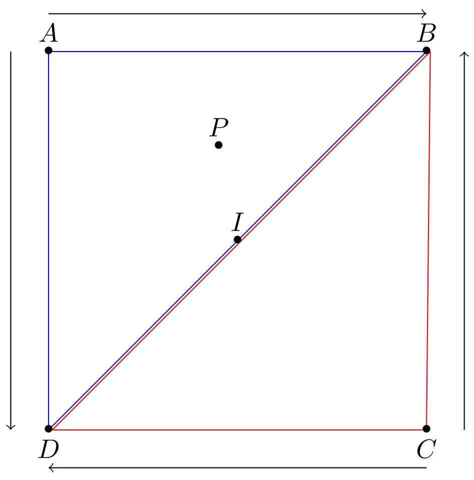
# Distance from a quadrilateral to a point

Benjamin Loison

22 decembre 2018

## Contents

## 1 Opening speech



We work with four points $A$, $B$, $C$, $D$ and $P$ which we know the components (x, y, z).
And we will have a lot of work on $P$ which is the point where is the player.
The aim of these algorithms is to compute the distance between the player and the ground under him. To solve this issue we are going to slice the quadrilateral in two, in order to get two triangles: $ABD$ and $BCD$.
Likewise we have one point and two vectors for each triangle, so we can define a unique plane.

## 2 Computing of a normal vector to a triangle

Let's first work on the $ABD$ triangle, the $BCD$ will have quite a same process.
We have:
$A, B, C(x_K, y_K, z_K)$
And more especially:

$$\left\{ \begin{array}{l} A(x_A, y_A, z_A) \\ \overrightarrow{AB} = (x_B - x_A, y_B - y_A, z_B - z_A) \\ \overrightarrow{AD} = (x_D - x_A, y_D - y_A, z_D - z_A) \end{array} \right.$$

Let's set $\vec{n}$ a normal vector: $(\vec{n}_x, \vec{n}_y, \vec{n}_z)$
We have:

$$\begin{cases} \overrightarrow{AB}.\vec{n} = 0 \\ \overrightarrow{AD}.\vec{n} = 0 \end{cases}$$

Notice: $\vec{a}.\vec{b} = \vec{a}_x * \vec{b}_x + \vec{a}_y * \vec{b}_y + \vec{a}_z * \vec{b}_z$

Which is equivalent to:

$$\begin{cases} \overrightarrow{AB}_x * \vec{n}_x + \overrightarrow{AB}_y * \vec{n}_y + \overrightarrow{AB}_z * \vec{n}_z = 0 \\ \overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_y * \vec{n}_y + \overrightarrow{AD}_z * \vec{n}_z = 0 \end{cases}$$

So two equations for three unknows, this is normal because we will fix one of this unknown (it will correspond as something like fixing the length of $\vec{n}$). This indeed depends on $\vec{n}_z$, which will be fixed to 1 (and not 0 to escape $\vec{0}$) and it simplify a final calculus.

$$\begin{cases} \overrightarrow{AB}_x * \vec{n}_x + \overrightarrow{AB}_y * \vec{n}_y + \overrightarrow{AB}_z = 0 \\ \overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_y * \vec{n}_y + \overrightarrow{AD}_z = 0 \end{cases}$$

## 2.1 First method with substitution

Important: this method here doesn't take care about hypothesis on denominator so please switch to Cramer's method

So:

$$\begin{cases} \vec{n}_x = -\frac{\overrightarrow{AB}_y * \vec{n}_y + \overrightarrow{AB}_z}{\overrightarrow{AB}_x} \\ \vec{n}_y = -\frac{\overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_z}{\overrightarrow{AD}_y} \end{cases}$$

Finally:

$$\vec{n}_x = \frac{1}{\overrightarrow{AB}_x} * \left( \frac{\overrightarrow{AB}_y * (\overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_z)}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z \right)$$

Or:

$$\vec{n}_x = \frac{1}{\overrightarrow{AB}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right) + \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_x * \vec{n}_x}{\overrightarrow{AB}_x * \overrightarrow{AD}_y}$$

So:

$$\vec{n}_x - \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_x * \vec{n}_x}{\overrightarrow{AB}_x * \overrightarrow{AD}_y} = \frac{1}{\overrightarrow{AB}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right)$$

$$\vec{n}_x * \left(1 - \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_x}{\overrightarrow{AB}_x * \overrightarrow{AD}_y}\right) = \frac{1}{\overrightarrow{AB}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right)$$

$$\vec{n}_x * \left(\frac{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x}{\overrightarrow{AB}_x * \overrightarrow{AD}_y}\right) = \frac{1}{\overrightarrow{AB}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right)$$

$$\vec{n}_x = \frac{\overrightarrow{AB}_x * \overrightarrow{AD}_y}{(\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x) * \overrightarrow{AB}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right)$$

$$\vec{n}_x = \frac{\overrightarrow{AD}_y}{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z}{\overrightarrow{AD}_y} - \overrightarrow{AB}_z\right)$$

As a result, we have:

$$\vec{n}_x = \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z - \overrightarrow{AD}_y * \overrightarrow{AB}_z}{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x}$$

To sum up:

$$\vec{n}\left(\frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z - \overrightarrow{AD}_y * \overrightarrow{AB}_z}{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x}, -\frac{\overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_z}{\overrightarrow{AD}_y}, 1\right)$$

Or, if we assume that: $\vec{n}_y = -\frac{\overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_z}{\overrightarrow{AD}_y}$

$$\vec{n}\left(-\frac{\overrightarrow{AB}_y * \vec{n}_y + \overrightarrow{AB}_z * \vec{n}_z}{\overrightarrow{AB}_x}, \vec{n}_y, 1\right)$$

Notice: for the algorithm part, it is important to remind us that for optimization, we need to firstly compute $\vec{n}_x$ before $\vec{n}_y$ (to use it to compute $\vec{n}_y$ of course).

Listing 1: Here is the implementation in C++ to compute $\vec{n}$

```cpp
Vector3D normalVectorOfTriangle(Vector3D AB, Vector3D AD)
{
    double Nx = (AB.y * AD.z - AD.y * AB.z) / (AB.x * AD.y - AB.y * AD.x);
    double Ny = -(AD.x * Nx + AD.z) / AD.y;
    return Vector3D(Nx, Ny, 1);
}
```

### 2.2 Second method with Cramer's determinant

Slackness... (hope the first method works)
Let's write a few lines to check several things:
We got:

$$\begin{cases} \overrightarrow{AB}_x * \vec{n}_x + \overrightarrow{AB}_y * \vec{n}_y = -\overrightarrow{AB}_z \\ \overrightarrow{AD}_x * \vec{n}_x + \overrightarrow{AD}_y * \vec{n}_y = -\overrightarrow{AD}_z \end{cases}$$

Let's compute the determinant of this system:

$$\Delta = \overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x$$

$\Delta = 0$ if, and only if $\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x = 0$ if, and only if $\overrightarrow{AB}_x * \overrightarrow{AD}_y = \overrightarrow{AB}_y * \overrightarrow{AD}_x$. If so there is an infinity of solution or there is no solution (depends respectively for a graphical interpretation, if lines are superimposed or not).
If there is no solution, this is the end and if there is an infinity it seems impossible because it means that for two random direction, we will go up of 1.

Let's assume $\Delta \neq 0$, the pair solution is:

$$\begin{cases} \vec{n}_x = \dfrac{\begin{vmatrix} -\overrightarrow{AB}_z & \overrightarrow{AB}_y \\ -\overrightarrow{AD}_z & \overrightarrow{AD}_y \end{vmatrix}}{\Delta} \\ \\ \vec{n}_y = \dfrac{\begin{vmatrix} \overrightarrow{AB}_x & -\overrightarrow{AB}_z \\ \overrightarrow{AD}_x & -\overrightarrow{AD}_z \end{vmatrix}}{\Delta} \end{cases}$$

Which is equivalent to:

$$\begin{cases} \vec{n}_x = \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z - \overrightarrow{AB}_z * \overrightarrow{AD}_y}{\Delta} \\ \\ \vec{n}_y = \frac{\overrightarrow{AB}_z * \overrightarrow{AD}_x - \overrightarrow{AB}_x * \overrightarrow{AD}_z}{\Delta} \end{cases}$$

Finally:

$$\begin{cases} \vec{n}_x = \frac{\overrightarrow{AB}_y * \overrightarrow{AD}_z - \overrightarrow{AB}_z * \overrightarrow{AD}_y}{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x} \\ \\ \vec{n}_y = \frac{\overrightarrow{AB}_z * \overrightarrow{AD}_x - \overrightarrow{AB}_x * \overrightarrow{AD}_z}{\overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x} \end{cases}$$

Listing 2: Here is the implementation in C++ to compute $\vec{n}$

```cpp
Vector3D normalVectorOfTriangle(Vector3D AB, Vector3D AD)
{
    double determinant = AB.x * AD.y - AB.y * AD.x;
    if(determinant == 0)
    {
        cout << "A fatal error occured, map theory involves this impossibility
        (AB, AD): " << AB.x << " " << AB.y << " " << AB.z << " " << AD.x << " "
        << AD.y << " " << AD.z << endl;
        exit(2);
    }
    double Nx = (AB.y * AD.z - AB.z * AD.y) / determinant;
    double Ny = (AB.z * AD.x - AB.x * AD.z) / determinant;
    return Vector3D(Nx, Ny, 1);
}
```

## 3 Computing the altitude of the ground at the player coordinates

### 3.1 First method

Reminder: Important: this method here doesn't take care about hypothesis on denominator so please switch to Cramer's method

With the previous writing, we got the cartesian equation of a triangle plane:

$\vec{n}_x * x + \vec{n}_y * y + z + D = 0$ with $D$ a real constant

Or:

$D = -\vec{n}_x * x - \vec{n}_y * y - z$

We compute D with a point of the triangle, for instance: $A$

$D = -\vec{n}_x * x_A - \vec{n}_y * y_A - z_A$

Likewise the equation is:

$\vec{n}_x * x + \vec{n}_y * y + z - \vec{n}_x * x_A - \vec{n}_y * y_A - z_A = 0$

And we get for the altitude:

$z = -\vec{n}_x * x - \vec{n}_y * y + \vec{n}_x * x_A + \vec{n}_y * y_A + z_A$

Listing 3: Here is the implementation in C++ to compute the player altitude

```
class Vector3D
{
    public:
        double x, y, z;
        Vector3D(double x, double y, double z);
};

Vector3D::Vector3D(double x, double y, double z) : x(x), y(y), z(z) {}

/*A and P are here considered as points
and we don't need the third component for P*/
double groundAltitude(Vector3D A, Vector3D AB, Vector3D AD, Vector3D P)
{
    double Nx = (AB.y * AD.z - AD.y * AB.z) / (AB.x * AD.y - AB.y * AD.x);
    double Ny = -(AD.x * Nx + AD.z) / AD.y;
    double D = Nx * A.x + Ny * A.y + A.z;
    double altitude = - Nx * P.x - Ny * P.y + D;
    return altitude;
}

double groundAltitudePoints(Vector3D A, Vector3D B, Vector3D C, Vector3D P)
{
    Vector3D AB = Vector3D(B.x - A.x, B.y - A.y, B.z - A.z);
    Vector3D AD = Vector3D(C.x - A.x, C.y - A.y, C.z - A.z);
    return groundAltitude(A, AB, AD, P);
}

int main(int argc, char** argv)
{
    Vector3D A = Vector3D(0, 0, 0);
    Vector3D B = Vector3D(1, 0, 1);
    Vector3D D = Vector3D(0, 1, 1);
    Vector3D P = Vector3D(0.4, 0.25, -99); // the last component doesn't matter

    cout << groundAltitudePoints(A, B, D, P) << endl;
    while(true);
    return 0;
}
```

There is still one kind of problems: division by zero.
It happends if (or):

$$\left\{\begin{array}{r} \overrightarrow{AD}_y = 0 \\ \overrightarrow{AB}_x * \overrightarrow{AD}_y - \overrightarrow{AB}_y * \overrightarrow{AD}_x = 0 \end{array}\right.$$

Which is equivalent to:

$$\left\{\begin{array}{r} \overrightarrow{AD}_y = 0 \\ \overrightarrow{AB}_x * \overrightarrow{AD}_y = \overrightarrow{AB}_y * \overrightarrow{AD}_x \end{array}\right.$$

If $\overrightarrow{AD}_y = 0$ then $\overrightarrow{AB}_y * \overrightarrow{AD}_x = 0$ which means $\overrightarrow{AB}_y = 0$ or $\overrightarrow{AD}_x = 0$

Condition on the denominator has been forgotten...

### 3.2 Second method

Listing 4: Here is the implementation in C++ to compute the player altitude

```
class Vector3D
```

```cpp
{
    public:
        double x, y, z;
        Vector3D(double x, double y, double z);
};

Vector3D::Vector3D(double x, double y, double z) : x(x), y(y), z(z) {}

/*A and P are here considered as points
and we don't need the third component for P*/
double groundAltitude(Vector3D A, Vector3D AB, Vector3D AD, Vector3D P)
{
    double determinant = AB.x * AD.y - AB.y * AD.x;
    if(determinant == 0)
    {
        cout << "A fatal error occured, map theory involves this impossibility
        (AB, AD): " << AB.x << " " << AB.y << " " << AB.z << " " << AD.x << " "
        << AD.y << " " << AD.z << endl;
        exit(2);
    }
    double Nx = (AB.y * AD.z - AB.z * AD.y) / determinant;
    double Ny = (AB.z * AD.x - AB.x * AD.z) / determinant;
    double D = Nx * A.x + Ny * A.y + A.z;
    double altitude = - Nx * P.x - Ny * P.y + D;
    return altitude;
}

double groundAltitudePoints(Vector3D A, Vector3D B, Vector3D C, Vector3D P)
{
    Vector3D AB = Vector3D(B.x - A.x, B.y - A.y, B.z - A.z);
    Vector3D AD = Vector3D(C.x - A.x, C.y - A.y, C.z - A.z);
    return groundAltitude(A, AB, AD, P);
}

int main(int argc, char** argv)
{
    Vector3D A = Vector3D(0, 0, 0);
    Vector3D B = Vector3D(1, 0, 1);
    Vector3D D = Vector3D(0, 1, 1);
    Vector3D P = Vector3D(0.5, 0.5, -99); // the last component doesn't matter

    cout << groundAltitudePoints(A, B, D, P) << endl;
    while(true);
    return 0;
}
```