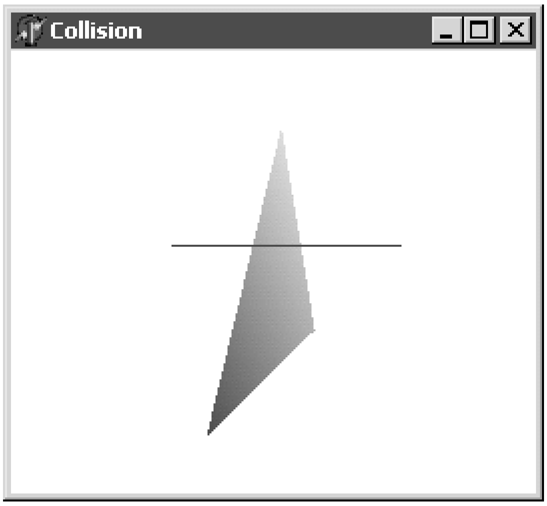
فصل بیست و پنجم

تشخیص تصادم



مقدمه :

یکی از موارد مهم در برنامه نویسی بازی ها و همچنین شبیه سازی هـای فیزیکـی ، تشـخیص تصادم بین اشیاء مختلف می باشد . تشخیص برخورد بین اشیاء مختلف ، موضوعی پیچیده بوده و راه حل ساده ای برای آن وجود ندارد . همانگونه کـه پـیش تـر نیـز ذکـر شـد ، OpenGL صـرفا کتابخانه ای برای انجام ترسیمات ۲ یا ۳ بعدی می باشد و در ایـن زمینـه امکانـاتی را در اختیـار برنامه نویس قرار نمی دهد . در این فصل که مطالعه آن نیاز به دانش کمی در باره هندسه تحلیلی نیز دارد ، به این موضوع پرداخته خواهد شد .

تعاریف اولیه :

در این فصل علامت گذاری های زیر بکار رفته است :

۱– ضرب درونی (Dot product) بردارها به صورت "|" نمایش داده شده است .

۲- len(V) به معنای اندازه و طول بردار V است .

۳- dot(V) معادل مجذور طول بردار V است (ضرب درونی با خودش).

۴- nrm(V) به مفهوم بردار V نرمالایز شده می باشد ((V/Sqrt(V|V)) .

یک اشعه (ray) به صورت زیر تعریف می شود :

$$P = O + D*t$$

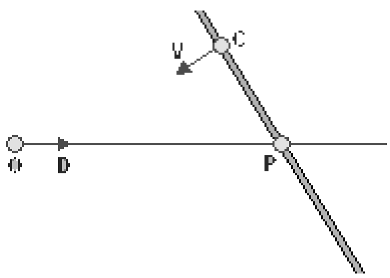
که در آن O مبدا اشعه و D جهت آن می باشد . t فاصله نقطه تصادم از مبدا بوده و معادله فوق برای بدست آوردن آن حل خواهد شد . اگر t منفی باشد بدین معنا است که نقطه برخورد پشت مبدا قرار گرفته است و نیز آنسوی علاقه ما در این فصل ! مگر اینکه تمام نقاط یک شکل در پشت مبدا قرار گرفته باشند .

در طول این فصل از معادله P به شکل زیر استفاده می شود :

$$P - C = D*t + X$$

که در آن C نقطه مرکزی شکلی است که با آن تصادم صورت گرفته است و X مساوی O-C می باشد .

تشخیص تصادم اشعه با یک صفحه :



تعریف :

C نقطه ای است که بر روی خط قرار می گیرد .

V بردار نرمال صفحه است (به طول واحد) .

برای برخورد با یک صفحه باید :

$$(P-C) | V = 0$$

در اینصورت بردار P-C عمود است بر بردار نرمال و اگر رابطه فوق برقرار باشد نقطه P بر روی صفحه قرار گرفته است .

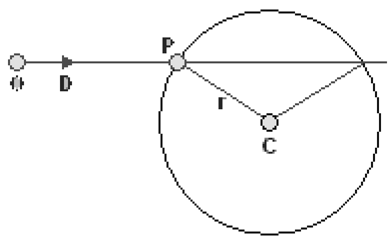
حل معادله فوق (برای بدست آوردن t) :

```
(D*t + X)|V = 0
D|V*t = -X|V
t = -X|V / D|V
```

پیش از انجام تقسیم فوق باید مطمئن شد که مخرج کسر صفر نمی باشد . همچنین می توان بررسی کرد که D|V و X|V مختلف العلامه هستند یا خیر ؟ (در غیر اینصورت، t حاصل منفی خواهد بود .)

بردار نرمال سطح در نقطه P ، با بردار نرمال صفحه معادل می باشد مگر اینکه D|V منفی باشد (N=-V) .

تشخیص تصادم اشعه با یک کره :



تعریف :

C مرکز کره است .

r شعاع کره می باشد .

برای برخورد با کره باید معادله زیر حاکم باشد :

```
len(P-C) = r
```

مفهوم معادله فوق این است که فاصله بین مرکز کره و نقطه تصادم مساوی r است و هنگامی صادق خواهد بود که P بر روی سطح کره قرار گیرد .

حل معادله فوق :

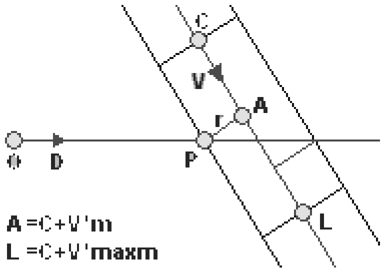
```
len(D*t+X) = r
dot(D*t+X) = r^2
D|D*(t^2) + 2*D|X*t + X|X - r^2 = 0
```

بنابراین برای حل معادله درجه دوم فوق می توان ضرایب زیر را تعریف کرد :

```
a   = D|D
b/2 = D|X
c   = X|X - r*r
```

و بردار عمود بر سطح N = nrm(P-C) می باشد .

تشخیص تصادم اشعه با یک استوانه :



تعریف :

C : نقطه آغازین راس (Cap) استوانه است .

r : شعاع استوانه است .

V : برداری به طول واحد که بیانگر محور استوانه است .

maxm : نقطه راس انتهایی استوانه است .

برای برخورد با استوانه باید :

```
A = C + V*m
( P-A )|V = 0
len( P-A ) = r
```

m تعیین کننده نزدیکترین نقطه روی محور با نقطه برخورد است . بردار P-A بر V عمود است و نزدیکترین فاصله به محور را تعیین می کند . P-A شعاع استوانه است .

حل :

```
(P-C-V*m)|V = 0
(P-C)|V = m*(V|V) = m   (len(V)=1)
m = (D*t+X)|V
m = D|V*t + X|V

len(P-C-V*m) = r
dot( D*t+X - V*(D|V*t + X|V) ) = r^2
dot( (D-V*(D|V))*t + (X-V*(X|V)) ) = r^2
dot( A-V*(A|V) ) = A|A - 2*(A|V)^2 + V|V * (A|V)^2 =
```

```
= A|A - (A|V)^2
a*t^2 + b*t + c = 0
a = D|D - (D|V)^2
c = X|X - (X|V)^2 - r^2
b = 2 * (D-V*(D|V))|(X-V*(X|V)) =
  = 2 * (D|X - D|V*(X|V) - X|V*(D|V) + (D|V)*(X|V)) =
  = 2 * (D|X - (D|V)*(X|V))
```

عاقبت :

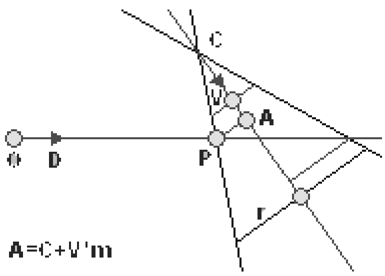
```
  a   = D|D - (D|V)^2
b/2 = D|X - (D|V)*(X|V)
  c   = X|X - (X|V)^2 - r*r
```

و بردار عمود بر سطح به صورت زیر است :

```
m = D|V*t + X|V
N = nrm( P-C-V*m )
```

دو نقطه روی استوانه وجود دارند که تصادم با آنها صورت گرفته است (و یا یک نقطه دوبار). باید ۲ مقدار m محاسبه گردیده و بررسی شود که آیا در بازه [0,maxm] قرار می گیرند یا خیر ؟

تشخیص تصادم اشعه با یک مخروط :



تعریف :

C راس مخروط است .

V بردار محور می باشد .

K تانژانت نیم زاویه مخروط است .

minm و maxm نقاط Cap را تعریف می کنند .

برای برخورد با مخروط باید :

```
A = C + V*m
( P-A )|V = 0
len( P-A )/m = k
```

حل :

```
m = D|V*t + X|V     (like for cylinder)
len( P-C-V*m ) = m*k
dot( D*t+X - V*(D|V*t + X|V) ) = k^2 * ( D|V*t + X|V )^2
dot( (D-V*(D|V))*t + X-V*(X|V) ) = k^2 * ( D|V*t + X|V )^2
```

اکنون ضرایب سمت چپ سه عبارت مشابه استوانه می باشند :

```
a   = D|D - (D|V)^2
b/2 = D|X - (D|V)*(X|V)
c   = X|X - (X|V)^2
```

و ضرایب سمت راست :

```
k^2 * ( D|V*t + X|V )^2 =
= k^2 * ( (D|V)^2 * t^2 + 2*(D|V)*(X|V)*t + (X|V)^2 )
a   = k*k*(D|V)^2
b/2 = k*k*(D|V)*(X|V)
c   = k*k*(X|V)^2
```

عاقبت :

```
a   = D|D - (1+k*k)*(D|V)^2
b/2 = D|X - (1+k*k)*(D|V)*(X|V)
c   = X|X - (1+k*k)*(X|V)^2
```

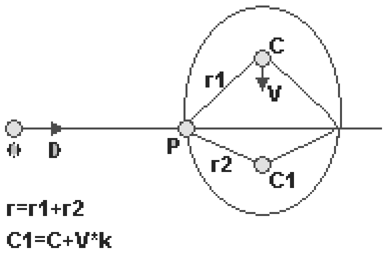
برای محاسبه بردار نرمال بر سطح باید خاطر نشان کرد که بردار نرمالایز شده زیر را داریم :

```
(P-C-V*m) - V*a
```

اما a چگونه محاسبه می شود ؟ زاویه بین نرمال و P-C-V*m باید با نیم زاویه مخروط یکسان باشد :

```
a/r = k
r/m = k
a = m*k*k
N = nrm( P-C-V*m - V*m*k*k )
N = nrm( P-C - (1+k*k)*V*m )
```

تشخیص تصادم اشعه با یک بیضیگون :



تعریف :

C یکی از دو مرکز بیضیگون است .

K فاصله بین دو مرکز است .

V برداری است به طول واحد ، از مرکز C به طرف مرکز دیگر .

r : جمع شعاع ها است .

برای تصادم با یک بیضیگون باید :

```
len( P-C ) + len( P-(C+V*k) ) = r
```

این بدان معنا است که جمع فواصل از P به مراکز مساوی r است .

حل :

```
len( P-C-V*k) ) = r - len( P-C )
dot( D*t + X-V*k ) = r*r - 2*r*len( D*t+X )
dot( D*t + X-V*k ) =
= D|D*t^2 + 2*D|(X-V*k)*t + dot(X-V*k) =
= D|D*t^2 + 2*(D|X-D|V*k)*t + X|X-2*X|V*k+k
dot( D*t + X ) =
= D|D*t^2 + 2*(D|X)*t + X|X
2*D|V*k*t - 2*X|V*k + k - r*r = -2*r*len( D*t + X )
(2*D|V*k*t - 2*X|V*k + k-r*r)^2 =
= 4*r*r*( D*D*t^2 + 2*(D|X)*t + X|X )
```

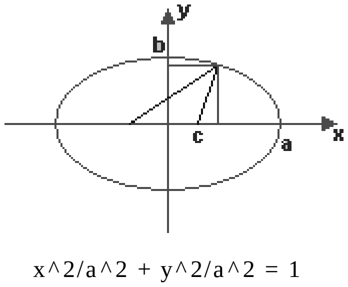
و ضرایب نهایی :

```
a   = 4*r*r*D|D - 4*k*k*(D|V)^2
b/2 = 4*r*r*D|X - 2*(D|V)*k * (r*r+2*X|V*k-k)
c   = 4*r*r*X|X - (r*r+2*X|V*k-k)^2
```

و برای ساده سازی آن می توان نوشت :

```
A1  = 2*k*(D|V)
A2  = r^2 + 2*k*(X|V) - k
a   = 4*r^2*D|D - A1^2
b/2 = 4*r^2*D|X - A1*A2
c   = 4*r^2*X|X - A2^2
```

محاسبه بردار نرمال یک بیضیگون پیچیده تر از حالت های پیشین می باشد . اگر بیضی ای که بیضیگون ما را تشکیل می دهد ، همانند تصویر زیر باشد ، داریم :




```
x^2/a^2 + y^2/a^2 = 1
```

برای قسمت بالایی تصویر :

y = b * sqrt(1 - x^2/a^2)

مشتق این تابع به صورت زیر است :

f'(x) = -b/a^2 * x / sqrt(1 - x^2/a^2)

عکس مشتق (1/f'(x)) ، نسبت نرمال های سطح y/x می باشد :

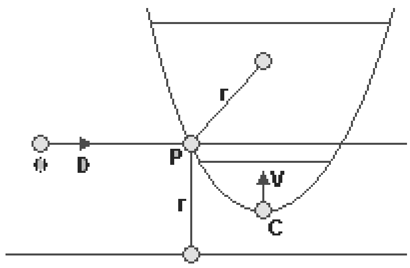
1/f'(x) = a^2/b * sqrt(1/x^2 - 1/a^2)

در تابع اصلی بیضیگون ما ، بردار V تعیین کننده محور x است . اکنون ما می خواهیم بدانیم که چه مقداری از V را باید به بردار P-C-V*k/2 اضافه کنیم (C+V*k/2 نقطه ای است بین دو مرکز بیضیگون) تا بردار نرمال صحیح بدست آید (بعد از نرمال سازی بردار نهایی) . از آنجائیکه باید نسبت y/x را در a^2/b^2 ضرب کنیم ، تنها کافی است که x را به آن فاکتور تقسیم کنیم . چون b^2/a^2<1 با تفریق x*(1-b^2/a^2) از x ، مقدار صحیح بدست می آید .

بدلیل اینکه V مستقیما به x مربوط است ، ما باید آنرا در آن فاکتور ضرب کنیم و سپس از مقدار P-C-V*k/2 کسر کنیم . مقدار x در این حالت m=(P-C-V*k/2)|V می باشد .

Cmid = C + V*k/2
R = P - Cmid
N = nrm(R - V*(1-b^2/a^2)*(R|V))

تشخیص تصادم اشعه با یک سهمیگون :



تعریف :

C نقطه حداکثر و بیشینه سهمیگون می باشد .

V برداری است به طول واحد که جهت سهمیگون را معین می کند .

K عددی است که بیانگر فاصله هسته (Kernel) سهمیگون و صفحه عبوری از C می باشد .

برای تصادم با یک سهمیگون خاطر نشان می شود که هر نقطه روی سطح سهمیگون فاصله مشابهی از هسته(C+V*k) دارد ، همانند صفحه (C-V*k, V) .

برای محاسبه فاصله از صفحه فرض کنید کـه نقطـه A روی صـفحه قـرار دارد و r فاصـله ای است که آنرا جستجو می کنیم :

```
P = A + V*r
A = P - V*r
(A-(C+V*k))|V = 0
(P-C+V*k-V*r)|V = 0
(P-C+V*k)|V - V|V*r = 0
r = (P-C+V*k)|V
```

اکنون معادله به صورت زیر می باشد :

```
(P-C+V*k)|V + k = r
len( P-(C+V*k) ) = r
```

حل :

```
len( P-C-V*k ) = (P-C+V*k)|V
dot( D*t + X-V*k ) = (D|V*t + X|V+k)^2
D|D*(t^2) + 2*(D|(X-V*k))*t + dot(X-V*k) =
= (D|V)^2*(t^2) + 2*D|V*(X|V+k)*t + (X|V+k)^2

a   = D|D - (D|V)^2
b/2 = D|X - D|V*k - D|V*(X|V+k) =
    = D|X - D|V*( X|V + 2*k )
c   = X|X - 2*X|V*k + k^2 - (X|V)^2 - 2*(X|V)*k - k^2 =
    = X|X - (X|V)^2 - 4*(X|V)*k =
    = X|X - X|V*( X|V + 4*k )
```

عاقبت :

```
a   = D|D - (D|V)^2
b/2 = D|X - D|V*( X|V + 2*k )
c   = X|X - X|V*( X|V + 4*k )
```

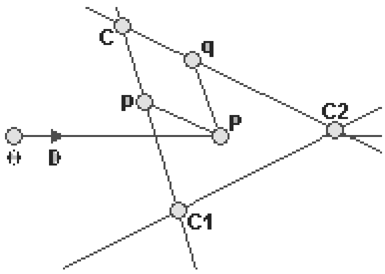
برای محاسبه بردار نرمال باید بـردار P-(C+V*m) + V*a را نرمـالایز کنـیم . در حقیقـت نسـبت مقدار len(P-(C+V*m))/a مشابه Ny/Nx می باشد . اگر سهمیگون را یـک معادلـه ۲ بعـدی معمـولی فرض کنیم y^2=2*k*x ، Nx و Ny مقادیر بردار نرمال ۲ بعدی خواهند بود .

```
f(x) = sqrt( 2*k*x )        our function
f'(x) = k / sqrt( 2*k*x )   derivative
Ny/Nx = 1/f'(x)             the ratio we need
Ny/Nx = sqrt( 2*x/k )
Nx = Ny * sqrt( k/(2*x) )   we are looking for a...
Ny = sqrt( 2*k*x )          ...while we know f(x)
Nx = sqrt( 2*x*k*k/(2*x) )
Nx = k                      and there we are!
```

مقدار a به طول بردار P-(C+V*m) وابسته نیست و مساوی فاصله نقطه حداکثر تا هسته می باشد :

N = nrm(P-C-V*(m+k))

تشخیص تصادم اشعه با یک مثلث :



تعریف :

C یک راس مثلث می باشد (راس مورد نظر) .

V بردار نرمال صفحه مثلث می باشد .

V1 برداری است از راس C به راس C1 (V1=C1-C) .

V2 برداری است از راس C به راس C2 (V2=C2-C) .

برای تصادم با یک مثلث در ابتدا باید با صفحه ای که مثلث در آن قرار می گیرد ، برخورد کرد . سپس دو راس دیگر مثلث در محاسبه اینکه آیا تصادم درون مثلث رخ داده است یا خیر ، به ما کمک می کنند .

t = - X|V / D|V
P = C + V1*p + V2*q

حل :

P - C = V1*p + V2*q
[Px-Cx] = [V1.x V2.x] * [p]
[Py-Cy]   [V1.y V2.y]   [q]

p و q از حل دستگاه ماتریسی فوق بدست می آیند (محاسبه ماتریس معکوس) .

برای اینکه دقیقا تعیین کنیم که آیا نقطه تصادم بر روی مثلث قرار دارد یا خیر باید مطمئن شد که مقادیر p و q و مجموع آنها درون بازه [0,1] قرار دارند یا خیر . اگر تصادمی با مثلث رخ داده باشد از P و q می توان برای محاسبه بردار نرمال نیز استفاده کرد .

برای کاهش محاسبات در این موارد ، بهترین راه تعریف یک حجم مرزی می باشد و سپس آزمایش تصادم با آن .

تشخیص تصادم اشعه با یک سطح دوار :

سطح حاصل از دوران بصورت یک spline دوران داده شده حول یک محور تعریف می شود . عموما spline های درجه ۱ تا ۳ بکار برده می شوند (خطی ، مربعی و مکعبی) . هر جزء سطح حاصل از دوران به صورت زیر تعریف می شود :

C نقطه شروع جزء (قطعه) می باشد .

V برداری به طول واحد که محور را معین می کند .

maxm : نقطه پایانی cap قطعه است .

a,b,c,d : ضرایب قطعه هستند (وابسته به درجه آن) .

ما m را همانند استوانه محاسبه می کنیم (استوانه ، مخروط و سهمیگون حالت خاصی از سطح حاصل از دروان هستند) .

```
m = D|V*t + D|X
```

سپس p را مطابق معادله تعریف شده توسط a,b,c,d محاسبه می کنیم . (فاصله P از محور مساوی است با m-f(m)) .

حل :

```
len( P-(C+V*m) ) = f(m)
(P-(C+V*m))|V = 0

dot( P-C-V*((P-C)|V) ) = f(m)^2
dot( D*t+X-V*(D|V*t+X|V) ) = f(m)^2
dot( (D-V*(D|V))*t + X-V*(X|V) ) = f(m)^2
```

ضرایب سمت چپ :

```
a = D|D - (D|V)^2
b = 2*( D|X - (D|V)*(X|V) )
c = X|X - (X|V)^2
```

ضرایب سمت راست :

```
f(m)^2 = (am^3 + bm^2 + cm + d)^2
f(m)^2 = m^6 * a^2 +
+ m^5 * 2*a*b +
+ m^4 * (b^2 + 2*a*c) +
+ m^3 * 2*(a*d + b*c) +
```

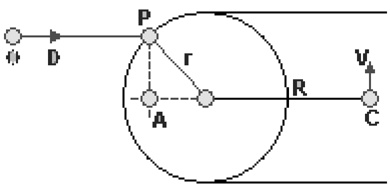
```
+ m^2 * (c^2 + 2*b*d) +
+ m^1 * 2*c*d +
+ m^0 * d^2
m = D|V*t + X|V
```

بنابراین ما یک معادله درجه ۲ تا ۶ برای t که حل معادلات بالاتر از درجه ۲ آن به زمان و تلاش زیادی نیاز خواهد داشت ، داریم . پس بهتر است که خود سطح حاصل از دوران را مدل کنیم بجای اینکه آنرا به مجموعه ای از مثلث ها تبدیل نماییم .

محاسبه بردار نرمال یک روش brute force می باشد :

```
m = D|V*t + X|V
dval = f'(m) = 3*a*m^2 + 2*b*m + c
R = P - C - V*m
N = nrm( R - V*len(R)/dval )
```

تشخیص تصادم اشعه با یک هلالی :



تعریف :

C مرکز هلالی (torus) است .

V برداری است که جهت روبه داخل هلالی را نشان می دهد .

R شعاع بزرگ هلالی است .

r شعاع کوچک هلالی است .

برای تصادم با آن خواهیم داشت :

```
P = A + V*k
(len( A-C ) - R)^2 + k^2 = r^2
```

حل :

```
A = P - V*k
(A-C)|V = 0
(P-C-V*k)|V = 0
k = (P-C)|V

r^2 - k^2 = ( len(P-C-V*k) - R )^2
```

```
Z = P-C = D*t+X          temporary substitution
r^2 - (Z|V)^2 = ( len( Z-V*(Z|V) ) - R )^2
dot(Z-V*(Z|V)) + R^2 - r^2 + (Z|V)^2 = -2*R*len(Z-V*(Z|V))
Z|Z + R^2 - r^2 = -2*R*len(Z-V*(Z|V))
4*R^2*dot(Z-V*(Z|V)) = (Z|Z + R^2 - r^2)^2
4*R^2*(Z|Z-(Z|V)^2) = (Z|Z)^2 + 2*(R^2-r^2)*(Z|Z) + (R^2-r^2)^2
(Z|Z)^2 - 2*(R^2+r^2)*(Z|Z) + 4*R^2*(Z|V)^2 + (R^2-r^2)^2 = 0
```

با کمی جایگزینی :

```
m = D|D, n = D|X, o = X|X, p = D|V, q = X|V

Z|Z = dot(D*t+X) = m*t^2 + n*t + o
Z|V = (D*t+X)|V = p*t + q

(m*t^2 + n*t + o)^2 - 2*(R^2+r^2)*(m*t^2 + n*t + o) +
+ 4*R^2*(p*t + q)^2 + (R^2-r^2)^2 = 0
m^2*t^4 + 4*n^2*t^2 + o^2 + 4*n*o*t + 4*m*n*t^3 +
+ 2*m*o*t^2 + 4*R^2*(p^2*t^2 + 2*p*q*t + p^2) +
+ (R^2-r^2)^2 = 0
```

و ضرایب نهایی عبارتند از :

```
a = m^2
b = 4*m*n
c = 4*m^2 + 2*m*o - 2*(R^2+r^2)*m + 4*R^2*p^2
d = 4*n*o - 4*(R^2+r^2)*n + 8*R^2*p*q
e = o^2 - 2*(R^2+r^2)*o + 4*R^2*q^2 + (R^2-r^2)^2
```

برای محاسبه بردار نرمال ، باید از K استفاده نماییم (فاصله نقطه برخورد تا صفحه اصلی هلالی) .

```
k = (P-C)|V
A = P - V*k
m = sqrt( r^2 - k^2 )
N = nrm( P - A - (C-A)*m/(R+m) )
```

مواردی که در این فصل مرور شدند صرفا مقدمات بسیار ساده شبیه سازی های فیزیکی انواع و اقسام بازی های رایانه ای می باشند . مرور کامل این نوع برنامه نویسی نیاز به یک کتاب کامل و جامع دیگر دارد (شاید وقتی دیگر!) .

برنامه فصل :

در ابتدا واحدی را بنام Math_3D برای انجام محاسبات ریاضی مورد نیاز بوجود می آوریم :

```
unit Math_3D;
```

```
interface

uses
   Math;

const
 // Used to cover up the error in floating point
 MATCH_FACTOR : double = 0.9999;
 ZERO : integer = 0;

type
 VECTOR3D = record
        x, y, z : single ;
 end;

type
   V3D = array of VECTOR3D;

type
  Cylinder=record
       Position : VECTOR3D;
       Axis : VECTOR3D;
       Radius : double;
  end;

type
  Plane=record
       Position : VECTOR3D;
       Normal : VECTOR3D;
 end;


type
   TMatrix33 = Record
     Mx : array[0..2] of array[0..2] of double;
 end;

type
    QUAD = record
           vVertices : V3D;
           vNormal : VECTOR3D;
           D : single;
    end;


function Vertexf(x0, y0 ,z0,
                 x1, y1 ,z1,
                 x2, y2 ,z2 : single): V3D;overload;

function Vertexf(x0, y0 ,z0,
                 x1, y1 ,z1,
                 x2, y2 ,z2,
                 x3, y3 ,z3 : single): V3D;overload;

 function Cross( vVector1,  vVector2 : VECTOR3D):VECTOR3D;
 function Vector( vPoint1,  vPoint2 : VECTOR3D):VECTOR3D;
 function Magnitude( vNormal :VECTOR3D):single;
```

```
 function Normalize( vNormal : VECTOR3D ):VECTOR3D;
 function Normal( vTriangle : V3D ):VECTOR3D;
 function PlaneDistance( Normal, Point :VECTOR3D): single;
 function IntersectedPlane( vTriangle , vLine : V3D ):boolean;
 function IntersectedPlane02( vPoly, vLine : V3D;
                              var vNormal : VECTOR3D ;
                              var originDistance : single):boolean;
 function Dot( vVector1, vVector2 :VECTOR3D ):single;
 function AngleBetweenVectors(Vector1, Vector2 :VECTOR3D ):double;
 function IntersectionPoint(vNormal : VECTOR3D ; vLine : V3D ;
                            distance : double ):VECTOR3D;
 function InsidePolygon( vIntersection : VECTOR3D ; Poly :V3D ;
                         verticeCount : LongInt ): boolean;
 function IntersectedPolygon( vPoly, vLine :V3D ;
                              verticeCount:integer ):boolean;

 function Matrix_making():TMatrix33;overload;
 function Matrix_making( Phi, Theta, Psi :double) : TMatrix33;overload;
 function Matrix_making( mx00, mx01, mx02,
                         mx10, mx11, mx12,
                         mx20, mx21, mx22 :double ): TMatrix33;overload;
 function Matrix_add(m1, m2 :TMatrix33 ):TMatrix33;
 function Matrix_subtract(m1,m2 :TMatrix33 ):TMatrix33;
 function Matrix_multiply( m1 , m2 : TMatrix33):TMatrix33;overload;
 function Matrix_multiply(
              m1 :TMatrix33; scale : double):TMatrix33;overload;
 function Matrix_multiply(
                m1 : TMatrix33; v :VECTOR3D):VECTOR3D;overload;
 function Matrix_determinant(m : TMatrix33):double;
 function Matrix_transpose(m : TMatrix33):TMatrix33;
 function Matrix_inverse(m1 : TMatrix33 ):TMatrix33;

 function Vector_add( v1 , v2 : VECTOR3D) : VECTOR3D;
 function Vector_subtract(v1,v2 : VECTOR3D) : VECTOR3D;
 function Vector_multiply(v1 : VECTOR3D; scale : double ): VECTOR3D;

 function Vector_Making(x, y ,z : single):VECTOR3D;

implementation

function Vector_Making(x, y ,z : single):VECTOR3D;
begin
    Vector_Making.x := x;
    Vector_Making.y := y;
    Vector_Making.z := z;
end;

function Vertexf(x0, y0 ,z0,
                 x1, y1 ,z1,
                 x2, y2 ,z2 : single): V3D;overload;
var
   res : V3D;
begin
  // Utility function to build a vectors:

  setLength(res,3);

  res[0].x := x0;
```

```
  res[0].y := y0;
  res[0].z := z0;

  res[1].x := x1;
  res[1].y := y1;
  res[1].z := z1;

  res[2].x := x2;
  res[2].y := y2;
  res[2].z := z2;

  Vertexf := res;
end;


function Vertexf(x0, y0 ,z0,
                 x1, y1 ,z1,
                 x2, y2 ,z2,
                 x3, y3 ,z3 : single): V3D;overload;
var
   res : V3D;
begin
  // Utility function to build a vectors:

  setLength(res,4);


  res[0].x := x0;
  res[0].y := y0;
  res[0].z := z0;

  res[1].x := x1;
  res[1].y := y1;
  res[1].z := z1;

  res[2].x := x2;
  res[2].y := y2;
  res[2].z := z2;

  res[3].x := x3;
  res[3].y := y3;
  res[3].z := z3;

  Vertexf := res;
end;


// The most important part of this tutorial
//is the plane equation, Ax + By + Cz + D = 0.

////////////////////// CROSS \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns a perpendicular vector from 2 given
/////   vectors by taking the cross product.
////////////////////// CROSS \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function Cross( vVector1,  vVector2 : VECTOR3D):VECTOR3D;
var
  // The vector to hold the cross product
```

```
  vNormal : VECTOR3D ;
begin
  // Once again, if we are given 2 vectors
  // (directions of 2 sides of a polygon)
  // then we have a plane define.  The cross
  // product finds a vector that is perpendicular
  // to that plane, which means it's point straight
  // out of the plane at a 90 degree angle.

  // The X value for the vector is:  (V1.y * V2.z) - (V1.z * V2.y)
                                                              // Get the X value
  vNormal.x := ((vVector1.y * vVector2.z) - (vVector1.z * vVector2.y));

  // The Y value for the vector is:  (V1.z * V2.x) - (V1.x * V2.z)
  vNormal.y := ((vVector1.z * vVector2.x) - (vVector1.x * vVector2.z));

  // The Z value for the vector is:  (V1.x * V2.y) - (V1.y * V2.x)
  vNormal.z := ((vVector1.x * vVector2.y) - (vVector1.y * vVector2.x));

  Cross := vNormal;       // Return the cross product (Direction the polygon is facing - Normal)
end;

//////////////////////// VECTOR \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns a vector between 2 points
//////////////////////// VECTOR \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function Vector( vPoint1,  vPoint2 : VECTOR3D):VECTOR3D;
var
  vVector : VECTOR3D;
begin
   // In order to get a vector from 2 points (a direction) we need to
   // subtract the second point from the first point.

   vVector.x := vPoint1.x - vPoint2.x;  // Get the X value of our new vector
   vVector.y := vPoint1.y - vPoint2.y;  // Get the Y value of our new vector
   vVector.z := vPoint1.z - vPoint2.z;  // Get the Z value of our new vector

   Vector := vVector;                              // Return our new vector
end;

//////////////////// MAGNITUDE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns the magnitude of a normal (or any other vector)
///////////////////// MAGNITUDE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function Magnitude( vNormal :VECTOR3D):single;
begin
 // This will give us the magnitude or "Norm" as some say, of our normal.
 // Here is the equation:
 //  magnitude = sqrt(V.x^2 + V.y^2 + V.z^2)  Where V is the vector

          Magnitude := sqrt( (vNormal.x * vNormal.x) +
                       (vNormal.y * vNormal.y) +
                       (vNormal.z * vNormal.z) );
end;

///////////////////// NORMALIZE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns a normalize vector (A vector exactly of length 1)
////////////////////// NORMALIZE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
```

```
function Normalize( vNormal : VECTOR3D ):VECTOR3D;
var
   m_magnitude : single;
begin
        // Get the magnitude of our normal
           m_magnitude := Magnitude(vNormal);

          // Now that we have the magnitude, we can
       // divide our normal by that magnitude.
          // That will make our normal a total length of 1.
       // This makes it easier to work with too.

       // Divide the X value of our normal by it's magnitude
          vNormal.x :=vNormal.x/ m_magnitude;
       // Divide the Y value of our normal by it's magnitude
          vNormal.y :=vNormal.y/ m_magnitude;
       // Divide the Z value of our normal by it's magnitude
          vNormal.z :=vNormal.z/ m_magnitude;

          // Finally, return our normalized normal.

          Normalize := vNormal;
          // Return the new normal of length 1.
end;

///////////////////// NORMAL \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns the normal of a polygon
////    (The direction the polygon is facing)
////////////////////// NORMAL \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function Normal( vTriangle : V3D ):VECTOR3D;
var
  vVector1 , vVector2 , vNormal : VECTOR3D;

begin
   // Get 2 vectors from the polygon (2 sides), Remember the order!
   vVector1 := Vector(vTriangle[2], vTriangle[0]);
   vVector2 := Vector(vTriangle[1], vTriangle[0]);
   // Take the cross product of our 2 vectors
   // to get a perpendicular vector
   vNormal  := Cross(vVector1, vVector2);

   // Now we have a normal, but it's at a
   // strange length, so let's make it length 1.
   // Use our function we created to normalize
   // the normal (Makes it a length of one)
   vNormal := Normalize(vNormal);

   Normal := vNormal; // Return our normal at our desired length
end;

///////////////// PLANE DISTANCE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns the distance between a plane and the origin
///////////////// PLANE DISTANCE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function PlaneDistance( Normal, Point :VECTOR3D): single;
 var
```

```
    distance : single ;
   // This variable holds the distance
   // from the plane tot he origin
begin

         //  D = -(Ax + By + Cz)

                  // Basically, the negated dot product of the normal
      //of the plane and the point. (More about the dot product in another tutorial)
         distance := - ((Normal.x * Point.x) +
                  (Normal.y * Point.y) +
                  (Normal.z * Point.z));

         PlaneDistance := distance;
         // Return the distance
end;

///////////// INTERSECTED PLANE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This checks to see if a line intersects a plane
///////////// INTERSECTED PLANE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function IntersectedPlane( vTriangle , vLine : V3D ):boolean;
var
   distance1, distance2 , originDistance : single ;
   vNormal : VECTOR3D;
begin

      // We need to get the normal of our plane to go any further
         vNormal := Normal(vTriangle);

         originDistance := PlaneDistance(vNormal, vTriangle[0]);

         // Get the distance from point1 from the plane using:
      // Ax + By + Cz + D = (The distance from the plane)

         distance1 := ((vNormal.x * vLine[0].x)  +  // Ax +
                     (vNormal.y * vLine[0].y)  +  // Bx +
                     (vNormal.z * vLine[0].z)) + originDistance; // Cz + D

         distance2 := ((vNormal.x * vLine[1].x)  + // Ax +
                     (vNormal.y * vLine[1].y)  + // Bx +
                     (vNormal.z * vLine[1].z)) + originDistance; // Cz + D
 // Check to see if both point's distances are both negative or both positive
         if(distance1 * distance2 >= 0)   then
      // Return false if each point has the same sign.
      // -1 and 1 would mean each point is on either side of the plane.
      // -1 -2 or 3 4 wouldn't...
           IntersectedPlane := false
      else
         // The line intersected the plane, Return TRUE
         IntersectedPlane := true;
end;

//////////////// INTERSECTED PLANE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This checks to see if a line intersects a plane
//////////////// INTERSECTED PLANE \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function IntersectedPlane02( vPoly, vLine : V3D;
```

```
                        var vNormal : VECTOR3D ;
                        var originDistance : single):boolean;
var
   distance1, distance2 : single ;
begin
      // We need to get the normal of our plane to go any further
         vNormal := Normal(vPoly);

         originDistance := PlaneDistance(vNormal, vPoly[0]);

         distance1 := ((vNormal.x * vLine[0].x)  + // Ax +
                        (vNormal.y * vLine[0].y)  + // Bx +
                        (vNormal.z * vLine[0].z)) + originDistance; // Cz + D

         distance2 := ((vNormal.x * vLine[1].x)  + // Ax +
                        (vNormal.y * vLine[1].y)  + // Bx +
                        (vNormal.z * vLine[1].z)) + originDistance;// Cz + D

// Check to see if both point's distances are both negative or both positive
         if(distance1 * distance2 >= 0) then
       // Return false if each point has the same sign.
       //  -1 and 1 would mean each point is on either side of the plane.
       //  -1 -2 or 3 4 wouldn't...
           IntersectedPlane02 := false
      else
        // The line intersected the plane, Return TRUE
            IntersectedPlane02 := true;

end;


//////////////////////////////// DOT \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////      This computers the dot product of 2 vectors
///////////////////////////////// DOT \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function Dot( vVector1, vVector2 :VECTOR3D ):single;
begin
  // The dot product is this equation:
  // V1.V2 = (V1.x * V2.x  +  V1.y * V2.y  +  V1.z * V2.z)
  // In math terms, it looks like this:  V1.V2 = ||V1|| ||V2|| cos(theta)
  // The '.' means DOT.   The || || is magnitude.

         Dot := ( (vVector1.x * vVector2.x) +
              (vVector1.y * vVector2.y) +
              (vVector1.z * vVector2.z) );
end;


////////////// ANGLE BETWEEN VECTORS \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////      This checks to see if a point is inside the ranges of a polygon
////////////// ANGLE BETWEEN VECTORS \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function AngleBetweenVectors(Vector1, Vector2 :VECTOR3D ):double;
var
   dotProduct ,vectorsMagnitude : single;
begin
         // Get the dot product of the vectors
          dotProduct := Dot(Vector1, Vector2);
```

```
         // Get the product of both of the vectors magnitudes
          vectorsMagnitude := Magnitude(Vector1) * Magnitude(Vector2) ;

         // Return the arc cosine of the
      //(dotProduct / vectorsMagnitude) which is the angle in RADIANS.
         AngleBetweenVectors := ( arccos( dotProduct / vectorsMagnitude ) );
end;

////////////// INTERSECTION POINT \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This returns the intersection point
////    of the line that intersects the plane
/////////////// INTERSECTION POINT \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function IntersectionPoint(vNormal : VECTOR3D ; vLine : V3D ;
                     distance : double ):VECTOR3D;
var
   vPoint , vLineDir : VECTOR3D;
   Numerator , Denominator , dist : double ;
begin
         //  We need to find the 3D point that is actually
         // on the plane.  Here are some steps to do that:

         vLineDir := Vector(vLine[1], vLine[0]);    // Get the Vector of the line
         vLineDir := Normalize(vLineDir);           // Normalize the lines vector
      // Use the plane equation with the normal and the line
         Numerator := - (vNormal.x * vLine[0].x +
                        vNormal.y * vLine[0].y +
                        vNormal.z * vLine[0].z + distance);

 // Get the dot product of the line's vector and the normal of the plane
         Denominator := Dot(vNormal, vLineDir);

      // Check so we don't divide by zero
         if( Denominator = 0.0)    then
        begin
            // Return an arbitrary point on the line
                 IntersectionPoint := vLine[0];
            exit;
        end;

      // Divide to get the multiplying (percentage) factor
         dist := Numerator / Denominator;

         vPoint.x := (vLine[0].x + (vLineDir.x * dist));
         vPoint.y := (vLine[0].y + (vLineDir.y * dist));
         vPoint.z := (vLine[0].z + (vLineDir.z * dist));

         IntersectionPoint := vPoint;       // Return the intersection point
end;


/////////////////// INSIDE POLYGON \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////     This checks to see if a point is inside the ranges of a polygon
/////////////////// INSIDE POLYGON \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function InsidePolygon( vIntersection : VECTOR3D ; Poly :V3D ;
                   verticeCount : LongInt ): boolean;
```

```
var
   Angle : double;
   vA, vB : VECTOR3D; // Create temp vectors
   i : integer;
begin
        Angle := 0;
        // Go in a circle to each vertex and get the angle between
           for  i := 0 to verticeCount-1 do
           begin
               // Subtract the intersection point from the current vertex
                     vA := Vector(Poly[i], vIntersection);
                     // Subtract the point from the next vertex
                     vB := Vector(Poly[(i + 1) mod verticeCount], vIntersection);
// Find the angle between the 2 vectors and add them all up as we go along
                     Angle := Angle+ AngleBetweenVectors(vA, vB);
           end;

        // If the angle is greater than 2 PI, (360 degrees)
           if(Angle >= (MATCH_FACTOR * (2.0 * PI)) )  then
                     InsidePolygon := TRUE  // The point is inside of the polygon
        else
                  InsidePolygon := FALSE;    // If you get here, it obviously
                  // wasn't inside the polygon, so Return FALSE
end;


//////////////// INTERSECTED POLYGON \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*
/////      This checks if a line is intersecting a polygon
///////////////// INTERSECTED POLYGON \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*

function IntersectedPolygon( vPoly, vLine :V3D ;
                         verticeCount:integer ):boolean;
var
  vNormal , vIntersection : VECTOR3D;
  originDistance : single ;
begin
           if(not(IntersectedPlane02(vPoly, vLine,
              vNormal,    originDistance))) then
          begin
                     IntersectedPolygon := false;
               exit;
          end;

           vIntersection := IntersectionPoint(vNormal,
                                    vLine, originDistance);

           if(InsidePolygon(vIntersection, vPoly, verticeCount)) then
          begin
                     IntersectedPolygon := true;
               exit;
          end;

           // If we get here, we must have NOT collided
           IntersectedPolygon := false;       // There was no collision, so return false
end;


function Matrix_making():TMatrix33;overload;
```

```
var
  res : TMatrix33;
begin
         res.Mx[0][0]:=1.0; res.Mx[0][1]:=0.0; res.Mx[0][2]:=0.0;

         res.Mx[1][0]:=0.0; res.Mx[1][1]:=1.0; res.Mx[1][2]:=0.0;
         res.Mx[2][0]:=0.0; res.Mx[2][1]:=0.0; res.Mx[2][2]:=1.0;
      Matrix_making := res;
end;


function Matrix_making( mx00, mx01, mx02,

                         mx10, mx11, mx12,
                         mx20, mx21, mx22 :double ): TMatrix33;overload;
var
  res : TMatrix33;
begin
         res.Mx[0][0]:=mx00; res.Mx[0][1]:=mx01; res.Mx[0][2]:=mx02;

         res.Mx[1][0]:=mx10; res.Mx[1][1]:=mx11; res.Mx[1][2]:=mx12;
         res.Mx[2][0]:=mx20; res.Mx[2][1]:=mx21; res.Mx[2][2]:=mx22;
      Matrix_making := res;
end;

function Matrix_making( Phi, Theta, Psi :double) : TMatrix33;overload;

  var

    c1 ,s1 , c2, s2, c3, s3 :double;

    res : TMatrix33;

begin
          c1:=cos(Phi);   s1:=sin(Phi);    c2:=cos(Theta);

       s2:=sin(Theta); c3:=cos(Psi);    s3:=sin(Psi);

         res.Mx[0][0]:=c2*c3;

         res.Mx[0][1]:=-c2*s3;
         res.Mx[0][2]:=s2;
         res.Mx[1][0]:=s1*s2*c3+c1*s3;
         res.Mx[1][1]:=-s1*s2*s3+c1*c3;
         res.Mx[1][2]:=-s1*c2;
         res.Mx[2][0]:=-c1*s2*c3+s1*s3;
         res.Mx[2][1]:=c1*s2*s3+s1*c3;
         res.Mx[2][2]:=c1*c2;

      Matrix_making := res;

end;


function Matrix_add(m1, m2 :TMatrix33 ):TMatrix33;
var
   res : TMatrix33;
begin
```

```
        res.Mx[0][0] := m1.Mx[0][0] + m2.Mx[0][0];
        res.Mx[0][1] := m1.Mx[0][1] + m2.Mx[0][1];
        res.Mx[0][2] := m1.Mx[0][2] + m2.Mx[0][2];
        res.Mx[1][0] := m1.Mx[1][0] + m2.Mx[1][0];
        res.Mx[1][1] := m1.Mx[1][1] + m2.Mx[1][1];
        res.Mx[1][2] := m1.Mx[1][2] + m2.Mx[1][2];
        res.Mx[2][0] := m1.Mx[2][0] + m2.Mx[2][0];
        res.Mx[2][1] := m1.Mx[2][1] + m2.Mx[2][1];
        res.Mx[2][2] := m1.Mx[2][2] + m2.Mx[2][2];

        Matrix_add := res;

end;

function Matrix_subtract(m1,m2 :TMatrix33 ):TMatrix33;
var
   res : TMatrix33;
begin
        res.Mx[0][0] := m1.Mx[0][0] - m2.Mx[0][0];

        res.Mx[0][1] := m1.Mx[0][1] - m2.Mx[0][1];
        res.Mx[0][2] := m1.Mx[0][2] - m2.Mx[0][2];
        res.Mx[1][0] := m1.Mx[1][0] - m2.Mx[1][0];
        res.Mx[1][1] := m1.Mx[1][1] - m2.Mx[1][1];
        res.Mx[1][2] := m1.Mx[1][2] - m2.Mx[1][2];
        res.Mx[2][0] := m1.Mx[2][0] - m2.Mx[2][0];
        res.Mx[2][1] := m1.Mx[2][1] - m2.Mx[2][1];
        res.Mx[2][2] := m1.Mx[2][2] - m2.Mx[2][2];

        Matrix_subtract := res;

end;

function Matrix_multiply( m1 , m2 : TMatrix33):TMatrix33;overload;
var
   res : TMatrix33;
begin
        res.Mx[0][0] := m1.Mx[0][0]*m2.Mx[0][0] +
                m1.Mx[0][1]*m2.Mx[1][0] +
                m1.Mx[0][2]*m2.Mx[2][0];
        res.Mx[1][0] := m1.Mx[1][0]*m2.Mx[0][0] +
                m1.Mx[1][1]*m2.Mx[1][0] +
                m1.Mx[1][2]*m2.Mx[2][0];
        res.Mx[2][0] := m1.Mx[2][0]*m2.Mx[0][0] +
                m1.Mx[2][1]*m2.Mx[1][0] +
                m1.Mx[2][2]*m2.Mx[2][0];
        res.Mx[0][1] := m1.Mx[0][0]*m2.Mx[0][1] +
                m1.Mx[0][1]*m2.Mx[1][1] +
                m1.Mx[0][2]*m2.Mx[2][1];
        res.Mx[1][1] := m1.Mx[1][0]*m2.Mx[0][1] +
                m1.Mx[1][1]*m2.Mx[1][1] +
                m1.Mx[1][2]*m2.Mx[2][1];
        res.Mx[2][1] := m1.Mx[2][0]*m2.Mx[0][1] +
                m1.Mx[2][1]*m2.Mx[1][1] +
                m1.Mx[2][2]*m2.Mx[2][1];
        res.Mx[0][2] := m1.Mx[0][0]*m2.Mx[0][2] +
                m1.Mx[0][1]*m2.Mx[1][2] +
                m1.Mx[0][2]*m2.Mx[2][2];
```

```
        res.Mx[1][2] := m1.Mx[1][0]*m2.Mx[0][2] +
                  m1.Mx[1][1]*m2.Mx[1][2] +
                  m1.Mx[1][2]*m2.Mx[2][2];
        res.Mx[2][2] := m1.Mx[2][0]*m2.Mx[0][2] +
                  m1.Mx[2][1]*m2.Mx[1][2] +
                  m1.Mx[2][2]*m2.Mx[2][2];
        Matrix_multiply := res;
end;


function Matrix_multiply(
          m1 :TMatrix33; scale : double):TMatrix33;overload;
var
  res : TMatrix33;

begin
        res.Mx[0][0] := m1.Mx[0][0] * scale;

        res.Mx[0][1] := m1.Mx[0][1] * scale;
        res.Mx[0][2] := m1.Mx[0][2] * scale;
        res.Mx[1][0] := m1.Mx[1][0] * scale;
        res.Mx[1][1] := m1.Mx[1][1] * scale;
        res.Mx[1][2] := m1.Mx[1][2] * scale;
        res.Mx[2][0] := m1.Mx[2][0] * scale;
        res.Mx[2][1] := m1.Mx[2][1] * scale;
        res.Mx[2][2] := m1.Mx[2][2] * scale;

        Matrix_multiply := res;

end;

function Matrix_multiply(
             m1 : TMatrix33; v :VECTOR3D):VECTOR3D;overload;
var
  res : VECTOR3D;
begin


  res.x := m1.Mx[0][0]*v.X + m1.Mx[0][1]*v.Y + m1.Mx[0][2]*v.Z;

  res.y := m1.Mx[1][0]*v.X + m1.Mx[1][1]*v.Y + m1.Mx[1][2]*v.Z;
  res.z := m1.Mx[2][0]*v.X + m1.Mx[2][1]*v.Y + m1.Mx[2][2]*v.Z;

  Matrix_multiply := res;

end;

function Matrix_determinant(m : TMatrix33):double;
begin
   Matrix_determinant :=

     m.Mx[0][0]*(m.Mx[1][1]*m.Mx[2][2]-m.Mx[1][2]*m.Mx[2][1])

     - m.Mx[0][1]*(m.Mx[1][0]*m.Mx[2][2]-m.Mx[1][2]*m.Mx[2][0])
     + m.Mx[0][2]*(m.Mx[1][0]*m.Mx[2][1]-m.Mx[1][1]*m.Mx[2][0]);
end;

function Matrix_transpose(m : TMatrix33):TMatrix33;
```

```
var
   t : double;
begin
        t:= m.Mx[0][2]; m.Mx[0][2] := m.Mx[2][0]; m.Mx[2][0] := t;

        t := m.Mx[0][1]; m.Mx[0][1] := m.Mx[1][0]; m.Mx[1][0] := t;

        t := m.Mx[1][2]; m.Mx[1][2] := m.Mx[2][1]; m.Mx[2][1] := t;

        Matrix_transpose :=m;

end;


function Matrix_inverse(m1 : TMatrix33 ):TMatrix33;

var

   det : double;

   res : TMatrix33;

begin
  det := Matrix_determinant(m1);

 res.Mx[0][0] := m1.Mx[1][1]*m1.Mx[2][2] - m1.Mx[1][2]*m1.Mx[2][1];

 res.Mx[0][1] := m1.Mx[2][1]*m1.Mx[0][2] - m1.Mx[2][2]*m1.Mx[0][1];
 res.Mx[0][2] := m1.Mx[0][1]*m1.Mx[1][2] - m1.Mx[0][2]*m1.Mx[1][1];
 res.Mx[1][0] := m1.Mx[1][2]*m1.Mx[2][0] - m1.Mx[1][0]*m1.Mx[2][2];
 res.Mx[1][1] := m1.Mx[2][2]*m1.Mx[0][0] - m1.Mx[2][0]*m1.Mx[0][2];
 res.Mx[1][2] := m1.Mx[0][2]*m1.Mx[1][0] - m1.Mx[0][0]*m1.Mx[1][2];
 res.Mx[2][0] := m1.Mx[1][0]*m1.Mx[2][1] - m1.Mx[1][1]*m1.Mx[2][0];
 res.Mx[2][1] := m1.Mx[2][0]*m1.Mx[0][1] - m1.Mx[2][1]*m1.Mx[0][0];
 res.Mx[2][2] := m1.Mx[0][0]*m1.Mx[1][1] - m1.Mx[0][1]*m1.Mx[1][0];

 Matrix_inverse := Matrix_multiply(res, 1.0/det);

end;


function Vector_add( v1 , v2 : VECTOR3D) : VECTOR3D;

var
  res : VECTOR3D;
begin
                res.x := v1.x + v2.x;

                res.y := v1.y + v2.y;
                res.z := v1.z + v2.z;
      Vector_add := res;
end;

function Vector_subtract(v1,v2 : VECTOR3D) : VECTOR3D;

var
  res : VECTOR3D;
begin
```

```
                res.x := v1.x - v2.x;

                res.y := v1.y - v2.y;
                res.z := v1.z - v2.z;
   Vector_subtract := res;
end;


function Vector_multiply(v1 : VECTOR3D; scale : double ): VECTOR3D;

var
  res : VECTOR3D;
begin
                res.x := v1.x * scale;

                res.y := v1.y * scale;
                res.z := v1.z * scale;
      Vector_multiply := res;
end;


end.
```

و اما اصل برنامه این فصل به شرح زیر می باشد :

```
unit ch25;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs , OpenGL , SPF , Math_3D;

type
  TForm1 = class(TForm)
    procedure FormKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```
var

 vTriangle : V3D ;
// This is our line that we will be checking
//against the polygon's plane for collision.
// We position the line going directly through the polygon at first.
 vLine : V3D;

 f_Hdc : longInt;

procedure initGL();
begin
 vTriangle := Math_3D.Vertexf(-1,0,0,  0,1,0,   1,0,0);
 vLine :=  Math_3D.Vertexf(0,0.5,-0.5,  0,0.5,0.5 ,0,0,0);
end;

procedure RenderScene( m_mode : integer );
var
   bCollided : boolean;
begin
       bCollided := false;

       // Clear The Screen And The Depth Buffer
       glClearColor(1,1,1,0);
          glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
          glLoadIdentity();   // Reset The matrix

          // Let's set our camera to the left a bit for a better view
       // This determines where the camera's position and view is
          gluLookAt(-2.5, 0.5, 0.5,  0, 0.5, 0,   0, 1, 0);

          // Below we give OpenGL the 3 vertices of
       //our triangle.  Once again, we put them
          // into an array of VECTOR3D structures so
       //we could dynamically move it around screen.

          glBegin (GL_TRIANGLES);          // This is our BEGIN to draw
             glColor3ub(255, 0, 0);
             glVertex3f(vTriangle[0].x, vTriangle[0].y, vTriangle[0].z);

             glColor3ub(255, 255, 0);
             glVertex3f(vTriangle[1].x, vTriangle[1].y, vTriangle[1].z);

             glColor3ub(0, 255, 255);
             glVertex3f(vTriangle[2].x, vTriangle[2].y, vTriangle[2].z);
          glEnd();     // This is the END of drawing

     if m_mode=0 then
       begin
// Below we use our function we just wrote to see if the plane of the
// triangle and the line intersect. It will return true if that's the case.

        bCollided := IntersectedPlane(vTriangle, vLine);
       end;

     if m_mode=1 then
       begin
```

```
// Now, instead of just testing against the plane, we take it a step further
// and test if we actually hit the polygon.  This is a more usable collision.
// We give our function the polygon, the line to test with, and the
// number of vertices of our polygon

          bCollided := IntersectedPolygon(vTriangle, vLine, 3);
     end;

// Below we draw the line that the polygon will be colliding with.
// We will check to see if the line collides with the polygons
// plane, and if it does,
// we will turn the line green to show when it is intersecting the plane.

          glBegin (GL_LINES);      // This is our BEGIN to draw
    // If we collided, change the color of the line to illustrate this.
                    if(bCollided) then
        // Make the line RED if we collided with the triangle's plane
                              glColor3ub(255, 0, 0)
                    else
                     // Make the line blue if we didn't collide
                              glColor3ub(0, 0, 25);

              // Let's draw the normal centered on the triangle
                    glVertex3fv(@vLine[0]);
             // Draw the normal of the polygon from the
             //center of the polygon to better see it
                    glVertex3fv(@vLine[1]);
          glEnd();   // This is the END of drawing

          // That's it, now use the LEFT and RIGHT arrow keys to
       //move it around to further see it in action.

          SwapBuffers(f_Hdc);// Swap the backbuffers to the foreground

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 f_Hdc := GetDC( form1.handle );
 SetDCPixelFormat(f_Hdc,16,16);// Create a rendering context.
 initGL();
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  wglMakeCurrent(f_Hdc,hrc); //activate the RC

          if (height=0) then  // Prevent A Divide By Zero error
          begin
                    height:=1;  // Make the Height Equal One
          end;

          glViewport(0,0,width,height);// Make our viewport the whole window

          glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
          glLoadIdentity();   // Reset The Projection Matrix

          gluPerspective(45.0,width/height,0.1,150.0);
```

```
        glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
        glLoadIdentity();               // Reset The Modelview Matrix

  InvalidateRect(Handle, nil, False);// Draw the scene.

end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 CleanUp(f_Hdc);// Clean up and terminate.
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
 wglMakeCurrent(f_Hdc,hrc); //activate the RC
  RenderScene(1);           // Draw the scene
end;


procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
    case key of
          VK_ESCAPE: // Check if we hit the ESCAPE key.
            PostQuitMessage(0);   // Tell windows we want to quit

          VK_UP:  // Check if we hit the UP ARROW key.
             begin
               // Move the left point of the triangle to the left
                     vTriangle[0].x :=vTriangle[0].x + 0.01;
               // Move the top point of the triangle to the left
                     vTriangle[1].x := vTriangle[1].x + 0.01;
               // Move the right point of the triangle to the left
                     vTriangle[2].x :=vTriangle[2].x + 0.01;
               // Redraw the scene to reflect the new position
               RenderScene(1);
              end;
         VK_DOWN:  // Check if we hit the DOWN ARROW key.
              begin
                     vTriangle[0].x :=vTriangle[0].x - 0.01;
                     vTriangle[1].x :=vTriangle[1].x - 0.01;
                     vTriangle[2].x :=vTriangle[2].x - 0.01;
                     RenderScene(1);
              end;
         VK_LEFT: // Check if we hit the LEFT ARROW key.
             begin
                    vTriangle[0].z :=vTriangle[0].z - 0.01;
                    vTriangle[1].z :=vTriangle[1].z - 0.01;
                    vTriangle[2].z :=vTriangle[2].z - 0.01;
                    RenderScene(1);
             end;
         VK_RIGHT:// Check if we hit the RIGHT ARROW key.
             begin
                    vTriangle[0].z :=vTriangle[0].z + 0.01;
                    vTriangle[1].z :=vTriangle[1].z + 0.01;
                    vTriangle[2].z :=vTriangle[2].z + 0.01;
              RenderScene(1);
```

```
            end;
          VK_PRIOR:// Check if we hit the PAGE UP key.
             begin
                   vTriangle[0].y := vTriangle[0].y+ 0.01;
                   vTriangle[1].y := vTriangle[1].y + 0.01;
                   vTriangle[2].y := vTriangle[2].y +0.01;
                   RenderScene(1);
             end;
         VK_NEXT:  // Check if we hit the PAGE DOWN key.
            begin
                   vTriangle[0].y := vTriangle[0].y - 0.01;
                   vTriangle[1].y := vTriangle[1].y - 0.01;
                   vTriangle[2].y := vTriangle[2].y - 0.01;
                   RenderScene(1);
            end;
   end;

end;


end.
```