z nich nie widzi wyraźnie[1].”

Nim zaczniemy definiować obiekty, światła i kamery, należy stworzyć świat, a właściwie reguły, na podstawie których będą rozmieszczani mieszkańcy naszego mikrokosmosu. Wszystko zaczyna się od punktu, to dla niego stworzony może zostać układ współrzędnych który będzie przypisywać mu skończony ciąg (krotkę) liczb rzeczywistych zwanych współrzędnymi punktu. Określenie wymiaru zależeć będzie od liczby współrzędnych niezbędnych do określenia położenia punktu. W matematyce rozważa się przestrzenie o większej liczbie wymiarów niż trzy. Definiując wirtualny świat możemy ograniczyć się do trzech wymiarów, należy tu jednak wspomnieć że istnieją próby wykorzystania czasoprzestrzeni do opisu wirtualnej sceny[5].

[1] KARTEZJUSZ

2.1 Punkt i wektor

Najpopularniejszym układem współrzędnych i najczęściej stosowanym do opisu przestrzeni jest kartezjański układ współrzędnych. Do opisu punktu w przestrzeni trójwymiarowej używamy trzech współrzędnych względem prostopadłych osi x,y,z. W zależności od wzajemnych orientacji osi układ może być lewostronny lub prawostronny. Istnieje prosta metoda rozstrzygania, jakiego typu jest dany układ. Wystarczy wyobrazić sobie, ze patrzymy na układ "od strony" płaszczyzny X-Y (czyli płaszczyzny rozpiętej na osiach X, Y). Wtedy w obrębie naszego pola widzenia oś X leży poziomo i jest skierowana w prawo, a os Y pionowo do góry. Jeżeli oś Z jest skierowana do nas, to układ jest prawostronny. W przeciwnym razie układ jest lewostronny. Punkt przecięcia osi X, Y, Z nazywamy środkiem lub początkiem układu współrzędnych. Ten punkt ma współrzędne x=0, y=0, z=0, co w skrócie zapisujemy (0,0,0).



Rys.1 Trójwymiarowy układ współrzędnych kartezjańskich

Drugim podstawowym pojęciem jest wektor [6,7]. Wektor opisuje n-wymiarowe przesunięcie lub położenie. Przy definiowaniu wektora dla potrzeb fotorealizmu, używa się wektorów dwu i trzywymiarowych. Zamiast ogólnej klasy opisującej wektor n-wymiarowy, można stworzyć dwie klasy vector2 oraz vector3 dla zwiększenia efektywności programu. Nie ma potrzeby definiowania osobnej klasy dla punktu, można użyć klasę vector do definiowania punktów. Wartość wektora jest przesunięciem od początku położenia punktu. Definicja wektora w kartezjańskim układzie współrzędnych:

$$\mathbf{v} = (v_x, v_y, v_z)$$

Poniżej znajduje się przykładowa definicja klasy w języku C#

Program 1 Definicja klasy wektor

```
public struct Vector
{
    private float x;
    public float X
    {
          get { return x; }
          set { x = value; }
    }
    private float y;
    public float Y
    {
          get { return y; }
          set { y = value; }
    }
    private float z;
    public float Z
    {
          get { return z; }
          set { z = value; }
    }
(…)
```

Długość wektora a =(a_x,a_y,a_z) można otrzymać za pomocą równania:

$$\|a\| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

Powinna być zatem zdefiniowana metoda zwracająca $\|a\|$ oraz $\|a\|^2$. Jeśli długość v wynosi jeden, mamy do czynienia z *wektorem jednostkowym*. Należy zdefiniować odpowiednie działania arytmetyczne na wektorach (odpowiednie operatory lub funkcje) Podobnie jak dla kolorów zdefiniujemy dla wektora a,b oraz skalara s następujące operacje:

+a = (a_x,a_y,a_z),
-a = (-a_x,-a_y,-a_z),
k*a = (ka_x,ka_y,ka_z),
a*k = (ka_x,ka_y,ka_z),
a/k = (a_x/k, a_y/k, a_z/k),
a+b = (a_x+ b_x, a_y+ b_y, a_z+ b_z),
a-b = (a_x - b_x, a_y - b_y, a_z - b_z),
a/b = (a_x / b_x, a_y / b_y, a_z / b_z),
a*b = ($a_x b_x$, $a_y b_y$, $a_z b_z$),
a += b = (a←a+b)
a - = b = (a←a-b)
a *= b = (a←a*b)
a /= b = (a←a/b)

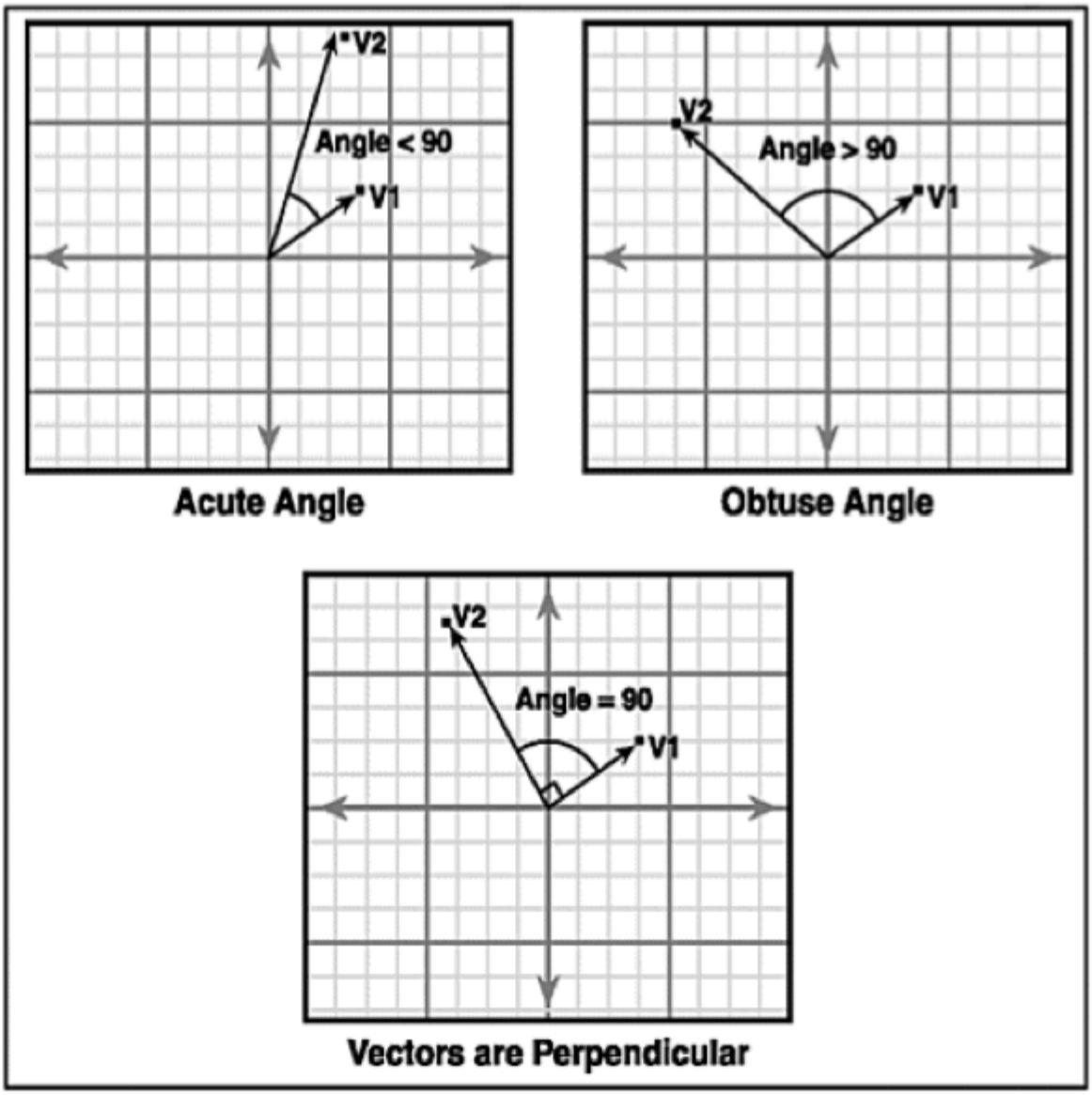
Dodatkowo dla wektorów, należy zdefiniować dwa bardzo istotne działania iloczyn wektorowy (×) i skalarny (·) :

$a \times b = (a_y b_z - a_z b_y,\ a_z b_x - a_x b_z,\ a_x b_y - a_y b_x)$,
$a \cdot b = a_x b_x + a_y b_y + a_z b_z$.

Zauważ że wynikiem iloczynu skalarnego jest skalar a wektorowego trójwymiarowy wektor. Iloczyn skalarny będzie używany wielokrotnie w przyszłości jako bardzo istotna operacja w grafice komputerowej, dzięki następującej własności:

$$a \times b = \|a\| \|b\| \cos\theta,$$

gdzie θ jest katem pomiędzy a i b. Najczęściej używać będziemy iloczyn skalarny w celu obliczenia cosinusa kąta pomiędzy dwoma wektorami. Jeśli a i b są wektorami jednostkowymi działanie sprowadzi się do trzech mnożeń i dwóch dodawań.



Rys.2 Wykorzystanie iloczynu skalarnego

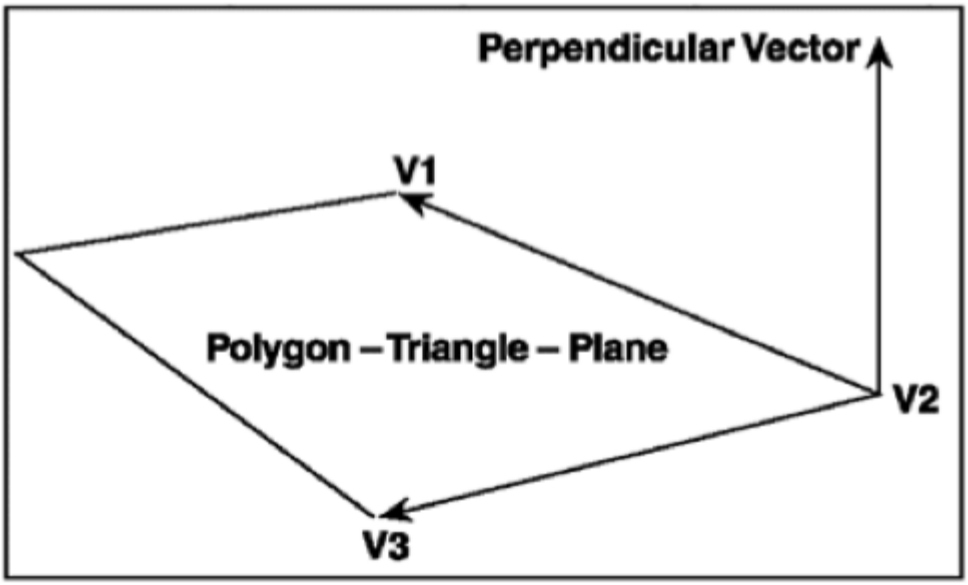
Iloczyn wektorowy ma inne lecz równie istotne właściwości geometryczne. Pierwsza to fakt że długość takiego wektora jest zależna od sinusa θ :

$$\|a \times b\| = \|a\|\|b\| \sin\theta \ .$$

Dodatkowo $a \times b$ jest prostopadły, zarówno do wektora a jak i b. Należy tu zwrócić uwagę, że istnieją tylko dwie możliwości kierunku takiego wektora. Rozpatrzmy przykład, gdzie wektory są w kierunku x-, y- oraz z- zdefiniowany następująco:

x = (1,0,0),
y = (0,1,0),
z = (0,0,1),

wtedy $x \times y$ musi być na plusie lub minusie kierunku z.

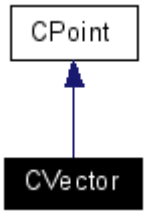


Rys.3 Wykorzystanie iloczynu wektorowego

Można to wywnioskować z faktu że $x \times y = z$. W jaki sposób można stwierdzić reprezentację geometryczną tych wektorów? W odpowiedzi na postawione pytanie używa się konwencji „zasady prawej dłoni". Zauważmy że podana zasada przyczynia się do powstania następujących własności:

$$x \times y = +z,$$
$$y \times x = -z,$$
$$y \times z = +x,$$
$$z \times y = -x,$$
$$z \times x = +y,$$
$$x \times z = -y$$

Zdefiniowanie wszystkich wspomnianych operacji powinno odbyć się w metodach klasy wektor.



Rys.4 Diagram klasy wektor

Program 2 Przykładowe Metody klasy Vector w C#

```
  public Vector(float x, float y, float z)
  {
      this.x = x;
      this.y = y;
      this.z = z;
  }
  public Vector(Vector p1, Vector p2)
  {
      this.x = p2.X - p1.X;
      this.y = p2.Y - p1.Y;
      this.z = p2.Z - p1.Z;
  }
  public Vector(Vector v)
  {
      this.x = v.X;
      this.y = v.Y;
      this.z = v.Z;
  }
  public override string ToString()
  {
  return "Vector(" + x.ToString() + "," + y.ToString() + "," + z.ToString() +
")";
  }
  public void normalize()
  {
      float n=this.length();
      if(n!=0)
      {
            this.div(n);
      }
      //else
  }
  public Vector normalizeProduct()
  {
      Vector newV=new Vector(this.x,this.y,this.z);
      float n = this.length();
      if(n!=0)
      {
      newV.div(n);
      return newV;
      }
  else
  return newV;// throw new Exception("Couldn't normalize");
  }
  public float length()
  {
```

```
  return (float)Math.Sqrt(Math.Pow(this.x, 2)+ Math.Pow(this.y, 2)+ Math.Pow
(this.z, 2));
  }
  public float lengthSquared()
  {
  return (float)(Math.Pow(this.x, 2) + Math.Pow(this.y, 2) + Math.Pow(this.z,
2));
  }
  public float dot(Vector v)
  {
  return (this.x * v.x + this.y * v.y + this.z * v.z);
  }

  public Vector cross(Vector v)
  {
  return new Vector(this.y * v.z - this.z * v.y, this.z * v.x - this.x * v.z,
this.x * v.y - this.y * v.x);
  }
  public void negate()
  {
     this.x = -this.x;
     this.y = -this.y;
     this.z = -this.z;
  }

  public void add(Vector v)
  {
     this.x += v.X;
     this.y += v.Y;
     this.z += v.Z;
  }

  public void sub(Vector v)
  {
     this.x -= v.X;
     this.y -= v.Y;
     this.z -= v.Z;
  }

  public void div(float f)
  {
     if (f != 0)
     {
           this.x /= f;
           this.y /= f;
           this.z /= f;
     }
     else
  throw new Exception("Cant divide by 0");
  }

  public void mag(float f)
  {
     this.x *= f;
     this.y *= f;
     this.z *= f;
  }
  #region Operators
  public static Vector operator *(float scalar, Vector right)
  {
```

```
      return new Vector(right.x * scalar, right.y * scalar, right.z * scalar);
   }
   public static Vector operator *(Vector left, float scalar)
   {
      return new Vector(left.x * scalar, left.y * scalar, left.z * scalar);
   }
   public static Vector operator *(Vector left, Vector right)
   {
      return new Vector(left.x * right.x, left.y * right.y, left.z * right.z);
   }
   public static Vector operator +(Vector left, Vector right)
   {
      return new Vector(left.x + right.x, left.y + right.y, left.z + right.z);
   }
   public static Vector operator -(Vector left, Vector right)
   {
      return new Vector(left.x - right.x, left.y - right.y, left.z - right.z);
   }
   public static Vector operator -(Vector left)
   {
      return new Vector(-left.x, -left.y, -left.z);
   }

   public static bool operator ==(Vector left, Vector right)
   {
      return (left.x == right.x && left.y == right.y && left.z == right.z);
   }
   public static bool operator !=(Vector left, Vector right)
   {
      return (left.x != right.x || left.y != right.y || left.z != right.z);
   }
   public static Vector operator /(Vector left, float scalar)
   {
      Vector vector = new Vector();
   // get the inverse of the scalar up front to avoid doing multiple divides
later
      float inverse = 1.0f / scalar;

      vector.x = left.x * inverse;
      vector.y = left.y * inverse;
      vector.z = left.z * inverse;

      return vector;
   }
   #endregion Operators
   public Vector reflect(Vector normal)
   {
   return this - (2 * this.dot(normal) *normal);
   }
   public static Vector magProduct(Vector v,float f)
   {
      return new Vector(v.X * f, v.Y * f, v.Z * f);
   }

   public Vector toPoint()
   {
      Vector p = new Vector(this.X,this.Y,this.Z);
      return p;
   }
   public Vector lerp(Vector v, float t)
```

```
  {
     Vector vector=new Vector();
     vector.x = this.x + t * (v.x - this.x);
     vector.y = this.y + t * (v.y - this.y);
     vector.z = this.z + t * (v.z - this.z);
     return vector;
  }
}
```

2.2 Bazy ortonormalne i współrzędne sferyczne

Programy graficzne zazwyczaj używają wielu układów współrzędnych [7]. Przykładowo, w symulatorze lotu, samolot ustawiony jest w układzie współrzędnym wraz ze skrzydłami, kokpitem oraz fuselage. Tego rodzaju układ najczęściej zdefiniowany jest za pomocą trzech ortonormalnych wektorów u, v oraz w, są one jednostkowe, liniowo niezależne i prawoskrętne ($u \times v = w$). Te trzy wektory noszą nazwę wektorów bazowych układu współrzędnych. Jako grupa stanowią ortonormalną bazę (ONB). Pewne wybrane wektory x,y,z ONB noszą nazwę bazy kanonicznej. Jest to naturalna baza dla naszej aplikacji i wszystkie inne bazy tworzone są na jej podstawie. Przykładowo każdy wektor może być określony za pomocą współrzędnych x,y,z:

$$a = (a_x, a_y, a_z) = a_x x + a_y y + a_z z.$$

Przyzwyczajenie do tego zapisu powoduje, że często zapomina się co te współrzędne oznaczają. Współrzędne (a_x, a_y, a_z) obliczane są za pomocą następującego iloczynu skalarnego:

$$a_x = a \times x,$$
$$a_y = a \times y,$$
$$a_z = a \times z.$$

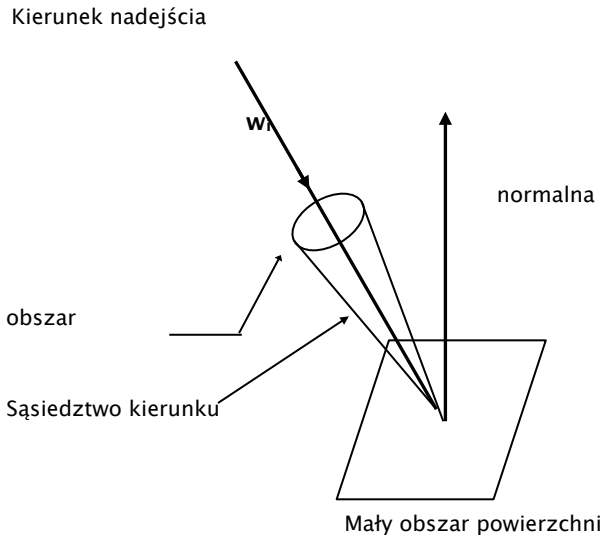
Podobne postępowanie jest dla każdej ONB. Przykładowo:

$$a = (a \times u)u + (a \times v)v + (a \times w)w.$$

W programach często zapisujemy te współrzędne jako (a_u, a_v, a_w) . Istnieje możliwość, że mając dwa różne wektory a oraz a-uvw, o różnych wartościach mogą reprezentować ten sam zwrot. Pierwszy wektor nie określiliśmy jako wektor a-xyz dlatego, że domyślnym układem współrzędnych jest kanoniczny. Geometrycznie te wektory prezentować się będą identycznie mimo że mają inne wartości współrzędnych. Można pomnożyć je przez inną bazę wektorową nim będziemy porównywać ich współrzędne. W przyszłości zaistnieje potrzeba użycia nie kanonicznego układu współrzędnych.

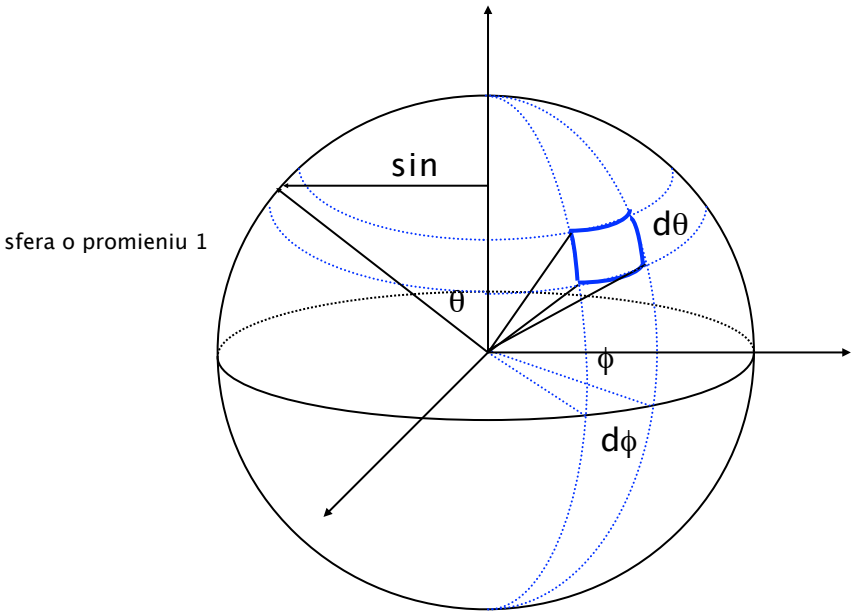
Wprowadzenie sferycznego układu współrzędnych będzie ważne do opisu zachowań światła opisanych za pomocą funkcji BRDF. Musimy dobrze określić w jaki sposób światło dociera do określonego obszaru płaszczyzny z określonego kierunku. W

tym celu należy wprowadzić pojęcie *różnicowego kąta bryłowego*. Kiedy mówimy o świetle nadchodzącym czy odchodzącym, właściwsze jest rozważanie porcji światłą przechodzącego przez określoną przestrzeń. Dlatego też określa się przepływ pewnej ilości światła przez określoną przestrzeń. Korzystając z zależności wprowadzonych w radiometrii, światło będziemy określać jako energię na obszar powierzchni (W/m^2). Oznacza to że nie ma specjalnego sensu mówić o pojedynczym kierunku tylko o całej części obszaru. Poniższy rysunek ilustruje kierunek nadejścia światła jako pewien obszar w przestrzeni do małego obszaru płaszczyzny oświetlanego obiektu



Rys.5 Kąt bryłowy

Idea kąta bryłowego jest dość teoretyczna i należałoby w jakiś sposób zaproksymować tę koncepcję. Nim do tego przejdziemy rozważmy czym jest taki obszar i w tym celu posłużymy się pojęciem obszaru na sferze.



Rys.6 Idea kąta bryłowego

Piramidalny obszar wyznaczony na sferze reprezentuje obszar pewnego kierunku. Część jednostkowej sfery przeciętej z piramidą wyznacza na sferze pewien płat. Różniczkowy kąt bryłowy możemy zdefiniować jako obszar tego płatu. Biorąc współrzędne we współrzędnych sferycznych (θ,ϕ) i kąt między płaszczyzną a piramidą, dθ, dϕ, kąt bryłowy *dw* określimy jako

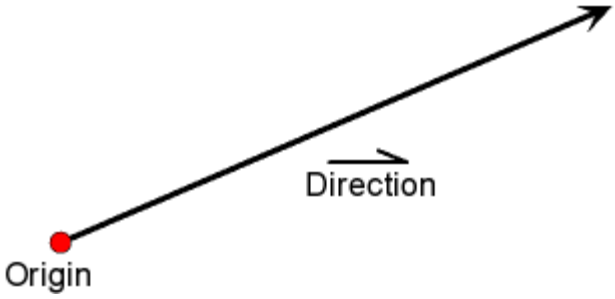
$$dw = (height)(width)$$
$$dw = (d\grave{e})(\sin \grave{e}\, d\ddot{o})$$
$$dw = \sin \grave{e}\, d\grave{e}\, d\ddot{o}$$

Podczas gdy szerokość i wysokość wyznaczonej powierzchni mierzona jest w radianach, obszar musi być mierzony w jednostce kwadratowej (stereoradiany). W praktyce nie musimy zawsze przejmować się dokładną definicją takiego kąta. Rozważamy często tylko pewną powierzchnię małego obszaru i określoną dla pewnego kąta.

3. Przecięcia promienia z obiektami

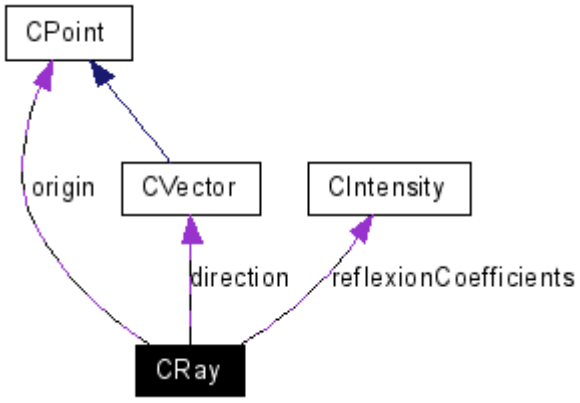
3.1 Promień

Promień reprezentuje drogę którą przebywa światło w wirtualnej scenie. Zgodnie z omówionym wcześniej algorytmem raytracingu, wysłany zostaje od obserwatora przez płaszczyznę rzutowania w głąb sceny. Następnie należy znaleźć przecięcie z najbliższym obiektem. Uogólniając problem, należy znaleźć punkt (o ile taki istnieje) na prostej 3d przecinającej powierzchnię 3d. Prosta zawsze zdefiniowana będzie za pomocą równania parametrycznego. Powierzchnia może być zdefiniowana za pomocą równania uwikłanego lub parametrycznego. Znalezienie przecięcia sprowadzać się będzie do rozwiązania prostego równania algebraicznego.



Rys.7 Definicja promienia

Promień po zdefiniowaniu wektora jest kolejną konieczną definicją dla systemu renderującego. Promień zdefiniowany jest przez punkt origin (początek) oraz przez wektor kierunkowy.



Rys.8 Diagram dla klasy promień

Program 6 Przykładowa definicja promienia w C++

```
class cRay
{
public:
   cVector3 origin,
                  direction,
                  destination;
   float distance;

public:
   cRay()
   {
      origin = cVector3::vZero;
      direction = cVector3::vZero;
      distance = 0.0f;
   }
   cRay( cVector3 o, cVector3 d )
   {
      origin = o;
      direction = d;
   }
```

3.1.1 Równanie parametryczne prostej

Prosta 2D może być zdefiniowana za pomocą równania y=ax+b lub parametrycznie. Równanie parametryczne zawiera pewien rzeczywisty parametr, określający pozycję na prostej. Przykładowo, równanie parametryczne prostej 2d może wyglądać następująco:

$$x(t) = 2 + 7t$$
$$y(t) = 1 + 7t$$

Taki zapis jest równoważny z zapisem:

$$y = x - 1$$

Analogicznie chcąc zapisać równanie parametryczne prostej 3d napiszemy:

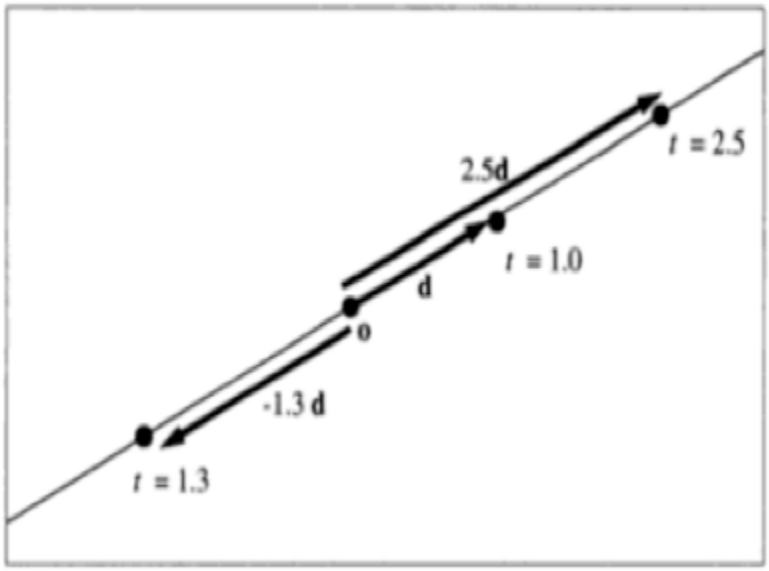
$$x(t) = 2 + 7t,$$
$$y(t) = 1 + 2t,$$
$$z(t) = 3 - 5t.$$

Używając ogólnego zapisu wektorowego:

$$p(t) = o + td$$

dla rozważanego przykładu o oraz d wynoszą odpowiednio:

o=(2,1,3), d=(7,2,-5).



Rys.9 Punkt o może być wyrażony za pomocą równania p(0)=o+0d i punkt t=2,5 można przedstawić za pomocą równania p(2,5)=o+2,5d. Każdy punkt wzdłuż prostej odpowiada pewnej wartości parametru t i każda wartość t odpowiada punktowi na prostej.

Ilustracją prostej parametrycznej jest linia przechodzącą przez o i równoległa do wektora d. Biorąc dowolną wartość parametru t otrzymamy punkt p(t) leżący na prostej. Przyjrzyjmy się następującemu przykładowi. Niech t =2, z równania prostej otrzymamy:

$$p(t) = (2,1,3) + 2(7,2,-5) = (16,5,-7)$$

Pewien segment prostej może być opisany przez prostą parametryczną 3d z pół otwartym przedziałem $t \in [t_a;t_b]$. Segment prostej pomiędzy punktami a i b otrzymamy z równania p(t) = a+t(b-a) gdzie $t \in [0;1]$. Gdzie p(0)=a, p(1)=b oraz p(0,5)=(a+b)/2 i jest punktem środkowym pomiędzy a i b.

Promień lub inaczej półprosta, jest parametryczną prostą z pół otwartym przedziałem zazwyczaj z przedziału [0;∞). Dla nas wszystkie proste, półproste i promienie będą promieniami. Jest to pewne uproszczenie pozwalające na pewien skrót myślowy pomocny w dalszych rozważaniach.

Zazwyczaj przypisuje się pewien przedział z każdym promieniem, określając w ten sposób min i max parametryzacji danego promienia. Tego rodzaju podejście będzie użyteczne w przyszłości, w momencie rzucania promieni cienia w celu sprawdzenia czy dana powierzchnia znajduje się w cieniu. Rzucając promienie cienia w kierunku źródła światła chcemy wiedzieć czy pomiędzy światłem a badanym punktem znajduje się obiekt, to co znajduje się na promieniu po za źródłem światła nie będzie nas już interesowało. Przedział może być zdefiniowany jako składowa promienia przechodzącego przez punkt przecięcia lub jako parametr odpowiedniej funkcji. Można zdefiniować te dwie wartości jako t_{min} oraz t_{max}.

3.2 Prymityw

3.2.1 Sfera

Sfera to zbiór wszystkich punktów w przestrzeni oddalonych dokładnie o zadaną odległość (zwaną promieniem sfery) od wybranego punktu (zwanego środkiem sfery). Chcąc zdefiniować sferę należy podać jej środek (punkt czy wektor) i promień (liczba rzeczywista).

Sfera o środku w $c = (c_x, c_y, c_z)$ i promieniu R może być przedstawiona za pomocą równania uwikłanego:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - R^2 = 0.$$

Możemy zapisać to równanie w formie wektorowej:

$$(p - c) \bullet (p - c) - R^2 = 0.$$

Każdy punkt p spełniający to równanie będzie się znajdował na powierzchni sfery. Mając równanie promienia $p(t) = o + td$, możemy rozwiązać układ równań ze względu na parametr t. Wyznaczając w ten sposób punkty przecięcia promienia ze sferą:

$$(o + td - c) \times (o + td - c) - R^2 = 0.$$

Grupując powyższe równanie otrzymamy:

$$(d \times d)t^2 + 2d \times (o - c)t + (o - c) \times (o - c) - R^2 = 0.$$

Pozostaje rozwiązać klasyczne równanie kwadratowe ze względu na parametr t:

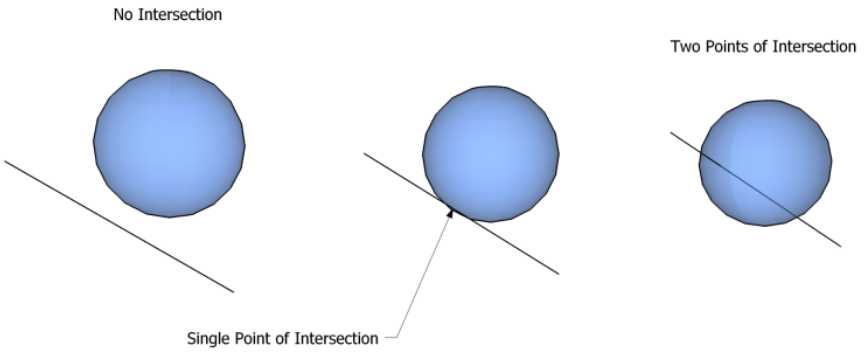
$$At^2 + Bt + C = 0.$$

Rozwiązanie równania:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

Wyrażenie pod pierwiastkiem: $B^2 - 4AC$ nazywamy wyróżnikiem kwadratowym (osławiona przez pierwsze lekcje programowania delta). Wyróżnik mówi nam ile rozwiązań będzie miało dane równanie. Istnieją trzy przypadki:

- $B^2 - 4AC < 0$ - nie ma rzeczywistych pierwiastków równania, oznacza to że promień minął sferę i nie ma punktu przecięcia
- $B^2 - 4AC > 0$ - istnieją dwa rzeczywiste pierwiastki równania kwadratowego, oznacza to że promień przeciął sferę w dwóch punktach (wejście i wyjście promienia przez sferę)
- $B^2 - 4AC = 0$ - istnieje jeden pierwiastek równania, oznacza to ze promień jest styczny do sfery.



Rys.10 Przypadki przecięcia promienia i sfery

Podstawiając do współczynników wartości, rozwiązanie równania będzie:

$$t = \frac{-2\mathrm{d} \times (\mathrm{o}-\mathrm{c}) \pm \sqrt{(2\mathrm{d} \times (\mathrm{o}-\mathrm{c}))^2 - 4(\mathrm{d} \times \mathrm{d})((\mathrm{o}-\mathrm{c}) \times (\mathrm{o}-\mathrm{c}) - \mathrm{R}^2)}}{2 \times \mathrm{d} \times \mathrm{d}}.$$

Skracając 2 otrzymamy:

$$t = \frac{-\mathrm{d} \times (\mathrm{o}-\mathrm{c}) \pm \sqrt{(\mathrm{d} \times (\mathrm{o}-\mathrm{c}))^2 - (\mathrm{d} \times \mathrm{d})((\mathrm{o}-\mathrm{c}) \times (\mathrm{o}-\mathrm{c}) - \mathrm{R}^2)}}{\mathrm{d} \times \mathrm{d}}.$$

Wektor normalny w punkcie p znajdujemy za pomocą gradientu n=2(p-c). Znormalizowany wektor normalny wyniesie (p-c)/R.

Program 7 Pseudo kod obliczający przecięcie ze sferą w C++

```
int Sphere::Intersect( Ray& a_Ray, float& a_Dist )
{
        vector3 v = a_Ray.GetOrigin() - m_Centre;
        float b = -DOT( v, a_Ray.GetDirection() );
        float det = (b * b) - DOT( v, v ) + m_SqRadius;
        int retval = MISS;
        if (det > 0)
        {
                det = sqrtf( det );
                float i1 = b - det;
                float i2 = b + det;
                if (i2 > 0)
                {
```

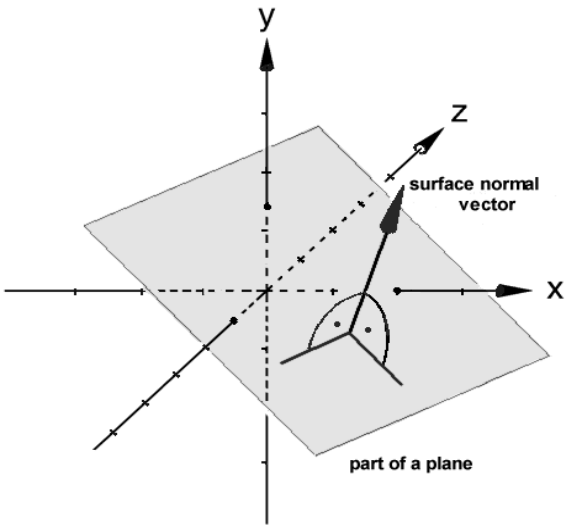
```
                        if (i1 < 0)
                        {
                                if (i2 < a_Dist)
                                {
                                        a_Dist = i2;
                                        retval = INPRIM;
                                }
                        }
                        else
                        {
                                if (i1 < a_Dist)
                                {
                                        a_Dist = i1;
                                        retval = HIT;
                                }
                        }
                }
        }
        return retval;
}
```

3.2.2 Płaszczyzna

W przestrzeni euklidesowej R^3 płaszczyzna jest zbiorem punktów, których współrzędne spełniają w danym kartezjańskim układzie współrzędnych równanie:

$$Ax + By + Cz + D = 0$$

przy czym liczby A, B, C nie mogą być jednocześnie równe zeru. Jest to tak zwane **równanie ogólne płaszczyzny**. Wektor [A, B, C] jest wektorem normalnym prostopadłym do tej płaszczyzny.



Rys.11 Normalna do płaszczyzny[2]

[2] © Friedrich A. Lohmueller, 2001

Zatem płaszczyznę możemy zdefiniować za pomocą dowolnego punktu (x,y,z), który do niej należy (spełnia równanie ogólne) oraz wektora normalnego [A, B, C] wyznaczającego orientację płaszczyzny w przestrzeni. Wykorzystując **reprezentację wektorową** możemy następująco opisać płaszczyznę:

$$N \text{ dot } Xp = d$$

gdzie:
N = [A,B,C] - wektor normalny płaszczyzny.
Xp - to punkt na płaszczyźnie.
dot - iloczyn skalarny wektorów
d = -D jest liczbą będąca przesunięciem, wzdłuż normalnej, płaszczyzny od środka układ współrzędnych.

Zdefiniowane we wcześniejszych punktach równanie wektorowe promienia ma postać:

$$X = Xr + t * V$$

gdzie:
Xr - punkt początka promienia
V - wektor kierunkowy promienia
X - dowolny punkt leżący na promieniu
t - wartość skalarna opisująca przesunięcie od Xr wzdłuż wektora V

Jeżeli promień przecina płaszczyznę, to musi istnieć taki punkt na promieniu, który spełnia równanie płaszczyzny. Zatem podstawiając z równania promienia do równania płaszczyzny Xp = X otrzymujemy:

$$N \text{ dot } (Xr + t * V) = d \quad => \quad N \text{ dot } Xr + t * (N \text{ dot } V) = d$$

Z powyższego równania możemy obliczyć t:

$$t = (d - N \text{ dot } Xr) / (N \text{ dot } V)$$

Aby wyznaczyć szukany punkt przecięcia (kolizji promienia z płaszczyzną) wystarczy podstawić t do równania promienia.

Program 8 Fragment kodu w C++ obliczający przecięcie promienia z płaszczyzną (bez uwzględnienia przypadków szczególnych)

```
{
    float wX = x0 - x1;
    float wY = y0 - y1;
    float wZ = z0 - z1;
    float sqr = sqrt ((wX*wX) + (wY*wY) + (wZ*wZ));
```

```
    wX /= sqr;
    wY /= sqr;
    wZ /= sqr;

    float dot1 = (x0*A + y0*B + z0*C) + D;
    float dot2 = (wX*A + wY*B + wZ*C);

    float ratio = dot1/dot2;

    float x_cross = x0 - (wX * ratio);
    float y_cross = y0 - (wY * ratio);
    float z_cross = z0 - (wZ * ratio);
}
```

Należy jednak rozpatrzyć dwa przypadki szczególne:

1. Jeżeli N dot V = 0 to wektory są prostopadłe (promień biegnie równolegle do płaszczyzny). Zatem nie będzie kolizji.
2. Jeżeli t jest ujemne to przecięcie jest przed punktem startu promienia. Zatem również nie ma kolizji.

4. Zadanie

1. Należy zaimplementować klasę wektor, promień, sfera i płaszczyzna.
2. Zdefiniować sferę S o środku w punkcie (0,0,0) i promieniu 10.
3. Zdefiniować promień R1 o początku w punkcie (0,0,-20) i skierowany w środek kuli.
4. Zdefiniować promień R2 o początku w tym samym punkcie, co R1, skierowany równolegle do osi Y.
5. Proszę sprawdzić, czy istnieje przecięcie sfery S z promieniami R1 oraz R2. Wynik w postaci współrzędnych punktu przecięcia należy wyświetlić.
6. Proszę zdefiniować dowolny promień R3, tak aby przecinał on sferę S w dokładnie jednym punkcie. Podać współrzędne punktu przecięcia.
7. Proszę zdefiniować płaszczyznę P przechodzącą przez punkt (0,0,0), której wektor normalny tworzy kąt 45 stopni z osiami Y i Z.
8. Proszę znaleźć punkt przecięcia płaszczyzny P z promieniem R2.

5. Bibliografia

[1] **Ashdown Ian, P. Eng., LC, FIES.** Photometry and Radiometry A Tour Guide for Computer Graphics Enthusiasts. : John Wiley & Sons in 1994

[2] **Woźniak Władysław Artur.** Radiometria i Fotometria. : Instytut Fizyki Politechniki Wrocławskiej 2004

[3] **Wynn Chris.** An Introduction to BRDF-Based Lighting : NVIDIA Corporation 2000

[4] **Rusinkiewicz Szymon.** A Survey of BRDF Representation for Computer Graphics : CS348c, Winter 1997

[5] **Glassner, Andrew S.** Space Subdivision for Fast Ray Tracing : IEEE Computer Graphics & Applications, March 1988, volume 8, number 2, pp. 60-70

[6] **Marlon John.** Focus On Photon Mapping : Premier Press 2003 ISBN 1-1-59200-008-8

[7] **Shirley Peter, R. Keith Morley.** Realistic Ray Tracing: Second Edition : AK Peters; 2nd edition (July 2003) ISBN-13: 978-1568811987

[8] **Whitted Turner.** An improved illumination model for shaded display : Communications of the ACM archive. Volume 23 , Issue 6 (June 1980)

[9] **Appel Arthur.** The notion of quantitative invisibility and the machine rendering of solids : Proceedings of ACM National Conference 1967

[10] **Sunday Dan.** Intersections of Rays, Segments,Planes and Triangles in 3D : softSurfer 2006

[11] **Maciej Falski** Przegląd modeli oświetlenia w grafice komputerowej: Praca magisterska Uniwersytet Wrocławski Wydział Matematyki i Informatyki, Instytut Informatyki 2004

[12] **Christophe Schlick.** An inexpensive BRDF model for physically-based rendering : Computer Graphics Forum 1994

[1in] **Dr inż. Władysław Artur Woźniak.** Strona informacyjna Instytut Fizyki . : http://www.if.pwr.wroc.pl/~wozniak/

[2in] **The Australian National UniversityFaculty of Engineering and Information Technology (FEIT) .** Global Illumination ModelsPhysically

Based Illumination, Ray Tracing and Radiosity : http://escience.anu.edu.au/lecture/cg/GlobalIllumination/printNotes.en.html

[4in] **Softsurfer.** List of Algorithm Titles in the softSurfer Archive. . : http://softsurfer.com/algorithm_archive.htm

[5in] **Henrik Wann Jensen.** Strona informacyjna Henrika Wann Jensena. . : http://www.gk.dtu.dk/~hwj

[6in] **Jiajun Zhu** CS 645 Computer Graphics http://www.cs.virginia.edu/~jz8p/