

برمجة الالعاب باستخدام الـ *DirectX* و

C++

الجزء الأول

2007

Mohammed alaa

هذه النسخه غير منقحه
و قد توجد بها أخطاء

لماذا تلك الكورسات أفضل من غيرها ؟

برمجة الألعاب تجمع بين ثنائياتها مجموعة من العلوم الأخرى و التى لابد و أن تستوعبها جيداً قبل البدء فى كتابة الكود. و هذا لا يحدث مع الاغلبية العظمى من المبرمجين الذين يطمحون فى دخول ذلك المجال. فتجد أنهم يبدؤا التعلم بأسلوب الممارسة، فيقوم هذا الشخص بالبحث عن أكواد و يبدأ فى فهمها ثم ينتقل لكود آخر و هكذا. و حيث أن هذا الأسلوب ينفج و لكن و بالرغم أنه يأخذ قدر كبير من الوقت و المجهود لفهم أكواد قد تكون غير مرتبة أو بها اخطاء إلا أيضاً انه قد تؤدي لهذا الشخص لكرهية ذلك المجال لعدم فهمه لمجموعة من الأكواد!

الكورسات التى أقوم بتقديمها مرتبة بحيث أن اخر مرحلة نقوم بها هيا كتابة الأكواد، و لعدم الفهم الخاطئ لابد منك أن تعرف ما هى تلك المراحل (يتم تطبيق تلك المراحل على كل كورس على حدى):

1. نبدأ فى معرفة ما العلم المتعلق بالكورس و شرحه شرحاً وافياً من جميع الجهات كنظريات و مفاهيم و علاقتها بالألعاب.

2. الخطوة الثانية هى البدء فى تطبيق هذا العلم باستخدام معادلات رياضياً و شرح كيفية استخدام تلك المعادلات (و كيف تستطيع تطويرها لتناسب إستخداماتك).

3. البدء فى تطبيق ما عرفنا من الخطوتين الأولى و الثانية داخل لغة برمجية معينة (داخل تلك الكورسات سنستخدم لغة الـ C++).

4. مشروع عملى تطبيقى على الخطوات الثلاثة السابقة.

كما تلاحظ فإنه عند وصولك للمرحلة الثالثة سيكون لديك القدرة فى تطبيق ما عرفتته باستخدام اللغة التى تفضلها، لأنك حينها ستكون لديك ما يؤهلك لذلك و سيكون لديك ما يجعلك تستطيع فيما بعد أن تطور نفسك بنفسك. لأنه حينها ستكون لديك مفاهيم و معارف أساسية.

لإى أستفسار أو معلومات عن الكورسات أرجو مراسلتى على :

Mohammed-alaa@hotmail.com

Mohammed1985@gmail.com

هذا الدرس غير مكتمل
الباقي سيرفع قريباً

الدرس الثانى : مبادئ الرسومات ثلاثية الأبعاد الجزء الثانى

3D Graphics Fundamentals Part2

إسم الفصل : الفصل الأول (الصفحات من 0 - 39)

أهداف هذا الدرس :

إبتداءً من آخر نقطة توقفنا عندها فى الدرس السابق ، سنبدأ فى التعامل مع المعادلات الرياضية و أنواع البيانات المختلفة التى سنستخدمها معا خلال هذا الكورس و التى ستؤثر على الاهداف المذكورة سابقاً و خصوصاً نظم التحويلات **The Transformation Pipeline** و التى سنتناولها بشكل أعمق ، سنقوم فى البداية بتعلم ثلاثة مفاهيم رياضية هامة هى : المتجهات **Vectors** و المسطحات **Planes** و أيضاً المصفوفات **Matrices** و سنعرف وظيفة كل منهم كما سنعرف معلومات أخرى عنهم ، سنتعلم أيضاً عمليات رياضية أخرى مثل الضرب القياسى **The Dot Product** ، الضرب الإتجاهى **The Cross Product** ، إتجاه أوجه الأسطح **Normalization** و أيضاً ضرب المصفوفات فى المتجهات **Matrix and Vector multiplication** سيتم شرحها بالتفصيل ، بعد ذلك سننظر للكائن **D3DX** و دواله التى يمكن أن نستخدمها لنقوم بكل العمليات المذكورة سابقاً ، وفى النهاية سنختم الدرس بشرح تحليلى لعمليات الرؤية من منظور طبيعى داخل الألعاب **Perspective Projection Operation** و كيف تنشئ مصفوفتها و أيضاً كيفية إنشاء مصفوفة الرؤية الطبيعية **Arbitrary Fields Of View** لتشكيل مجموعة مختلفة من الكاميرات المعرفة أو كاميرات جديدة.

العناصر الرئيسية فى هذا الدرس :

- أساسيات الرياضيات للمجسمات ثلاثية الأبعاد **3D Mathematics Primer**
 - 0 المتجهات **Vectors**
 - طول المتجهات **Magnitude**
 - جمع و طرح المتجهات **Addition/ Subtraction**
 - ضرب المتجهات فى الأعداد **Scalar Multiplication**
 - المسطحات **Normalization**
 - الضرب الإتجاهى **Cross Product**
 - الضرب القياسى **Dot Product**
 - 0 المسطحات **Planes**
 - 0 المصفوفات **Matrices**
 - ضرب المصفوفات فى المصفوفات **Matrix/Matrix Multiplication**
 - ضرب المصفوفات فى المتجهات **Vector/Matrix Multiplication**
 - تدوير المصفوفات الثابتة الأبعاد **3D Rotation Matrices**
 - مصفوفة الوحدة **Identity Matrices**
 - التحجيم الثابت و المتغير للمصفوفات **Scaling and Shearing**
 - دمج المصفوفات **Concatenation**
 - نظم الإحداثيات المحايدة **Homogenous Coordinates**

- رياضيات الكائن D3DX
 - o أنواع البيانات Data Types
 - الكائن D3DXMATRIX
 - الكائن D3DXVECTOR3
 - الكائن D3DXPLANE
 - الكائن D3DQUATERNION
 - o الدوال الخاصة بالتحويلات و المصفوفات Matrix and Transformation Functions
 - الدالة D3DXMatrixMultiply
 - الدالة D3DXMatrixRotation{XYZ}
 - الدالة D3DXMatrixTranslation
 - الدالة D3DXMatrixRotationYawPitchRoll
 - الدالة D3DXVecTransform{...}
 - o دوال المتجهات Vector Functions
 - الضرب الإتجاهى Cross Product
 - الضرب القياسى Dot Product
 - طول المتجهات Magnitude
 - اتجاه أوجه الأسطح Normalization
 - نظم التحويلات بشكل مباشر 2 The Transformation Pipeline II
 - o المصفوفة الكلية The World Matrix
 - o المصفوفة الرؤية The View Matrix
 - o مصفوفة الرؤية الطبيعية The Perspective Projection Matrix
 - مدى الرؤية Field of View
 - معدل سرعة العرض Aspect Ratio

مشاريع مطلوبة لهذا الدرس :
 Lab Project 1.1: Wireframe Renderer
 أسئلة / اختبارات : لا يوجد
 عدد ساعات الدراسة : من 8 إلى 10 ساعات

الدرس الثانى

العناصر الرئيسية	رقم الصفحة
المتجهات Vectors	40
حساب طول المتجهات Vector Magnitude	42
جمع و طرح المتجهات Vector Addition and Subtraction	44
ضرب المتجهات فى الأعداد Vector Scalar Multiplication	46
متجهات الوحدة Unit Vectors	47
الضرب الإتجاهى The Cross Product	49
أتجاه أوجه الأسطح Normals	50
الضرب القياسى The Dot Product	51
المسطحات Planes	53
المصفوفات Matrices	57
ضرب المصفوفات فى المصفوفات Matrix/Matrix Multiplication	58
ضرب المصفوفات فى المتجهات Matrix/Vector Multiplication	60
تدوير المصفوفات الثلاثية الأبعاد 3D Matrix Rotation	62
مصفوفة الوحدة Identity Matrices	63
التحجيم الثابت و المتغير للمصفوفات Scaling and Shearing Matrices	64
دمج المصفوفات Matrix Concatenation	65
نظم الإحداثيات المحايدة Homogeneous Coordinates	66
الأعداد المركبة و المتجهات Quaternion's	70
الكائن D3DXMATRIX	71
الكائن D3DXVECTOR3	72
الكائن D3DXPLANE	73
الكائن D3DXQUATRNION	74
دوال الكائن D3DX	75
مصفوفة اللعبة The World Matrix	77
مصفوفة الرؤية The View Matrix	81
مصفوفة الرؤية على الشاشة The Perspective Projection Matrix	83
الرؤية الحرة Arbitrary FOV	86
حجم شاشة العرض Aspect Ratio	
خاتمة Conclusion	

المتجهات **Vectors** هي كيان رياضي يصف مكان و طول نقطة معينة في الفراغ . المتجهات في لغة ال **C++** تمثل كما يلي :

```
class CVector
{
    float x;
    float y;
    float z;
};
```

المتجهات موضوع مهم للغاية لأي مبرمج **3D** . لأن أي شيء داخل العالم الثلاثي الأبعاد لابد و أن يمثل بمتجه حتى نستطيع التعامل معه ، و حيث أننا و صلنا لتلك النقطة ، قد تلاحظ أنه يوجد تشابه بين النقطة و المتجه داخل الإحداثي الديكارتي الثلاثي الأبعاد ، في الواقع النقطة ثلاثية الأبعاد هي متجه و لنكون أكثر تحديداً هي متجه ثلاثي الأبعاد ، و من هذا المنطلق نجد أنه يوجد أيضاً متجه ثنائي الأبعاد و متجه رباعي الأبعاد و هكذا لباقي الأبعاد .

و بالرغم أن بعض المبرمجين يستخدموا اللفظين متجه **Vector** و نقطة **Point** كأنهم ذات معنى واحد ، إلا أنه يوجد بين المصطلحين إختلاف جذري بينهم و هو **أن النقطة دائماً و أبداً ما تكون متجه و لكن العكس ليس صحيح دائماً** ، لأنها تعتمد على كيفية تمثيل القيم المخزنة داخل المتجه : فهل هي تمثل مكان معين في الفراغ (حيث يكون المتجه يمثل فعلاً نقطة معينة) ، أو هي تمثل إتجاه و طول (حيث أنه من الممكن أن يعبر عن نقاط أخرى داخل الفراغ) .

قد تقوم بتعريف المتجهات باستخدام إحداثي معين يبدأ من نقطة أصل النظام الإحداثي ، و طول المتجه هو المسافة اللازمة للوصول لهذا الإحداثي ، لذا إذا تحركنا بدءاً من نقطة الأصل في اتجاه معين و لمسافة معينة سننتهي عند إحداثي معين داخل الفراغ الثلاثي الأبعاد . يتم تمثيل المتجهات رياضياً كما يلي $A <x,y,z>$ حيث **A** هو إسم المتجه و **x,y,z** يمثلوا إحداثيات المتجه.

في الصورة القادمة ، نرى مجموعة من النقاط رسمت داخل النظام الإحداثي الديكارتي الثنائي الأبعاد ، و بالرغم أن كل نقطة على حدى من الممكن أن توصف بأنها مجموعة من الإحداثيات تبدأ من نقطة الأصل و تمتد بمحاذاة محورها ، إلا أن كل نقطة منهم من الممكن أن توصف أيضاً بأنها متجه يبدأ من نقطة الأصل و يمتد حتى يصل لتلك النقطة (الأسهم الخضراء).

المتجهات

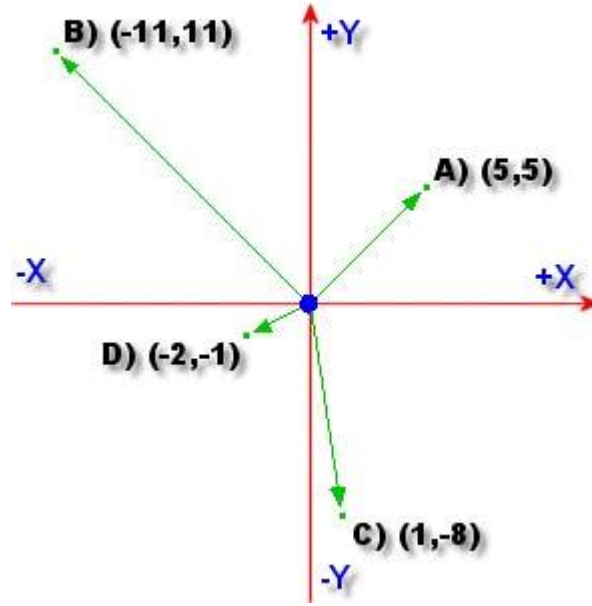


Fig 1.24

المتجه $A <5,5>$ يمكن وصفه بأن مكانه يقع على بعد 8 وحدات يمين نقطة الأصل ، و يقع على بعد 3 وحدات أعلى نقطة الأصل ، و هي أيضاً تصف السهم الأخضر و الذي له اتجاه معين - و هو من نقطة الأصل إلى مكان النقطة - و طول معين (و هي المسافة من نقطة الأصل إلى النقطة A) .

لإيجاد طول المتجه يمكن الإستعانة بنظرية فيثاغورس ، وهى تنص على أن طول المتجه يساوى الجذر التربيعى لمجموع مربع المحاور . و طول المتجه - كما قلنا سابقاً - هو المسافة من نقطة الأصل و حتى النقطة **A** ، أو هو طول السهم الأخضر الذى يصل بينهم .

$$2DVectorMagnitude = \sqrt{X^2 + Y^2}$$

عند التعامل مع المتجهات داخل نظام إحداثى ثلاثى الأبعاد ، فإن المعادلة السابقة يتظل كما هى و سنقوم فقط بإضافة المحور الجديد إليها ، لذا فالمعادلة الجديدة ستكون كالتالى :

$$3DVectorMagnitude = \sqrt{X^2 + Y^2 + Z^2}$$

سنقوم الآن بتحويل المعادلة السابقة لدالة - داخل لغة **C++** - تعود بطول المتجه :

```
long VectorLength3D ( CVector v )
{
    return Sqrtf ( ( v.x * v.x ) + ( v.y * v.y ) + ( v.z * v.z ) );
}
```

سنقوم بتطبيق المعادلة السابقة على متجه **2D** و هو **A<5,5>**

$$Length = \text{Sqrtf} ((5 * 5) + (5 * 5))$$

$$= \text{Sqrtf} (50)$$

$$= 7.0710$$

إذا تحركنا لمسافة 7.0710 بدءاً من نقطة الأصل و بمسافة متساوية للمحورين **X** و **Y** - لإن قيمة كل منهما مساوية لأخرى - حينها سنصل للنقطة (5,5) .

فالنحرب مرة أخرى مع متجه آخر **C<1,-8>**

$$Length = \text{Sqrtf} ((1 * 1) + (-8 * -8))$$

$$= \text{Sqrtf} (65)$$

$$= 8.06225$$

إذا تحركنا لمسافة 8.06225 بدءاً من نقطة الأصل و فى الإتجاه الموجب للمحور **X** و الإتجاه السالب للمحور **Y** و بنسبة 1:8 سنجد أنفسنا عند النقطة (1,-8) .

يتم توزيع المسافة على المحورين بنسبة **X:Y** فى الانظمة الـ **2D** و بنسبة **X:Y:Z** فى الأنظمة الـ **3D** ، بمعنى أنه كلما تحرك **X** وحدة تتحرك امامها **Y** وحدة .

رياضياً يتم تمثيل طول المتجه بإستخدام عمودين متجاورين و بينهم أسم المتجه .

$$Length \text{ of } \mathbf{C} = |\mathbf{C}|$$

على الرغم من أستخدامنا للمتجهات فى هذا الدرس بشكل محدود إلا أننا سنقوم بإستخدامها لتمثيل رؤوس المجسمات التى سنتعامل معها فيما بعد ، و لكن عمل المتجهات لا يقتصر على ذلك فحسب و لكن لها إستخدامات عديدة فى البرمجة ثلاثية الأبعاد كمعرفة الإتجاه الذى تواجهه الكاميرا أو اتجاه انعكاس الضوء على مجسم ما أو رأس معينة . سيتم أستخدام المتجهات أيضاً داخل نظام أكتشاف التصادمات **Collision Detection System** لجعل الكائنات تتحرك داخل اللعبة و إن صادف أن اصطدم كائنين ببعضهما البعض لا يندمجوا و لكن يرتدوا .

عملية جمع متجهين تتم عن طريق جمع المحاور المتشابهة مع بعضها البعض لينتج لنا متجه جديد. سنقوم الآن بكتابة دالة تقوم بجمع متجهين و العودة بمتجه هو مجموعهما :

```
CVector AddVectors3D (CVector A, CVector B)
{
    CVector C;
    C.x=A.x + B.x;
    C.y=A.y + B.y;
    C.z=A.z + B.z;
    return C;
}
```

لتقوم بجمع متجهين 2D ، فقط قم بحذف السطر الخاص بالمحور Z .

يمكن تمثيل المتجه C الناتج من إضافة المتجهين A و B لبعضهما - داخل نظام 2D - بيانياً عن طريق أخذ بداية المتجه B و وضعه عند نهاية المتجه A و بعد ذلك نرسم متجه بدايته عند بداية المتجه A و نهايته عند نهاية المتجه B كما فى الشكل التالى .

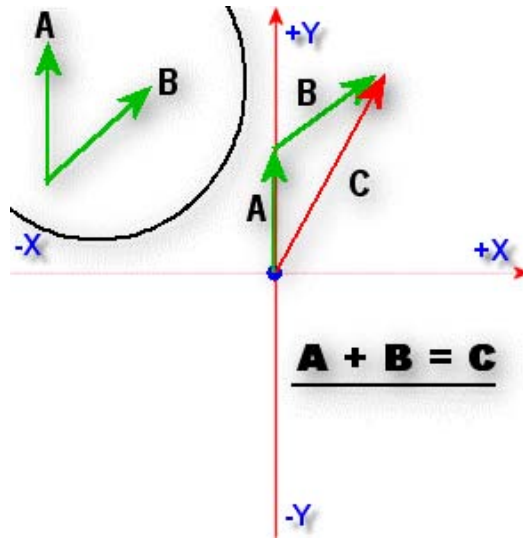


Fig 1.25

فى الشكل السابق يظهر بداخل النصف دائرة المتجهين A و B و العلاقة بينهم قبل عملية الإضافة . أثناء عملية الإضافة ، يبدأ المتجهان A و B من نقطة الأصل ويتم بعدها إضافة المتجه B إلى المتجه A لذا فبداية المتجه B ستكون من نهاية المتجه A . و النتيجة النهائية هى المتجه C والذى يظهر باللون الأحمر .

عملية طرح متجهين مماثلة لعملية الجمع (تقريباً) :

```
CVector SubtractVectors3D (CVector A, CVector B)
{
    CVector C;
    C.x=A.x - B.x;
    C.y=A.y - B.y;
    C.z=A.z - B.z;
    return C;
}
```

لأن طرح B من A مماثلة لجعل B سالبة ثم إضافتها إلى A لذا قد نمثل تلك العملية كالتالى :

$$C = A - B$$

$$C = A + (-B)$$

أو النسخة السالبة من المعادلة :

يمكن تمثيل المتجه الناتج من تلك العملية بيانياً عن طريق أخذ بداية المتجه **B** ثم وضعه في نهاية المتجه **A** كما حدث في عملية الجمع . هذه المرة سنقوم بعكس إتجاه المتجه **B** ليواجه المنطقة المقابلة لإتجاهه السابق ، الصورة التالية توضح عملية طرح **A** و **B** داخل نظام **2D** . و فيها يتم طرح **B** من **A** لينتج لنا المتجه **C** و الذى يظهر باللون الأحمر .

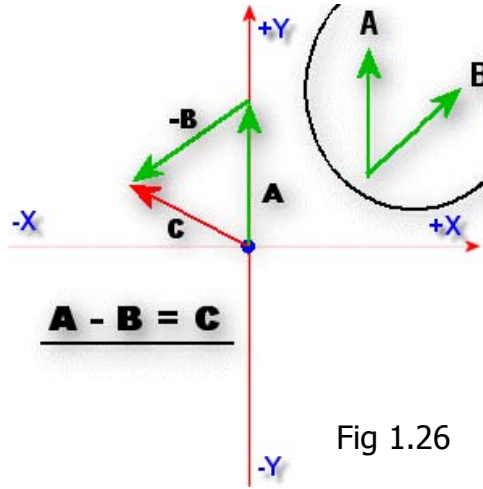


Fig 1.26

طرح المتجهات هى عملية مفيدة جداً ، فهى تتيح لنا معرفة العلاقة بين المجسمات و بعضها داخل اللعبة . فعلى سبيل المثال ، لدينا طائرتان حريتان داخل اللعبة إحداهما فى الموقع **A** و الأخرى فى الموقع **B** ، فإذا قمنا بطرح الموقع **B** من **A** سينتج لنا متجه جديد يعطينا الإتجاه الذى على الطائرة **A** أن تتحرك خلاله لتصل لموقع الطائرة **B** كما يعطينا المسافة اللازمة للوصول للطائرة **B** و هى عن طريق حساب المتجه الجديد . الشكل التالى يوضح لنا تلك العملية :

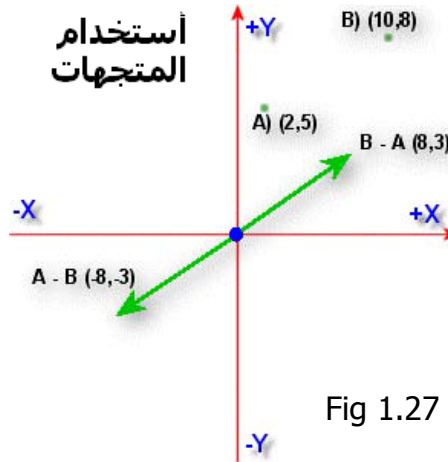


Fig 1.27

فى الشكل السابق ، توجد نقطتين هما **A** و **B** يمثلوا الطائرتان ، فإذا قمنا بطرح النقطة **B** من النقطة **A** سينتج لنا المتجه **B-A** و يساوى $\langle 8, 3 \rangle$ و يمثل بالسهم الأخضر الذى يشير لأعلى اليمين . من هنا يمكن حساب طول المتجه كالتالى :

$$\begin{aligned} |B-A| &= \sqrt{8^2 + 3^2} \\ &= \sqrt{64 + 9} \\ &= 8.544 \end{aligned}$$

المسافة بين الطائرتان تساوى 8.544 وحدة . لذا لتصل الطائرة **A** للطائرة **B** لابد و أن تتحرك بمعدل 8:3 بمحاذاة المحورين **X** و **Y** الموجبين على التوالى .

يمكن ضرب المتجهات فى أرقام ، و فى هذه الحالة يتم ضرب ذلك الرقم فى قيمة كل محور على حدى .

التالى دالة صممت لتقوم بضرب متجه 3D فى قيمه عددية و هى كالتالى :

```
CVector VectorMultiply3D (CVector A, float scalar)
{
    CVector C;
    C.x=A.x * scalar;
    C.y=A.y * scalar;
    C.z=A.z * scalar;
    return C;
}
```

الصورة التالية توضح تلك العملية :

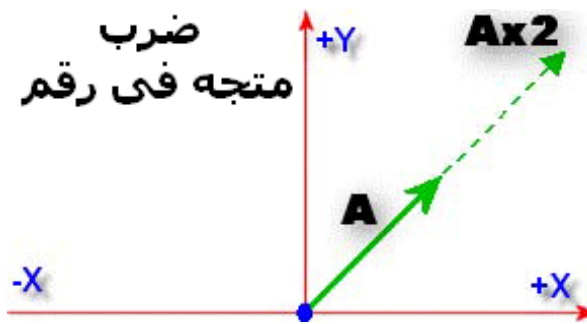


Fig 1.28

متجه الوحدة هو نوع خاص جداً من المتجهات و هو مفيد للغاية لمبرمجي الـ 3D ، و سمي بهذا الإسم لأن طوله يساوى واحد صحيح . عملية تحويل المتجه العادى لمتجه وحدة تسمى بتوحيد المتجه **Normalization** ، و هى تتم بقسمة قيمة كل محور داخل المتجه على طوله .

مثال :

$$\text{VectorC} = (8, 3, 10)$$

$$\begin{aligned} |\text{VectorC}| &= \sqrt{8^2 + 3^2 + 10^2} \\ &= \sqrt{64 + 9 + 100} \\ &= 13.152946 \end{aligned}$$

$$\begin{aligned} \text{UnitVectorC} &= \left(\frac{C.x}{|C|}, \frac{C.y}{|C|}, \frac{C.z}{|C|} \right) \\ &= \left(\frac{8}{13.152946}, \frac{3}{13.152946}, \frac{10}{13.152946} \right) \\ &= (0.60822, 0.2280, 0.7602859) \end{aligned}$$

المتجه **C** الآن هو متجه وحدة ، و لإثبات ذلك لنقوم بحساب طوله و الذى لابد و أن يكون واحد .

$$\text{UnitVectorC} = (0.60822, 0.2280, 0.7602859)$$

$$\begin{aligned} |\text{UnitVectorC}| &= \sqrt{0.60822^2 + 0.2280^2 + 0.7602859^2} \\ &= 0.99999999..... \end{aligned}$$

الرقم السابق أقرب ما يكون للواحد الصحيح و سيكون كذلك إذا قمنا باستخدام التقريب .

لاحظ أن طالما أن طول المتجه أصبح 1 ستجد أن اتجاهه أصبح مثل طوله ، و فى الواقع جميع قيم محاور المتجه تم تحجيمها بما يتناسب مع طوله .

الدالة التالية تقوم بتحويل المتجه لمتجه وحدة و فيها نقوم باستخدام أحد الدوال التى سبق انشائها وهى **VectorLength3D** :

```
CVector VectorNormalize3D (CVector A)
{
    float length=VectorLength3D( A );
    CVector C;
    C.x = A.x / length;
    C.y = A.y / length;
    C.z = A.z / length;
    return C;
}
```

عرفنا فيما سبق ان المتجهات تستخدم فى تحريك المجسمات ، فالفترض إذاً أنه لدينا سفينة فضاء وجهها للأسفل باتجاه المحورين **X** و **Y** داخل اللعبة ، سنمثل إتجاه السفينة بمتجه سنسميه **DirectionVector** و إحداثياته هى (1,0,1) ، فالتخيل أننا نريد تحريك السفينة 100 وحدة .

$$\begin{aligned} \text{DirectionVector} &= (1, 0, 1) \\ \text{Speed} &= 100 \end{aligned}$$

$$\begin{aligned} \text{Movement.x} &= \text{DirectionVector.x} * \text{Speed}; \\ \text{Movement.y} &= \text{DirectionVector.y} * \text{Speed}; \\ \text{Movement.z} &= \text{DirectionVector.z} * \text{Speed}; \end{aligned}$$

$$\text{Movement.x} = 1 * 100 = 100$$

$$\text{Movement.y} = 0 * 100 = 0$$

$$\text{Movement.z} = 1 * 100 = 100$$

السؤال التالي هل البيانات السابقة صحيحة ؟ لقد قلنا أن السفينة تحركت 100 وحدة لذا فالتحقق من طول المتجه **Movement** .

$$|\text{Movement}| = \sqrt{100^2 + 0^2 + 100^2} = 141.42135$$

النتيجة السابقة توضح لنا مدى خطأها ، لأن المفروض ان السفينة تحركت 100 وحدة و ليس 141 وحدة . هذا الخطأ حدث لأن المتجه **DirectionVector** ليس متجه وحدة ، و الدليل على ذلك يظهر عند حساب طوله كما يلي :

$$|\text{DirectionVector}| = \sqrt{1^2 + 0^2 + 1^2} = 1.4142135$$

الخطأ يظهر فى أن السفينة تحركت 100 وحدة بمحاذاة المحور **X** و 100 وحدة بمحاذاة المحور **Z** (وهى ليست مثل تحريك السفينة 100 وحدة بشكل قطرى). لذا فقبل إستخدام المتجه **DirectionVector** لحساب متجه الحركة **Movement** لابد أولاً من تحويله لمتجه وحدة :

$$\text{DirectionVector} = \left(\frac{1}{1.4142135}, \frac{0}{1.4142135}, \frac{1}{1.4142135} \right) = (0.7071068, 0, 0.7071068)$$

لنقوم الآن بحساب قيمة متجه الحركة **Movement** مع المتجه **DirectionVector** بعد توقيده :

$$\text{Movement.x} = 0.7071068 * 100 = 70.71068$$

$$\text{Movement.y} = 0 * 100 = 0$$

$$\text{Movement.z} = 0.7071068 * 100 = 70.71068$$

لنقوم الآن بإضافة المتجه **Movement** للمتجه **DirectionVector** لتحرك السفينة :

$$\text{NewPosition.x} = \text{DirectionVector.x} * \text{Movement.x};$$

$$\text{NewPosition.y} = \text{DirectionVector.y} * \text{Movement.y};$$

$$\text{NewPosition.z} = \text{DirectionVector.z} * \text{Movement.z};$$

لنتأكد أن السفينة تحركت 100 وحدة لابد من إيجاد طول متجه الحركة **Movement** :

$$|\text{Movement}| = \sqrt{70.71068^2 + 0^2 + 70.71068^2} = 100$$

لدينا الآن سفينة فضاء موجودة بمكان جديد ، هذه السفينة تحركت مسافة 100 وحدة تماماً بدءاً من مكانها السابق ، التحرك لـ 100 وحدة تساوى التحرك 70.7 وحدة بمحاذاة المحور **X** ، 70.7 وحدة بمحاذاة المحور **Z** . إذا كانت متجهات الوحدة تستخدم لتحديد الإتجاه الذى تواجهه المجسمات داخل اللعبة ، لذا فإستخدام الأسلوب السابق يمكننا تحريك المجسمات للإمام (ولا يهم ما الإتجاه الذى يواجهه المستخدم) .

تحريك المجسمات هى أحد الإستخدامات العديدة لمتجهات الوحدة ، لأنها تستخدم أيضاً فى وصف اتجاه أوجه المجسمات ، و أستخدامات أخرى عديدة أهمها خلال عمليات الإضاءة .

ستكتشف بعد فترة أن متجهات الوحدة تستخدم طوال الوقت فى برمجة الـ **3D** ، و سوف نعرف الكثير و الكثير عن تلك الإستخدامات فى الدروس القادمة .

عملية الضرب الإتجاهى بين متجهين تنتج متجه ثالث عمودى على المتجهين الآخرين ، حرف الـ "x" يشير لعملية الضرب الإتجاهى بين متجهين .

مثال :

$$A = (0, 1, 0)$$

$$B = (1, 0, 0)$$

عملية الضرب الإتجاهى :

$$C = A \times B$$

$$C = ((A.y * B.z) - (A.z * B.y), (A.z * B.x) - (A.x * B.z), (A.x * B.y) - (A.y * B.x))$$

$$= ((1 * 0) - (0 * 1), (0 * 1) - (0 * 0), (0 * 0) - (1 * 1))$$

$$= (0, 0, -1)$$

المتجه الناتج - و هو C - عمودى بزاوية قدرها 90 درجة على المتجهين A و B ، فى الشكل القادم السهمين ذى اللون الأخضر يمثلوا المتجهين A و B . و المتجه C الناتج من عملية الضرب الإتجاهى يظهر باللون الأحمر و عمودى على الآخرين .

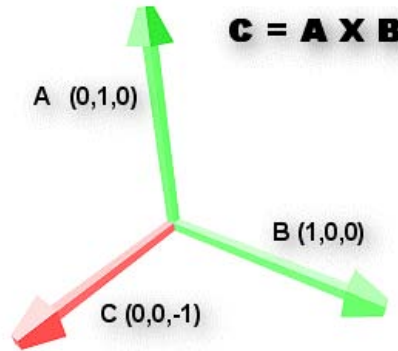


Fig 1.29

فى المثال السابق قمنا بإستخدام متجهين وحدة فى عملية الضرب الإتجاهى لذا فالمتجه الناتج هو متجه وحدة أيضاً و هو عمودى عليهم . عملية الضرب الإتجاهى لا تشترط أن يكون المتجهين متجهى وحدة . و لكن ليس معنى ذلك أن المتجه الناتج لن يكون عمودى عليهم ، **لذا فلاى عملية ضرب إتجاهى سواء كان عنصرى المعادلة أحدهما أو كلاهما أو لا أحد منهما متجه وحدة فإن المتجه الناتج سيكون دائماً عمودى عليهم** . ولكن فى بعض الأحيان يتطلب أن يكون المتجه المدخل لعملية الضرب الإتجاهى أن يكون متجه وحدة ، لذا فى تلك الحالة ستقوم بتوحيد المتجه أولاً **Normalize** قبل إدخاله للمعادلة .

الترتيب السابق الخاص بالمتجهات داخل معادلة الضرب الإتجاهى هام جداً ، حيث إذا قمنا بتبديل عنصرى المعادلة أى قمنا بتنفيذ المعادلة $B \times A$ بدلاً من $A \times B$ فإن المتجه C الناتج سيكون عمودى عليهم و لكن فى الإتجاه المعاكس ، فالتحاول معرفة ناتج المعادلة الجديدة على ورقة خارجية لترى الناتج بنفسك .

لذا فعملية الضرب الإتجاهى تتم على أى متجهين و تنتج فى النهاية متجه عمودى على كلاهما .

CVector VectorCrossProduct (CVector A, CVector B)

```
{
    CVector C;
    C.x= (A.y * B.z) - (A.z * B.y);
    C.y= (A.z * B.x) - (A.x * B.z);
    C.z= (A.x * B.y) - (A.y * B.x);
    return C;
}
```

أحد الإستخدامات المفيدة للضرب الإتجاهى إنشاء بما يسمى بوجه السطح **Surface Normal** .
 اتجاه الوجه **Normal** هو متجه وحدة يصف الإتجاه الذى يواجه سطح المجسم .

فى الصورة القادمة نرى سطح مكون من 3 متجهات هما (v_0, v_1, v_2) إذا قمنا بطرح v_0 من v_1 فالمتجه الناتج سيصف احد حواف هذا السطح **Edge 1** . و بالمثل إذا قمنا بطرح v_0 من v_2 سنحصل على **Edge 2** . عملية الضرب الإتجاهى لهاتين الحافتين ينتج عنهم متجه بعد توحيدده يصبح هو متجه إتجاه وجه السطح **Polygon Surface Normal** .

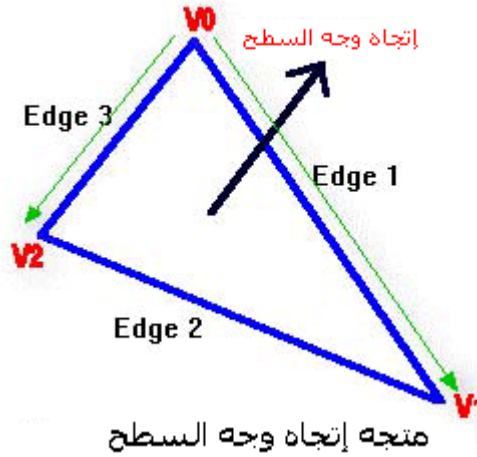


Fig 1.30

الدالة القادمة تفترض أنك أنشئت كائن من نوع **CPolygon** و أضفت داخله مجموعة من الرؤوس من نوع **CVertex** . تقوم الدالة التالية بإستخدام مجموعة من الدوال التى صنعناها من قبل حتى تعمل تلك الدالة بالشكل المرجو منها . نقوم بإدخال كائن من نوع **CPolygon** للدالة و تقوم هى بحساب متجه وجه السطح لذا فالقيمة العائدة منها هى من نوع **CVector** .

```
CVector GeneratePolygonNormal (CPolygon P)
{
    CVector Edge1, Edge2, Normal;

    Edge1 = SubtractVector3D (P.Vertices [1], P.Vertices [0]);
    Edge2 = SubtractVector3D (P.Vertices [2], P.Vertices [0]);
    Normal = VectorCrossProduct (Edge1, Edge2);
    Normal = VectorNormalize3D (Normal);
    return Normal;
}
```

ملاحظة هامة :

متجه الوجه **Normal** دائماً ما يكون عمودى على السطح **Polygon** .

فى حال أننا قمنا بتدوير سطح موجود داخل مجسم ما ، لابد من إستدعاء الدالة السابقة ببيانات السطح الجديدة حتى نعرف إتجاه وجه الحالى .

الضرب القياسى - أو الضرب الداخلى - هى عملية تتم بين متجهين **3D** ، يستخدم الرمز • أو ⊙ لتمثيل عملية الضرب القياسى بين متجهين . عملية الضرب القياسى بين المتجهين **A** و **B** تتم كالتالى :

$$A \odot B = (A.x * B.x) + (A.y * B.y) + (A.z * B.z)$$

نتاج عملية ضرب المحاور المتشابه ثم جمعها مع بعضها البعض ينتج لنا رقم وليس متجه - على عكس الضرب الإتجاهى الذى ينتج متجه و ليس رقم . عملية الضرب القياسى يمكن أتمامها بشكل آخر و هو التالى:

$$A \odot B = \cos(\theta) * |A| * |B|$$

القيمة العائدة من الضرب القياسى تساوى جيب تمام الزاوية بين المتجهين مضروباً فى طول المتجه **A** مضروباً فى طول المتجه **B** ، لذا نستطيع الحصول على جيب تمام الزاوية بين المتجهين كالتالى :

$$\cos(\theta) = \frac{A \odot B}{|A| * |B|}$$

فى حال أن المتجهين **A** و **B** متجهى وحده ، لذا فطولهما يساوى واحد ، و هذا يتيح لنا حذف المقام لأن المقام يساوى واحد .

$$\cos(\theta) = A \odot B$$

تحويل الدالة **cos** إلى مقلوبها **acos** يتيح لنا معرفة الزاوية بين المتجهين ، حيث أن جيب تمام الزاوية هو الذى يساوى ناتج الضرب القياسى و ليست الزاوية هى التى تساوى ناتج الضرب القياسى . والزاوية الناتجة من الدالة **acos** تكون بالنظام العشري .

سنقوم الآن بكتابة دالة لمعرفة الزاوية بين متجهين و التى نقوم بغدخال متجهين إليها من نوع **CVector** و تعود بالزاوية بينهم ، و لكن و لسوء الحظ إيجاد الزاوية بين المتجهين بهذا الأسلوب يتطلب منا معرفة طول المتجهان وتلك العملية تتطلب إجراء 3 عمليات ضرب و 3 عمليات جذر تربيعى ، العمليات السابقة طويلة و بطيئة نسبياً ، لذا فإننا سنحاول جاهدين استخدام متجهات وحدة كلما أمكن لإنها تساعد فى تسريع العملية كثيراً ، الدالة التالية تقوم بحساب الضرب القياسى لمتجهين :

```
float VectorDotProduct3D (CVector A, CVector B)
{
    return (A.x * B.x + A.y * B.y + A.z * B.z);
}
```

إذا كانا المتجهان المدخلان للدالة السابقة متجهى وحدة فإن ناتج الدالة السابقة هو جيب تمام الزاوية بين المتجهان .

فى حال أننا نريد معرفة الزاوية الموجودة بين متجهين و ليس من المتوقع أن يكونا متجهى وحدة ، لذا فإننا سنقوم بكتابة دالة تقوم بشكل إفتراضى بحساب الضرب القياسى للمتجهين ثم القسمة على طولهما :

```
float FindVectorAngles3D (CVector A, CVector B)
{
    float LengthOfA = VectorLength3D (A);
    float LengthOfB = VectorLength3D (B);
    return ((A.x * B.x) + (A.y * B.y) + (A.z * B.z) / LengthOfA * LengthOfB);
}
```

الشكل التالى يوضح كيف يعمل الضرب القياسى فى تحديد الزاوية بين متجهين **2D** :

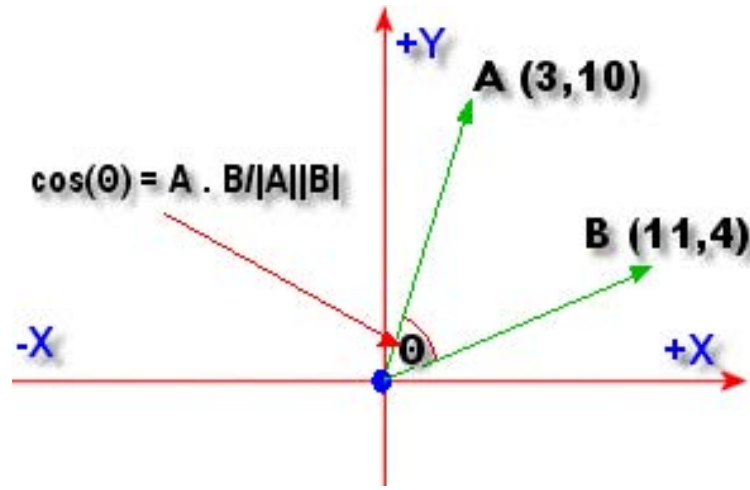


Fig 1.31

$$|A| = \sqrt{3^2 + 10^2} = 10.440306$$

$$|B| = \sqrt{11^2 + 4^2} = 11.704699$$

$$\cos(\theta) = ((3 * 11) + (10 * 4)) / (10.440306 * 11.704699) \\ = 73 / 122.20063$$

$$\theta = \arccos(0.5973782)$$

$$\theta = 0.93056 \text{ Radian}$$

$$\theta = 53.31 \text{ Degree}$$

نقطة هامة و هى أن أثناء عملية الضرب القياسى جميع المتجهات تشترك جميعاً فى نقطة أصل واحدة . يمكنك تذكر تلك النقطة بوضع بداية جميع المتجهات فى نقطة واحدة هى نقطة الأصل .

فى أغلب الاحوال سنريد معرفة ما إذا كانت الزاوية بين المتجهين أكبر من اول أقل من 90 ، فإذا لم يكن هناك داع من الحصول على قيمة الزاوية بين المتجهين - أى لن نقوم باستخدامها داخل البرنامج - لذا فلا داعى لإستخدام متجهات وحدة أو القسمة على أطوال المتجهات ، لأن إشارة الناتج لن تتغير لأن أطوال المتجهات - التى نقسم عليها - دائماً ما تكون موجبة .

لذا توجد بعض العناصر التى لابد من معرفتها عن الضرب القياسى لمتجهين :

1. إذا كانت الزاوية أكبر من 90 درجة سيكون ناتج الضرب القياسى موجب .
2. إذا كانت الزاوية تساوى 90 درجة سيكون ناتج الضرب القياسى صفر .
3. إذا كانت الزاوية أقل من 90 درجة سيكون ناتج الضرب القياسى سالب .

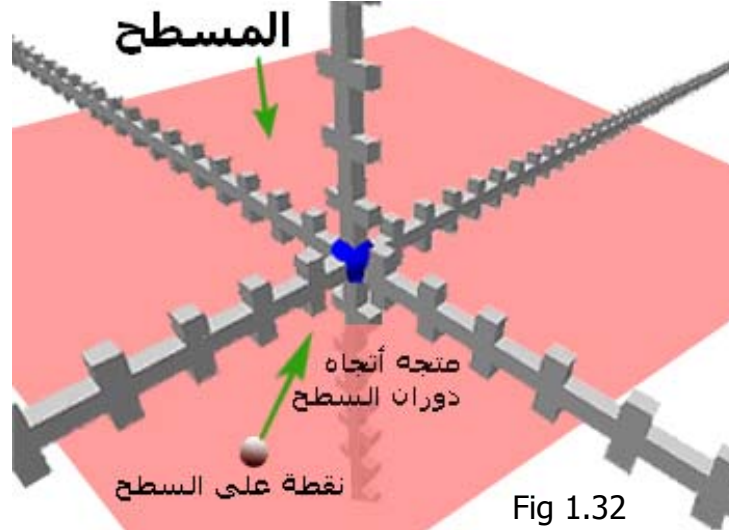
	الصورة الموجودة باليسار توضح بشكل مبسط العلاقة بين ناتج الضرب القياسى و الزاوية بين المتجهين . إذا كان المتجهان متجهى وحدة فإن الناتج سيكون جيب تمام الزاوية بينهم ، بعد ذلك نقوم بتمرير الناتج للدالة acos لمعرفة الزاوية بين المتجهين بقياس Radian (نقوم بالحصول على الزاوية عند الحاجة فقط). لاحظ أن المتجهان اللذان يتم تمريرهما إلى الدالة تتحرك نقطة البداية لهما لتتشاركا فى نقطة أصل واحدة خلال عملية الضرب القياسى .
--	--

المسطح **Plane** هو شريحة رفيعة تمتد بشكل لا نهائى فى جميع الإتجاهات داخل العالم الـ **3D** ، أما تمثيل المسطح داخل عالم **2D** فيكون بخط رفيع يمتد إلى مالا نهاية . و حتى تستوعب فكرة المسطحات جيداً ، أمسك ورقة و قم بتدويرها لأى زاوية (تأكد من انك ترى جميع حواف الورقة) . تخيل معى الآن أن هذه الورقة لا حواف لها و أنها تمتد لجميع الإتجاهات بشكل لا نهائى ، هكذا يكون المسطح . و بالرغم من أن المسطحات تمتد بشكل لا نهائى إلا أنه يوجد لها زاوية دوران داخل العالم الـ **3D** .

تدوير قطعة الورقة لمجموعة مختلفة من الزوايا يرينا عدد لا نهائى من المسطحات ، و بذلك فلكل زاوية دوران مسطح خاص بها . إذا قمت برسم سطح ما **Polygon** فى مركز قطعة الورقة ستجد أنه بتدويرك للورقة يتم تدوير السطح الموجود بالورقة بنفس زاوية تدوير الورقة و أيضاً تلاحظ أن السطح مازال موجود على هذه الورقة . فى الواقع الأسطح **Polygons** هى مسطح تم تحديد أبعاده ، و الدليل على ذلك ، أنك إذا تخيلت أن ذلك السطح الذى قمنا فى مركز الورقة لا حواف له و أنه يمتد فى جميع الإتجاهات ، إذا فعلت ذلك ستجد لديك المسطح الذى يرقد عليه ذلك السطح .

المسطحات مفيدة جداً فى حالات كثيرة . فعبي سبيل المثال ، إذا كانت لدينا نقطة (فالفترض أنها مكان الكاميرا) و هى خلف مسطح معين (أى أن الكاميرا تنظر لظهر المسطح) ، من السابق نعرف أن جميع الأسطح الموجودة على هذا المسطح هى على الجانب الآخر من الكاميرا و لا يمكن أن ترى ، و هذا يتيح لنا عدم إجراء اختبار على هذه الأسطح لإننا لن نقوم برسمها أصلاً .

فى الصورة القادمة يظهر مسطح داخل النظام الديكارتى الثلاثى الأبعاد ، المسطح الأحمر بشكل إفتراضى يمتد إلى مالا نهاية و لكننا هنا قطعنا أطرافه و جعلناه يمتد لأطراف الصورة فقط لتسهيل الرؤية ، أيضاً على هذا المسطح توجد نقطة و معها متجه ليصف إتجاه دوران المسطح (يظهر باللون الأخضر) . هذا المتجه يسمى بمتجه إتجاه دوران المسطح **Plane Normal** . و هو يشبه كثيراً متجه إتجاه وجه السطح **Polygon Normal** و قد تم شرحه سابقاً فى هذا الدرس . متجه إتجاه دوران السطح هو أيضاً متجه وحدة و يصف إتجاه دوران المسطح .



يمكن إنشاء **Class** لتمثل المسطح و شكلها كالتالى :

```
class CPlane
{
Public:
    CVector PlaneNormal;
    float DistanceToPlane;
};
```

المتغير **PlaneNormal** هو متجه وحدة و يصف إتجاه دوران المسطح ، و المتغير **DistanceToPlane** هو المسافة اللازمة للوصول للمسطح و يتم قياسها بدءاً من نقطة الأصل و حتى أقرب نقطة موجودة على المسطح (و هى النقطة السوداء الموجودة فى الصورة السابقة) . يمكن معرفة أقرب مسافة من نقطة الأصل و حتى أقرب نقطة على المسطح عن طريق صنع خط **يوازي متجه إتجاه دوران المسطح** و هذا الخط يتم تحريكه حتى يكون أحد نقطتي بدايته عند نقطة الأصل و ستجد أن الأخرى بشكل تلقائى موجودة على المسطح ، كما يظهر بالشكل التالى :

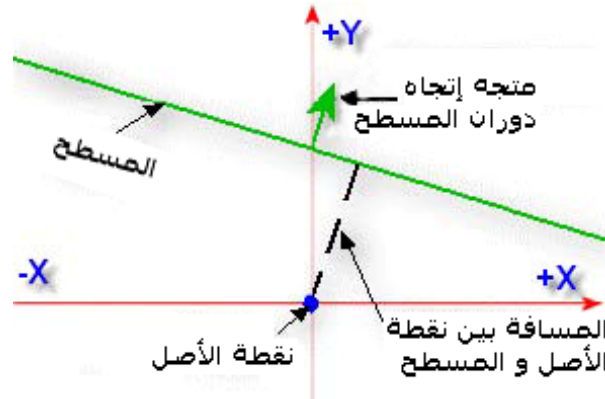


Fig 1.33

الشكل السابق هو صورة **2D** للعالم الـ **3D** كما تتم رؤيته من الجانب ، و يظهر فيها المسطح كخط رفيع لا نهائى ، هذا الخط له عمق يمتد إلى داخل الورقة إلى مالا نهاية و كذلك إلى خارج الورقة .

و حيث أن متجه دوران المسطح يتجه بعيداً عن نقطة الأصل ، لذا يقال أن نقطة الأصل **تقع خلف المسطح** ، و فى هذه الحالة تصبح المسافة من نقطة الأصل للمسطح قيمة موجبة . و لكن إذا كان متجه دوران المسطح يواجه نقطة الأصل فهنا يقال ان **نقطة الأصل تقع أمام المسطح** ، و فى هذه الحالة تصبح المسافة بين نقطة الأصل و المسطح قيمة سالبة .

لتجد المسطح الذى يقع عليه سطح ما (و أيضاً متجه دوران المسطح) سنقوم بحساب متجه إتجاه وجه السطح باستخدام الضرب الإتجاهى لحواف السطح كما شرحنا سابقاً ، و بعد ذلك سنقوم بنسخ البيانات مباشرة للمتغير **PlaneNormal** . لنقوم بحساب المسافة بين نقطة الأصل و المسطح سنستخدم الضرب القياسى .

كما ذكرنا سابقاً ، الضرب القياسى لمتجهين وحدة يساوى جيب تمام الزاوية بينهم ، و فى حال أن أحد المتجهين ليس بمتجه وحدة فالناتج سيساوى جيب تمام الزاوية مضروباً فى طول المتجه الغير أحادى الطول .

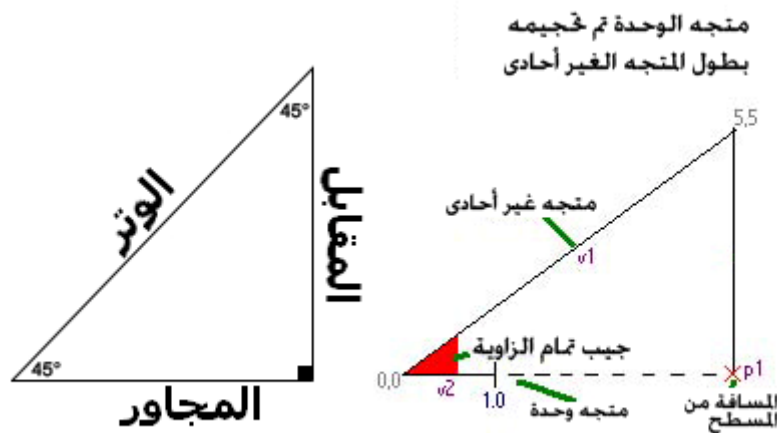


Fig 1.34

فى الصورة السابقة ، المتجه v_1 و هو ليس متجه وحدة يمثل القطر فى المثلث القائم الزاوية ، و متجه الوحدة v_2 يمثل الضلع المجاور فى نفس المثلث و طوله 1 . أيضاً جيب تمام الزاوية بين المجاور و الوتر يساوى طول الضلع المجاور فى المثلث القائم الزاوية .

نتيجة الضرب القياسى بين متجه وحدة و متجه عادى هى طول المتجه العادى فى إتجاه متجه الوحدة ، بمعنى أن المساحة التى يحتلها متجه الوحدة بمقدار نتيجة الضرب القياسى لمتجه الوحدة و المتجه العادى . و فى حالتنا هذه ، إجراء عملية الضرب القياسى لـ v_1 و v_2 سينتج متجه جديد إتجاهه هو إتجاه متجه الوحدة v_2 و طوله يساوى طول المتجه العادى v_1 و بداية هذا المتجه تبدأ من نقطة الأصل (0,0) و نهايته عند النقطة p_1 . فالتخيل معاً أن الضلع المقابل فى ذاك المثلث هو مسطح حيث يوجد سطح ما عليه ، و النقطة (5,5) موجودة على ذلك السطح . لذا أصبح المتجه v_1 هو المتجه الذى يصل بين نقطة الأصل و النقطة (5,5) . و أصبح المتجه v_2 هو متجه إتجاه دوران المسطح (الذى يوجد به السطح) ، لذا فنتائج الضرب القياسى لـ v_1 و v_2 هو أقصر مسافة للمسطح من نقطة الأصل . لاحظ أن هذا لا يعطينا المسافة من نقطة الأصل للمسطح الموجود على المسطح - حيث ليس بالضرورة أن تكون أقصر مسافة للمسطح تقع على السطح ، و لكن تعطينا المسافة بين نقطة الأصل و المسطح الذى يوجد السطح عليه .

متجه يبدأ من نقطة الأصل لأى نقطة على المسطح // $CVector PointOnPlane = (5.0, 5.0, 0.0);$
 متجه وحدة يمثل إتجاه دوران المسطح // $CVector Normal = (1.0, 0.0, 0.0);$
 $float DistanceToPlane = VectorDotProduct3D (PointOnPlane, Normal);$

نتيجة الضرب القياسى هى 5 . لنقوم الآن بإيجاد طول المتجه $PointOnPlane$:

$Length = Sqrtf (5*5 + 5*5 + 0*0);$
 $Length = 7.071067;$

فى الصورة السابقة ، المثلث الموجود باليمين ، الزاويتين الموجودين به قياسهما 45 درجة (و لتأكد من ذلك قم بقسمة طول المتجه العادى - و هو 5 - على ناتج الضرب القياسى السابق) :

الناتج هو جيب تمام الزاوية بين المتجهين // $5 / 7.0710678 = 0.7071067;$
 $float angle = acosf (0.7071067) = 0.7853981;$ // $0.7853981 \text{ Radian} = 45 \text{ Degrees}$

و الخلاصة هى لحساب أقصر مسافة بين المسطح و نقطة الأصل لابد من وجود متجهين . إحداهما هو متجه وحدة و يصف إتجاه دوران المسطح - نفس المتجه قد يكون إتجاه وجه السطح - $PlaneNormal$ ، و الآخر هو متجه عادى يبدأ من نقطة الأصل و ينتهى عند أى نقطة معروفة و موجودة على ذلك المسطح . بعد ذلك نقوم بإجراء عملية الضرب القياسى على هاتين المتجهين و الناتج هو المسافة لذلك المسطح ، و هو كما سظهر فى الصورة التالية :

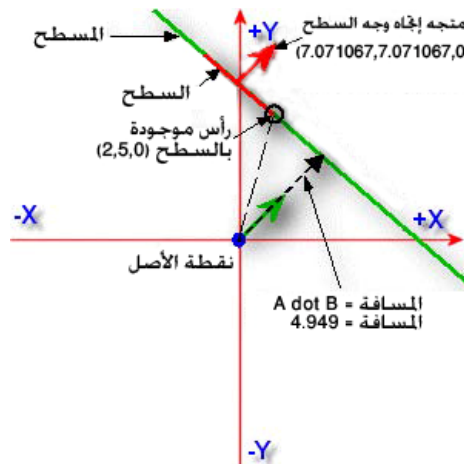


Fig 1.35

سنقوم الآن بكتابة دالة تقوم بحساب المسطح الذي يوجد عليه سطح ما. سنقوم فيما بعد باستخدام دوال المكتبات الـ **DirectX** لتقوم بكل تلك العمليات و لكن من الأفضل فهم كيفية عمل هذه الدوال و ماذا تفعل فى الخفاء :

CPlane GetPolygonPlane (CPolygon P)

```
{
    CVector Edge1, Edge2, Normal;
    CPlane Plane;

    Edge1 = SubtractVector3D (P.Vertices [1], P.Vertices [0]);
    Edge2 = SubtractVector3D (P.Vertices [3], P.Vertices [0]);
    Normal = VectorCrossProduct (Edge1, Edge2);
    Plane.PlaneNormal = VectorNormalize3D (Normal);
    Plane.DistanceToPlane = VectorDotProduct3D (P.Vertices [0],
                                                Plane.PlaneNormal);

    return Plane;
}
```

معرفة المسطح الذي يوجد عليه سطح ما يساعدنا فى تحديد ما إذا كان ذلك السطح يواجه الكاميرا أم لا . سنقوم بعمل اختبار للأسطح داخل فضاء اللعبة **World Space** للتأكد من عدم إجراء تحويلات رؤوس الأسطح التى لن تظهر للمستخدم - أيضاً رؤوس تلك الأسطح سيكون ترتيبها فى عكس عقارب الساعة .

الأسطح الخلفية سيتم حذفها داخل المجال العام

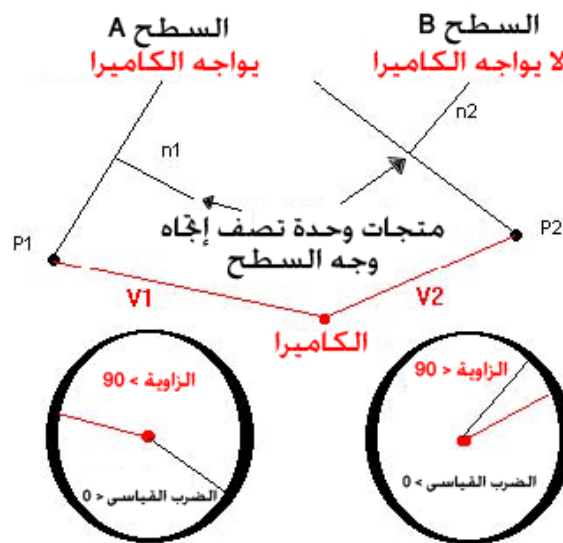


Fig 1.36

فى الصورة السابقة قمنا بإجراء الضرب القياسى بين موقع الكاميرا و متجه إتجاه وجه السطح . قمنا بإنشاء متجه من موقع الكاميرا لى نقطة على المسطح (من الممكن إعتبار أن النقطة موجودة على السطح الموجود على المسطح) . و بعد ذلك قمنا بإجراء الضرب القياسى بين هذا المتجه و متجه إتجاه وجه السطح ، و هذا أعطانا فى النهاية المسافة بين الكاميرا و المسطح .

داخل الدائرة الموجودة أسفل السطح **A** . عند إجراء الضرب القياسى بين المتجه n_1 و v_1 (و الذى قمنا بإنشائه عن طريق طرح قيمة أى رأس موجودة على السطح **A** من موقع الكاميرا) يتم إنشاء زاوية بينهم قياسها أكبر من 90 درجة . و حيث أن $n_1 \odot v_1$ ينتج رقم سالب لذا فالسطح **A** يواجه الكاميرا . و هذا سظهر بشكل واضح مع السطح **B** الذى لا يواجه الكاميرا ، لأن $n_2 \odot v_2$ الزاوية بينهم أقل من 90 درجة و بالتالى فنتائج الضرب القياسى رقم موجب .

المصفوفة هي مجموعة من القيم رتب في جدول على شكل صفوف و أعمدة ، هذا الجدول قد يكون بأى عدد من الصفوف و الأعمدة .

الشكل التالى يبين مصفوفة 3x3 :

$$\begin{bmatrix} 5 & 4 & -1 \\ 0 & 5 & 29 \\ 11 & -7 & 2 \end{bmatrix}$$

التعامل مع عناصر المصفوفة السابقة - و بالطبع جميع المصفوفات - لابد و أن يتم باستخدام أسلوب موحد . يوجد أسلوبين للتعامل مع عناصر المصفوفة ، الأسلوب الأول يسمى **بأسلوب الصف الرئيسى** . العنصر [2][3] فى المصفوفة يعنى القيمة الموجودة فى [العمود 3][الصف 2] و هى تقابل الرقم 29 . الأسلوب الثانى هو **أسلوب العمود الرئيسى** . فمثلا العنصر [2][3] يشير إلى [الصف 3][العمود 2] و عند النظر للمصفوفة فى المثال السابق ستجد انها تطابق الرقم -7 ، لذا فالتعامل مع النظام الخاطئ للدخول للمصفوفة سيعود بالتأكد ببيانات خاطئة .

ملاحظة هامة :

مكتبات الـ **DirectX** تستخدم أسلوب الصف الرئيسى للدخول إلى عناصر المصفوفات و هو الأسلوب الذى سنقوم باستخدامه طوال هذا الكورس .

المصفوفة الموجودة بالمثال السابق هى نوع خاص جداً من المصفوفات و تسمى بالمصفوفة المربعة. و هى تعنى أنها عدد الصفوف التى تحتوى عليها يساوى عدد الأعمدة الموجودة بها . المصفوفة القادمة هى مثال على مصفوفة 3x1 ، و تعنى انها تحتوى على 3 صفوف و عمود واحد فقط :

$$\begin{bmatrix} 3 \\ 9 \\ 3 \end{bmatrix}$$

المصفوفة التالية هى مصفوفة 1x3 :

$$\begin{bmatrix} 3 & 9 & 3 \end{bmatrix}$$

يمكن إستبدال القيم الموجودة فى المصفوفة السابقة بمجموعة من المتغيرات مثل **x,y,z** لتصبح كالتالى :

$$\begin{bmatrix} x & y & z \end{bmatrix}$$

المصفوفة السابقة تطابق المتجه الثلاثى الأبعاد الذى قمنا بتصميمه من قبل . فى الواقع ، يمكن التعامل مع المتجه الـ **3D** كأنه مصفوفة ذات أبعاد 1x3 .

المصفوفة التى تحتوى على عدد **m** من الصفوف و عدد **n** من الأعمدة يمكن الإشارة إليها بأنها مصفوفة **m x n** .

أى شئ فى الوجود له أسم ، حتى المصفوفات يتم تسميتها ، و لكن أسلوب التسميه يختلف من شئ إلى آخر ، فتسمية المصفوفات تتم باستخدام حروف كبيرة مثل **M** أو **W** . كل عنصر داخل المصفوفة له أسم تعريفى و يأخذ الشكل التالى 13 ، هذا الأسم يتكون من 3 أجزاء : اما الجزء الأول فهو يمثل الحرف الذى يعبر عن المصفوفة - الحرف ذو الخلفية الصفراء ، الجزء الثانى هو رقم يمثل رقم الصف - فى هذه الحالة هو الرقم 1 ، الجزء الثالث هو يمثل رقم العمود - فى هذه الحالة هو الرقم 3 .

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

يتم ضرب المصفوفات فى بعضها باستخدام **قاعدة الضرب الداخلى** و **قاعدة الضرب الخارجى**
Inner and Outer Dimension Rule

عند ضرب مصفوفتين يتم كتابة أبعاد المصفوفتين و بينهم رمز النجمة "*" كالتالى : $2 \times 3 * 3 \times 4$

البعد الداخلى لعملية الضرب يشترط أن يكون قيمة الأعمدة فى المصفوفة الأولى - ناحية اليسار - تساوى قيمة الصفوف فى المصفوفة الثانية - ناحية اليمين ، فى هذا المثال تكون $3 * 3$.

عملية ضرب مصفوفة فى مصفوفة ينتج أخرى جديدة ، يتم معرفة أبعاد المصفوفة الجديدة عن طريق عكس قاعدة الضرب الداخلى فبدلاً من أخذ أعمدة الأولى و صفوف الثانية يتم أخذ قيمة الصفوف من المصفوفة الأولى ، و يتم أخذ قيمة الأعمدة من المصفوفة الثانية ، لتصبح أبعاد المصفوفة الجديدة بعد الضرب هى 2×3 .

مثال :

فالفترض وجود المصفوفة $A(3 \times 3)$ و المصفوفة $B(3 \times 6)$ و نريد تطبيق عملية الضرب عليهم .

$$A * B = 3 \times 3 * 3 \times 6$$

عند إجراء عملية الضرب لابد و أن يتساوى البعد الداخلى للمصفوفتين و هو كما واضح من الخطوة السابقة متساوى . لذا فالمصفوفة الجديدة الناتجة من عملية الضرب أبعادها تساوى الأبعاد الخارجية للمصفوفتين ليكون 3×6 .

لاحظ أن عملية ضرب المصفوفة $A * B$ يمكن إتباعها على عكس $B * A$ حيث أن البعد الداخلى لهما حينئذٍ لن يكون متساوى .

يمكن فهم عملية ضرب المصفوفات بشكل أسهل عندما ننظر للشكل التالى حيث أننا نريد ضرب المصفوفتين A و B :

المصفوفة A	المصفوفة B
$a_{11} \ a_{12} \ a_{13}$	$b_{11} \ b_{12} \ b_{13}$
$a_{21} \ a_{22} \ a_{23}$	$b_{21} \ b_{22} \ b_{23}$
$a_{31} \ a_{32} \ a_{33}$	$b_{31} \ b_{32} \ b_{33}$

فى المصفوفة A قمنا بالتعامل مع كل صف على أنه متجه ، أيضاً المصفوفة B تعاملنا مع أعمدتها على أنها متجهات ، و حيث أن أبعاد المصفوفتين هو (3×3) لذا فكل متجه سيمثل بثلاثة أبعاد ، مرة أخرى نقول أن المتجهات قد تكون $1D$, $2D$, $3D$, $4D$, $5D$ وهكذا إلى ما لا نهاية . الطريقة التى سنتبعها فى ضرب هاتين المصفوفتين الآن يمكن تطبيقها على أى مصفوفتين لهما نفس البعد الداخلى . المصفوفة A تتكون من 3 متجهات (كل متجه يشكل صف) وهما $Vector1(a_{11}, a_{12}, a_{13})$ و $Vector2(a_{21}, a_{22}, a_{23})$ و $Vector3(a_{31}, a_{32}, a_{33})$. و المصفوفة B تتكون من 3 متجهات (كل متجه يشكل عمود) و هما $Vector1(b_{11}, b_{12}, b_{13})$ و $Vector2(b_{21}, b_{22}, b_{23})$ و $Vector3(b_{31}, b_{32}, b_{33})$. لنقوم بعملية الضرب و إنتاج المصفوفة الجديدة لابد و أن نقوم بحساب قيمة كل عنصر موجود فى تلك المصفوفة الجديدة و التى شكلها سيكون كالتالى :

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

نبدأ فى حساب قيمة العنصر m_{11} لأنه موجود بأول صف و أول عمود فى المصفوفة الجديدة - أطلقنا عليها **M** ،القيمة التى سيتم تخزينها فى هذا العنصر هى حاصل الضرب القياسى للمتجه 1 فى المصفوفة **A** و المتجه 1 فى المصفوفة **B** ، لأن رقم الصف لهذا العنصر 1 و المصفوفة التى متجهاتها تمثل بصفوف هى المصفوفة **A** لذا فإننا نأخذ المتجه 1 من المصفوفة **A** ، و نأخذ المتجه رقم 1 من المصفوفة **B** لأن متجهات تلك المصفوفة تمثل بأعمدة و رقم العمود للعنصر m_{11} هو 1 .

من السابق و باستخدام تحديد موقع العنصر m_{32} نعرف ما هما المتجهين الواجب تطبيق الضرب القياسى عليهما ، فالرقم 3 يشير للمتجه 3 فى المصفوفة **A** و الرقم 2 يشير للعمود 2 فى المصفوفة **B** لذا يمكن حساب قيمته كالتالى :

الصف الأخضر فى المصفوفة **B** $\odot$ الصف الأزرق فى المصفوفة **A**

بعد حساب قيمة العنصر m_{33} نحصل على مصفوفة بالشكل التالى :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{Row1} \odot b_{Col1} & a_{Row1} \odot b_{Col2} & a_{Row1} \odot b_{Col3} \\ a_{Row2} \odot b_{Col1} & a_{Row2} \odot b_{Col2} & a_{Row2} \odot b_{Col3} \\ a_{Row3} \odot b_{Col1} & a_{Row3} \odot b_{Col2} & a_{Row3} \odot b_{Col3} \end{bmatrix}$$

عملية الضرب القياسى سمحت لنا إختصار معادلة الناتج لكل عنصر ، فاللعنصر m_{11} قد نكتب المعادلة بالشكل التالى :

$$a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} \quad \text{أو بالشكل التالى :}$$

بعض الطرق المتبعة لضرب مصفوفتين تتم بكتابة أكواد قليلة باستخدام جملة مثل **for** و بشكل متداخل ، و لكننا لن نفعل ذلك هنا ، لأن عملية الضرب لابد و أن تكون أسرع ما يمكن لأنه يمكن إستدعائها الآلاف المرات لذا سنقوم بكتابة الكود كاملاً و قد تشعر أنه طويل بعض الشيء و لكنه سيكون أسرع من استخدام حلقات التكرار .

الكود التالى يقوم بضرب مصفوفتين 4x4 لأن مكتبات الـ **DirectX** تتعامل مع المصفوفات 4x4 :

MATRIX MatrixMultiply (MATRIX a, MATRIX b)

```
{
    MATRIX result;
    result.m11 = a.m11*b.m11 + a.m12*b.m21 + a.m13*b.m31 + a.m14*b.m41;
    result.m12 = a.m11*b.m12 + a.m12*b.m22 + a.m13*b.m32 + a.m14*b.m42;
    result.m13 = a.m11*b.m13 + a.m12*b.m23 + a.m13*b.m33 + a.m14*b.m43;
    result.m14 = a.m11*b.m14 + a.m12*b.m24 + a.m13*b.m34 + a.m14*b.m44;
    result.m21 = a.m21*b.m11 + a.m22*b.m21 + a.m23*b.m31 + a.m24*b.m41;
    result.m22 = a.m21*b.m12 + a.m22*b.m22 + a.m23*b.m32 + a.m24*b.m42;
    result.m23 = a.m21*b.m13 + a.m22*b.m23 + a.m23*b.m33 + a.m24*b.m43;
    result.m24 = a.m21*b.m14 + a.m22*b.m24 + a.m23*b.m34 + a.m24*b.m44;
    result.m31 = a.m31*b.m11 + a.m32*b.m21 + a.m33*b.m31 + a.m34*b.m41;
    result.m32 = a.m31*b.m12 + a.m32*b.m22 + a.m33*b.m32 + a.m34*b.m42;
    result.m33 = a.m31*b.m13 + a.m32*b.m23 + a.m33*b.m33 + a.m34*b.m43;
    result.m34 = a.m31*b.m14 + a.m32*b.m24 + a.m33*b.m34 + a.m34*b.m44;
    result.m41 = a.m41*b.m11 + a.m42*b.m21 + a.m43*b.m31 + a.m44*b.m41;
    result.m42 = a.m41*b.m12 + a.m42*b.m22 + a.m43*b.m32 + a.m44*b.m42;
    result.m43 = a.m41*b.m13 + a.m42*b.m23 + a.m43*b.m33 + a.m44*b.m43;
    result.m44 = a.m41*b.m14 + a.m42*b.m24 + a.m43*b.m34 + a.m44*b.m44;
    return result;
}
```

يمكن التعامل مع المتجه الثلاثى الأبعاد كأنه مصفوفة 1×3 ، و عند ضربه فى مصفوفة 3×3 يكون الناتج مصفوفة جديدة أبعادها 1×3 ، بضربنا متجه $3D$ فى مصفوفة 3×3 نحصل على متجه $3D$ جديد :

$$\text{Matrix A} \begin{bmatrix} x & y & z \end{bmatrix} \times \text{Matrix B} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \text{Matrix C} \begin{bmatrix} aRow1 \odot bCol1 & aRow1 \odot bCol2 & aRow1 \odot bCol3 \end{bmatrix} = \begin{bmatrix} X & Y & Z \end{bmatrix}$$

عملية الضرب السابقة قامت بتحويل المتجه $A \langle x, y, z \rangle$ للمتجه $C \langle X, Y, Z \rangle$

عملية ضرب متجه فى مصفوفة هى عملية مفيدة جداً ، لإننا قد نريد تحويل رؤوس مجسم معين ، أى تحجيمها و تحريكها و تدويرها لتتحول من نظام إحداثى معين إلى آخر – على سبيل المثال من المجال الخاص للمجسم **Local Space** للمجال العام للعبة **World Space** . فإذا كان لدينا المعادلات اللازمة لتنفيذ تلك التحويلات ، فإننا سنقوم بعمل مصفوفة لتحتوى على بيانات ذلك المجسم .

داخل اللعبة ، سنقوم بعمل مصفوفة لكل كائن ، تلك المصفوفة تصف إتجاه الكائن حول الثلاث محاور و موقعه داخل اللعبة . بعد ذلك ، سنقوم بتحويل رؤوس كائن ما من المجال الخاص للمجال العام عن طريق ضرب قيمة كل رأس فى تلك المصفوفة .

فالفترض أن لدينا متجه $2D \langle x, y \rangle$ و نريد تدويره بمقدار زاوية معينة θ حول نقطة الأصل لينتج لنا المتجه $\langle X, Y \rangle$. و كما ذكرنا سابقاً فإن تدوير المتجهات الـ $2D$ يماثل تدوير متجه $3D$ حول المحور Z ، و بذلك فإن قيمة المحورين x و y هما ما سيتغير ، و بالتالى قيمة المحورين بالمتجه الجديد ستأتى من المعادلة التالية :

$$\begin{aligned} X &= x * \cos(\theta) - y * \sin(\theta) \\ Y &= x * \sin(\theta) + y * \cos(\theta) \end{aligned}$$

فالفترض أن الفالفترض أن المتجه الذى سيدخل فى عملية الضرب أسمه V و أن همصفوفة 1×2 لذا:

$$V = \begin{bmatrix} x & y \end{bmatrix}$$

بما ان المتجه V يتكون من محورين لذا فإن المتجه الناتج بعد عملية التدوير سيتكون أيضاً من محورين ، و سنسمى المتجه الناتج C لذا :

$$C = \begin{bmatrix} X & Y \end{bmatrix}$$

المتجه V يتكون من عمودين و صف واحد أى 1×2 ، لذا المصفوفة التى ستضرب فى المتجه V لابد و أن يكو فيها صفان – راجع قاعدة البعد الداخلى . و لأن المتجه C يحتوى على عمودين لذا فالمصفوفة لابد و أن تكون 2×2 ، و سنطلق عليها أسم M و شكلها كالتالى :

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

و حيث أن $C = M * V$ لذا فباستخدام المصفوفة السابقة يكون :

$$\begin{bmatrix} x & y \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} X = V \odot mCol1 & Y = V \odot mCol2 \end{bmatrix}$$

X و Y هما العناصر التى يحتوى عليها المتجه C ، تذكر أن العمود الاول فى المصفوفة M يستخدم فى حساب قيمة X فى المتجه V ، و العمود الثانى فى المصفوفة M يستخدم فى حساب قيمة Y فى المتجه V لذا:

$$X = x * m_{11} + y * m_{21}$$

$$Y = x * m_{12} + y * m_{22}$$

و الآن فالنظر لمعادلات التدوير

$$X = x * \cos(\theta) - y * \sin(\theta)$$

$$Y = x * \sin(\theta) + y * \cos(\theta)$$

معادلات حساب قيمة X و Y و معادلات التدوير تبدو متشابهة ، لذا فإننا سنقوم بعملية إحلال و تبديل بين عناصر المعادلات الأربعة و سنبدأ بـ Y :

$$Y = x * m_{12} + y * m_{22}$$

$$Y = x * \sin(\theta) + y * \cos(\theta)$$

فى المصفوفة M ، سنقوم بتبديل العنصر m_{12} بـ $\sin(\theta)$ و تبديل العنصر m_{22} بـ $\cos(\theta)$ لتصبح المصفوفة كالتالى :

$$M = \begin{bmatrix} m_{11} & \sin(\theta) \\ m_{21} & \cos(\theta) \end{bmatrix}$$

و نفس الشئ سنفعله مع X :

$$X = x * m_{11} + y * m_{21}$$

سنقوم بمقارنة المعادلة السابقة بمعادلة التدوير التالية :

$$X = x * \cos(\theta) - y * \sin(\theta)$$

نلاحظ أن الإشارة فى المعادلة التدوير الخاصة بـ X مختلفة عن الإشارة فى معادلة الحصول على قيمة X ، لذا فسنقوم بإعادة ترتيب الإشارات فى معادلة التدوير كالتالى :

$$X = x * \cos(\theta) + y * -\sin(\theta)$$

و بالتالى :

$$X = x * m_{11} + y * m_{21}$$

$$X = x * \cos(\theta) + y * -\sin(\theta)$$

فى المصفوفة M ، سنقوم بتبديل العنصر m_{11} بـ $\cos(\theta)$ و تبديل العنصر m_{21} بـ $-\sin(\theta)$ لتصبح المصفوفة تحتوى على إحداثيات النقل و التدوير و شكلها كالتالى :

$$M = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

ضرب المصفوفة M فى المتجه V سينتج المتجه C و الذى يحتوى على X و Y و قد تم تدويرهما . أحد المزايا الموجودة هنا - سنستخدمها فى الدوال فيما بعد - هى أننا نستطيع بعد إدخال القيم فى المصفوفة ، أن نستدعى الدالة $\cos$ مرتين و الدالة $\sin$ مرتين (من الممكن إستدعاء الدوال الرياضية مرة واحدة فقط إذا أستخدمنا متغيرات محلية) و بعد ذلك نخزن البيانات داخل مصفوفة . قمنا بالخطوات السابقة لأننا قد نضرب هذه المصفوفة فى ألاف المتجهات لنقوم بتحويلهم ، فإستخدام هذا الأسلوب لن نضطر لإستدعاء الدوال $\sin$ و $\cos$ العديد من المرات و هذا يسرع عملية تنفيذ الكود المكتوب ، و أيضاً هذا الأسلوب يتيح لنا تحويل المجسم كله دفعة واحدة بدلاً من تحويل رؤوس كل مجسم على حدى كما كنا نفعل سابقاً .

مصفوفة M لتحويل متجه 3D أسمه V حول المحور Z و بزاوية θ بالقياس العشري

$$V = \begin{bmatrix} x & y & z \end{bmatrix} \times M = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = C = \begin{bmatrix} X = V \odot mCol1 & Y = V \odot mCol2 & Z = V \odot mCol3 \end{bmatrix}$$

تغييرات بسيطة ستحدث عند التحول للـ 3D عندما يتم التدوير حول المحور Z ، لأن قيمة X و Y فقط التى ستتغير و هذا يعنى أن قيمة المحور Z لن تتغير ، و بذلك يمكن إعتبار أن العمود الثالث فى المصفوفة M كأنه المتجه الذى يقوم بتحويل المحور Z ليظهر فى المتجه C .

1. تدوير المصفوفات حول المحور X

$$NewY = OldY \times \cos(\theta) - OldZ \times \sin(\theta)$$

$$NewZ = OldY \times \sin(\theta) + OldZ \times \cos(\theta)$$

$$\begin{bmatrix} 1 & 0 & 0 \\ \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \end{bmatrix}$$

2. تدوير المصفوفات حول المحور Y

$$NewX = OldX \times \cos(\theta) + OldZ \times \sin(\theta)$$

$$NewZ = OldX \times \sin(\theta) + OldZ \times \cos(\theta)$$

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

3. تدوير المصفوفات حول المحور Z

$$NewX = OldX \times \cos(\theta) - OldY \times \sin(\theta)$$

$$NewY = OldX \times \sin(\theta) + OldY \times \cos(\theta)$$

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

قد تعتقد أننا قمنا بملء العمود الثالث فى المثال السابق بصفر أثناء عملية التدوير لأننا لا نريد تغيير قيمة Z . فى الواقع الأمر ليس كذلك ، لأن قيمة Z فى المتجه C سيتم حسابها كالتالى :

$$Z = x * m_{31} + y * m_{32} + z * m_{33}$$

فمثلاً إذا كانت قيمة المحور z فى المتجه V تساوى 10 ، لذا فى هذه الحالة نريد أن نتأكد أن هذا الرقم لن يتغير أثناء عملية التدوير . أيضاً فى المتجه الناتج لابد و ان تكون قيمة المحور Z فيه تساوى 10 ، فإذا كان آخر عمود فى المصفوفة ممتلئ بالصفر ، لذا قد تحسب قيمة Z كالتالى :

$$Z = x * 0 + y * 0 + z * 0 = 0$$

ما نريده فعلاً هو جعل قيمة المحور Z ثابتة قبل التدوير و بعده ، لذا فسنضع الرقم 1 مكان العنصر m_{33} ، وبهذا قيمة المحور Z فى المتجه الناتج ستحسب كالتالى :

$$\therefore Z = x * 0 + y * 0 + z * 1 = Z$$

$$\therefore Z = x * 0 + y * 0 + 10 * 1 = 10$$

العمود الجديد فى المصفوفة يسمى بالعمود الأحادى **Identity Column** لأن القيمة الناتجة منه تساوى القيمة المدخلة إليه . باستخدام ذلك العمود سننشئ نوع خاص من المصفوفات يسمى بمصفوفة الوحدة **Identity Matrix** و سميت بهذا الاسم لأن قيمتها النهائية تساوى 1 :

$$I_{\text{Identity Matrix}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

فالفترض أننا لدينا متجه V و يساوى $[x \ y \ z]$ و مصفوفة وحدة تسمى بـ I . عند ضرب $V * I$ سنحصل على المتجه C و قيمته تساوى $[X \ Y \ Z]$ و حيث أن I مصفوفة وحدة لذا لابد أن يكون $C=V$:

$$\therefore V * I = C$$

$$\therefore [x \ y \ z] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x * 1 + y * 0 + z * 0 \quad x * 0 + y * 1 + z * 0 \quad x * 0 + y * 0 + z * 1]$$

و بحذف العناصر التى تم ضربها فى 0 نحصل على :

$$C = [X * 1 \ Y * 1 \ Z * 1]$$

مصفوفة الوحدة هي المصفوفة التي تضرب متجه معين في الرقم 1 ، سنقوم بتطوير هذه المصفوفة بحيث يتم ضرب المتجهات في أرقام أخرى بخلاف الرقم 1 . المصفوفة الجديدة تسمى بمصفوفة التحجيم الثابت **Uniform Scaling Matrix** و سميت بهذا الاسم لأنك تستبدل الرقم 1 بها بإى رقم آخر و الذى نريد أن نعيد تحجيم المتجه به . فمثلاً إذا كنا نريد إعادة تحجيم كل المتجهات بعشرة أضعاف حجمها الأصلي ، لذا فمصفوفة التحجيم ستكون بذلك الشكل :

$$10 \times \text{IdentityMatrix} = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

إذا قمنا بضرب رؤوس مجسم ما في المصفوفة السابقة ، فسيصبح الكائن بعشرة أضعاف حجمه الأصلي ، أيضاً يمكن أن ننشئ نوع آخر من المصفوفات يسمى بمصفوفة التحجيم المتغير **Non-Uniform Scaling Matrix** أو **Shearing Matrix** و سميت بهذا الاسم لأنك تستبدل الرقم 1 بها بإى رقم آخر و لا يشترط تساوى الأرقام بكل متجه .

$$\begin{bmatrix} 20 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 34 \end{bmatrix}$$

عملية ضرب المصفوفات هي عملية تجميعية حيث أن $(A(BC)) = (AB)C$. فمثلاً إذا كان لدينا المصفوفة A وكانت مجهزة للتدوير حول المحور Z ، و المصفوفة B وكانت مجهزة للتدوير حول المحور Y ، لذا فدمج المصفوفتين A و B في مصفوفة واحدة (سنسميها M) ستجد أنها تقوم بدور المصفوفتين معاً . و بالتالى إذا كان لدينا المتجه V و أردنا أن نطبق عليه التدوير حول Z و Y فبدلاً من تطبيق $V * A = C$ ثم $C * B = D$ ، سنقوم باستخدام أسلوب جديد و هو $M = A * B$ ثم $V * M = D$ و هو نفس التأثير الذى سنحصل عليه من $V * A = C$ ثم $C * B = D$. دمج المصفوفات باستخدام الأسلوب الجديد - المصفوفة M - يعنى أنه قد يكون لديك العديد من المصفوفات ، كل واحدة تقوم بعملية معينة ، و بدمجهم جميعاً في مصفوفة واحدة باستخدام ضرب المصفوفات ستنتج لك مصفوفة تقوم بجميع العمليات التى يقوموا بها منفردين . لذا فعملية دمج المصفوفات هي عملية تعطيك مرونة شديدة في تطبيق أساليب جديدة و بإقل مجهود .

عملية ضرب المصفوفات ليست تبادلية أى أن $A * B$ لا تساوى $B * A$. لذا فكن حذراً بالترتيب مهم . أهمية الترتيب رأيناها من قبل في الدرس السابق و هي تأثير تدوير رؤوس المجسمات قبل نقل إحداثياتها و الاختلاف الشديد بينها و بين نقل الإحداثيات يتبعه تدوير الرؤوس . فتدوير المجسم قبل نقل إحداثياته يعطينا الإحساس بأن الكائن يدور حول نفسه أولاً - أو حول مركزه - ثم يتحرك (و هو فى الأغلب التأثير الذى نريده) ، أما التدوير بعد نقل الإحداثيات يجعل الكائن يتحرك من مكانه أولاً ثم يدور حول نقطة الأصل للنظام الإحداثى العام للعبة . و حيث أننا سنقوم باستخدام المصفوفات لحفظ بيانات التحويلات - التدوير و نقل الإحداثيات و غيرهم - لذا فكن حذراً فترتيب الضرب فى تلك المصفوفات هام جداً .

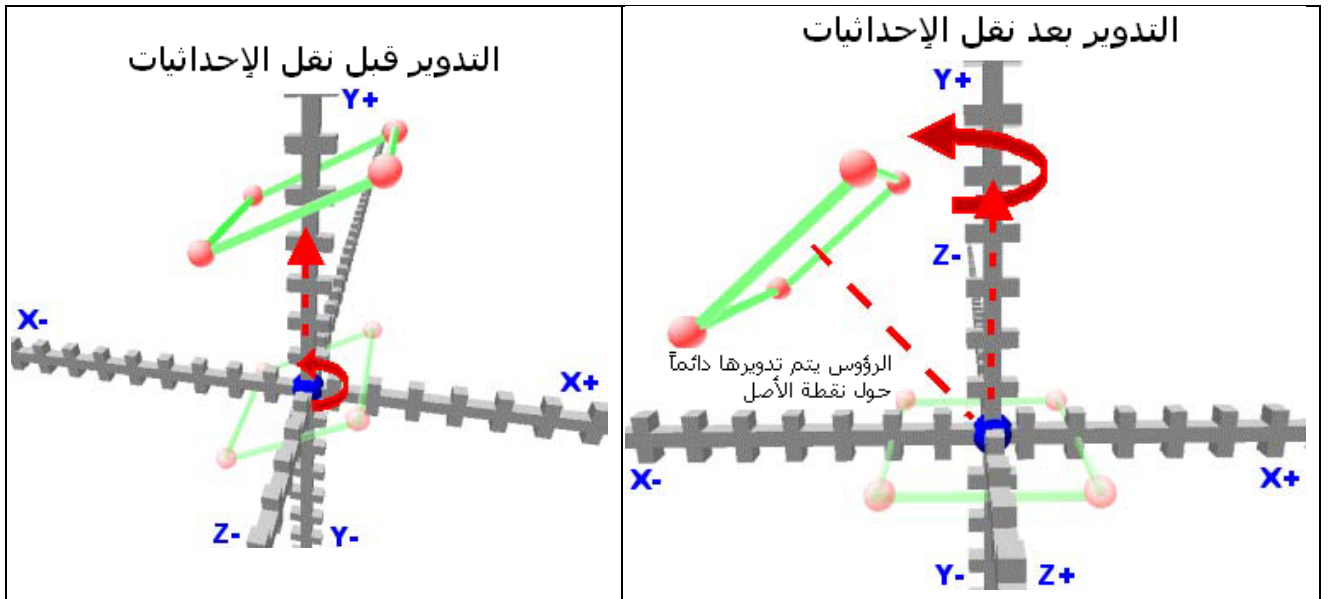


Fig 1.37

يتم ضرب متجه بمصفوفة عن طريق سلسلة من ضرب الصفوف بالأعمدة . لذا قد تقول أن المصفوفات تستخدم فقط فى التحويلات الخطية (حيث المخرجات تتناسب مع المدخلات) . و سنأخذ مصفوفة الوحدة كمثال على ذلك ، حيث تقوم بتحجيم المتجه الداخل إلى المتجه الخارج . عمليات التحجيم و التدوير هى تحويلات خطية حيث أن الناتج يمثل (وهو فى الأغلب متجه واحد فقط) العديد من المدخلات . و لكن ماذا يحدث إذا أردنا تصنيع مصفوفة و التى دائماً ما تنتج قيمة المحور X بزيادة 5 عليه على الرغم من قيمة المحور X فى المتجه المدخل للمصفوفة .

$$\begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} X & Y & Z \end{bmatrix}$$

فالتجاهل قيمة Y و Z فى المتجه الناتج الآن ، و لنركز فقط على كيفية حساب قيمة المحور X فى المتجه الناتج :

$$X = x * m_{11} + y * m_{21} + z * m_{31}$$

بسبب أن x و y و z يتم إستخدامهم فى حساب قيمة المحور X فى المتجه الناتج ، لذا فإنه من المستحيل وضع أى قيمة داخل المصفوفة لتنتج X دائماً زائدة بمقدار 5 .

لذا ، فإذا كان من المفترض أن يتحرك الكائن للإحداثى (50,70,10) فلا بد من صنع مصفوفة التى تقوم بنقل جميع رؤوس المجسم بـ 50 بمحاذاة المحور X ، و 70 بمحاذاة المحور Y ، و 10 بمحاذاة المحور Z .

قد يترأى لنا أن الحل الوحيد المتاح هو إستخدام مصفوفة 3D أولاً و الخاصة بالتدوير و بعد ذلك نقوم بضرب الناتج فى متجه الحركة كالتالى :

$$V' = VM + T$$

حيث أن V' و V هما المتجه الناتج و المتجه المدخل على التوالى ، M هى مصفوفة 3D خاصة بالتدوير ، T هو متجه الحركة و يمثل بمصفوفة 1×3 .

فى حين أن الأسلوب السابق السابق يعمل بكفاءة ، و لكننا نفضل إتمام العملية كلها بإستخدام مصفوفة واحدة فقط ، و لكن كيف سنقوم بصنع مصفوفة و التى تضيف للمتجه المدخل القيم 50 للمحور x ، 70 للمحور y ، 10 للمحور z على الرغم من وجود قيم داخل المتجه المدخل ؟ محاولة إتمام تلك العملية بإستخدام المصفوفة السابقة مستحيل ، قم بالنظر للعمود الأول بالمصفوفة السابقة و المسئول عن حساب قيمة المحور X فى المتجه الناتج ، ستجد أنه مستحيل تبديل قيمة m_{11} و m_{21} و m_{31} بأى قيمة أخرى ، لذا لابد من إيجاد حل لهذه المشكلة ، و الحل هنا يتمثل فى البعد الإضافى .

على الرغم أن البشر لا يستطيعوا رؤية ما هو أكثر من ثلاثة أبعاد ، إلا انه توجد فى الرياضيات العديد من الأبعاد . هذه الأبعاد (و بالخصوص البعد الرابع فى حالتنا هذه) تعطينا ما يمكننا من حل مشكلة الإحداثيات السابقة . قم بالنظر إلى المتجه رباعى الأبعاد التالى ، و ستجده كما توقعت يمثل بمصفوفة 1×4 :

$$\begin{bmatrix} x & y & z & w \end{bmatrix}$$

المتجه الـ 4D السابق يحتوى على الأبعاد x و y و z كما فى المتجه الـ 3D الذى تعاملنا معه من قبل بالإضافة إلى بعد رابع أسمه w . إذا قمنا بقسمة المتجه الـ 4D على بعده الرابع سنجد انه أصبح لدينا متجه 3D كما يلى :

$$\begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} & \frac{w}{w} = 1 \end{bmatrix}$$

فى الواقع المتجه الـ 4D حيث $w=1$ هو متجه 3D

$$\begin{bmatrix} x & y & z \\ 1 & 1 & 1 \end{bmatrix} w = 1$$

هذا النوع من المتجهات حيث البعد الرابع يساوى 1 يسمى بنظم الإحداثيات المحايدة **Homogeneous Coordinates**. عملية قسمة المحاور x و y و z على w تسمى بمحايدة الإحداثيات. هذه العملية تقوم بتحويل المتجه ذو N بعد لمتجه ذو $N-1$ بعد. الإحداثيات المحايدة لا يتم تطبيقها على المتجهات الـ 4D فقط و لكنها متاحة لـ 1 بعد. فإذا أردنا أن نحيد متجه 3D يسمى V سنقوم بالتالى :

$$V = (x, y, z)$$

$$V = (x/z, y/z, z/z)$$

المتجه R الناتج من تلك العملية سيكون كالتالى :

$$R = (x, y, 1)$$

العملية السابقة قامت بتحويل المتجه من فضاء 3D إلى فضاء 2D. قد تتذكر هذا الأسلوب عندما إتبعناه فى الدرس السابق. فالمعادلة التى تقسم x و y على z تستخدم فى تحويلات الرؤية من 3D إلى 2D و عند $z=1$.

لذا لمحايدة متجه يحتوى على N بعد نقوم بقسمة كل أبعاده على البعد الأخير له، هذه العملية تقوم بتحويل المتجه من N بعد إلى $N-1$ بعد.

باستخدام متجه 4D محايد سنقوم بتجاهل البعد الرابع w حيث أنه يساوى 1، و لكن توجد بعض الإحيان التى سيكون البعد الرابع فيها لا يساوى 1 و هذه الحالات سيتم شرحها فى حينها.

و لكن كيف يقوم هذا الأسلوب بحل مشكلة تمثيل هذا المتجه داخل مصفوفة ؟ الحل يكمن فى أن البعد $w=1$ بالإسفل تجد متجه 4D تم ضربه فى مصفوفة وحدة 4×4 أسماها I لتنتج لنا متجه 4D يسمى R . لاحظ أنه لا بد من إضافة عمود جديد للمصفوفة لنقوم من خلاله بحساب قيمة w . و فى حال أن $w=1$ (ونحن نعرف انها كذلك) لذا فإن ضرب متجه ما بأى مصفوفة سينتج دائماً متجه حيث w به تساوى 1.

$$V = \begin{bmatrix} x & y & z & w = 1 \end{bmatrix} \times I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R = \begin{bmatrix} X & Y & Z & W = 1 \end{bmatrix}$$

باستخدام تلك المعلومة سنقوم بإنشاء مصفوفة جديدة 4×4 تحتوى على قيم التدوير حول المحور Z و شكلها كالتالى :

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

المصفوفة السابقة مماثلة تماماً لما انشأناها من قبل - المصفوفة الـ 3D الخاصة بالتدوير حول Z - باستثناء أن آخر عمودين أصبحوا أعمدة وحدة، و تذكر أنا نريد فقط قيمة x و y من المتجه المدخل لينم تطبيق عمليات التدوير عليهم. لذا قيمة المحورين w و z لا بد و أن تأخذ كما هى و توضع فى المتجه الناتج دون تغيير (لأنه يتم ضرب قيمه كل منهم فى 1). عمود الوحدة الجديد سيقوم بإخذ قيمة المحور w من المتجه المدخل و وضعها داخل المحور w فى المتجه الناتج دون تغيير :

$$W = x * 0 + y * 0 + z * 0 + w * 1 = w$$

لاحظ أيضاً أننا قمنا بإضافة صف رابع للمصفوفة، و قد قمنا بذلك حتى نستطيع ضرب المتجه 1×4 بالمصفوفة 4×4 (راجع قاعدة البعد الداخلى). الصف الرابع يسمح لنا بتمثيل قيم الإحداثيات الجديدة داخل مصفوفتنا. بإلقاء نظرة على المصفوفتين التاليتين و اللتان يمثلوا عملية ضرب المتجه V بالمصفوفة M :

$$V[x \ y \ z \ 1] \times M \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = R[X \ Y \ Z \ 1]$$

لنقوم بحساب قيمة المحور X فى المتجه R الناتج من العملية السابقة:

$$X = Vx * m_{11} + Vy * m_{21} + Vz * m_{31} + m_{41}$$

و لإننا نعرف أن w دائماً ستساوى 1 ، لذا فأخر جزء من المعادلة السابقة سيكون دائماً $1 * m_{41}$. بمعنى آخر، أياً كانت القيمة الموجودة فى m_{41} سيتم إضافتها دائماً إلى المحور X فى المتجه الناتج.

و بالمثل أيضاً للأعمدة الخاصة بالمحاور y و z داخل المصفوفة. فأى قيمة يتم تخزينها داخل m_{42} سيتم إضافتها للمحور Y فى المتجه الناتج، أيضاً أى قيمة موجودة بـ m_{43} سيتم إضافتها للمحور Z فى المتجه الناتج. لذا فالصف الرابع داخل المصفوفة بالإضافة إلى البعد الرابع داخل المتجهين يسمحوا لنا بتمثيل معادلات الحركة داخل المصفوفة لإنهم معاً يمكن جمعهم لإعطائنا التأثير المطلوب. قد تفكر فى الصف الرابع بالمصفوفة كأنه جزء مستقل عن المصفوفة و الذى لايقوم بتطبيق أى عمليات على المتجهات كما يحدث من باقى المصفوفة و التى بذلك أصبحت أبعادها 3×3 ، هذا الجزء المنفصل سنستخدمه لإضافة او طرح القيم من المتجه المدخل للمصفوفة.

لذا، و باستخدام تلك المعلومات سنقوم ببناء مصفوفة تقوم بتحريك المتجه بمقدار TX حول المحور X ، و بمقدار TY حول المحور Y ، و بمقدار TZ حول المحور Z كما يلى :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ TX & TY & TZ & 1 \end{bmatrix}$$

فعلى سبيل المثال إذا افترضنا أن $TX=200$ ، $TY=0$ ، $TZ=-50$ ، المتجه المدخل V يساوى $(200,70,500)$ ، نحتاج كخطوة أولى تحويل المتجه V ليصبح متجه $4D$ كالتالى :

$$V = (200, 70, 500, 1)$$

$$V[200 \ 70 \ 500 \ 1] \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 200 & 0 & -50 & 1 \end{bmatrix} = R[X \ Y \ Z \ 1]$$

سنقوم بحساب قيمة R بأنفسنا كالتالى:

$$Rx = 200*1 + 70*0 + 500*0 + 1*200 = 400 \quad (Rx=400)$$

$$Ry = 200*0 + 70*1 + 500*0 + 1*0 = 70 \quad (Ry=70)$$

$$Rz = 200*0 + 70*0 + 500*1 + 1*-50 = 450 \quad (Rz=450)$$

$$Rw = 200*0 + 70*0 + 500*0 + 1*1 = 1 \quad (Rw=1)$$

$$\therefore R = [400 \ 70 \ 450 \ 1]$$

عملية تحريك النقطة تمت بنجاح باستخدام المصفوفة الرباعية. يمكن دمج مصفوفة التحريك السابقة باى عدد من المصفوفات الأخرى. كمثال يمكن دمج مصفوفة التحريك مع مصفوفة التدوير و بعد ذلك إدخال جميع المتجهات للمصفوفة النهائية ليتم تحويلهم.

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ TX & TY & TZ & 1 \end{bmatrix}$$

لقد عرفنا حقيقة أن المحور w فى المتجه المدخل دائماً ما يساوى 1 عند إدخاله للمعادله. لذا سنقوم بالإستفادة من هذه المعلومة فى صنع دالة تقبل متجه $3D$ و مجموعة من المصفوفات الـ $4D$ و التى تستخدم فى الدور و التحريك و التحجيم ... إلخ. الدالة سنسميها **VectorMatrixMultiply**، و هى تأخذ متجه $3D$ كمعامل و تحفظه داخلها على هيئة متجه $4D$ حتى تستطيع إتمام العملية الحسابية. هذه الدالة تقوم بتوحيد المتجه العوده بمتجه $3D$ و ذلك عن طريق قسمة x و y و z على w .

```
bool VectorMatrixMultiply (VECTOR3D& vDest, VECTOR3D& vSrc, MATRIX& Mat)
{
    float x = vSrc.x*mat.m11 + vSrc.y*mat.m21 + vSrc.z* mat.m31 + mat.m41;
    float y = vSrc.x*mat.m12 + vSrc.y*mat.m22 + vSrc.z* mat.m32 + mat.m42;
    float z = vSrc.x*mat.m13 + vSrc.y*mat.m23 + vSrc.z* mat.m33 + mat.m43;
    float w = vSrc.x*mat.m14 + vSrc.y*mat.m24 + vSrc.z* mat.m34 + mat.m44;

    // Prevent Divide by 0.
    if (fabsf (w) < g_EPSILON) return false;

    // homogenize the coordinate.
    vDest.x = x / w;
    vDest.y = y / w;
    vDest.z = z / w;

    return true;
}
```

تقوم بعض الدوال التى تتعامل مع المصفوفات بضرب متجه $3D$ بمصفوفة $4D$ ، و لكننا نعلم أنه لا يمكن حدوث ذلك بسبب قاعدة البعد الداخلى. و لكن كل ما تفعله هذه المصفوفات هو تقريباً ما قمنا نحن بفعله أى التعامل مع المتجه الـ $3D$ كأنه $4D$ و قيمة المحور w به تساوى 1. فى بعض الأحيان لن نريد التعامل مع المحور w على أنه يحتوى على القيمة 1 و بالتالى فإن استخدام دالة تأخذ متجه $4D$ سيكون إلزامى.

لكل محور موجود فى المتجه الناتج من عملية الضرب قمنا بإضافة قيمة الصف الرابع بالمصفوفة له. و بعد ذلك قمنا بمحايدة المتجه الناتج ليصبح $3D$ عن طريق القسمة على المحور w . يمكن تعديل الكود السابق ليعود لنا بمتجه $4D$ بدلاً من $3D$ عن طريق إتمام عملية القسمة على w بشكل مختلف. نقطة أخرى و هى إذا كنا نعرف على وجه اليقين أن العمود الرابع فى المصفوفة هو عمود وحدة (و نحن نعرف أنه كذلك)، فإن المحور w فى المتجه الناتج سيكون دائماً يساوى 1، و بهذا فإنه يمكن تجاهل عملية حساب قيمة w و يمكن تجاهل القسمة على w حيث أنه لا داعى منها (لأن w تساوى 1). هذا الأسلوب سوف نستخدمه لاحقاً عند التعامل مع مواقع الرؤوس.

ال **Quaternion** هى نظام أعداد مركب تم تعريفه باستخدام متجه **3D** سنسميه **V** و قيمته $\langle x, y, z \rangle$ و عدد سنسميه **S** و قيمته **w**. ال **Quaternion** الإحدى الطول يعطينا إمكانية حفظ بيانات التدوير بأسلوب جديد و مختلف. المتجه **V** السابق يمثل التدوير حول محور معين و قد تم حفظه على شكل متجه وحده تم تدويره باستخدام الدالة **Sin** لنصف الزاوية θ (بالنظام العشري). القيمة العددية **S** تم حفظها باستخدام الدالة **Cos** لنصف الزاوية θ .

$$q = (S, V)$$

$$q = [w, \langle x, y, z \rangle]$$

$$q = [\cos(\theta/2), |V| * \cos(\theta/2)]$$

توجد العديد من الميزات لدى ال **Quaternion** تجعل إستخدامها فى تمثيل بيانات التدوير أفضل من استخدام المصفوفات. و من أكثر مميزات ال **Quaternion** هو حجزها لمساحة ذاكرة أقل من المصفوفات حيث تحجز 16 بايت على عكس المصفوفات 36 بايت. أيضاً عملية دمج ال **Quaternion** أسرع كثيراً من عملية دمج المصفوفات. و حيث أن رؤوس المجسمات لا يمكنها التعامل مع ال **Quaternion** بشكل مباشر، فلا بد من تحويل ال **Quaternion** إلى مصفوفة. أيضاً ميزة أخرى فى ال **Quaternion** و هى أنه يمكن تحويلها للمصفوفات بشكل سريع جداً، و سنجد كلما تقدما فى هذا الكورس أن ال **Quaternion** هى أفضل حل لتمثيل البيانات الخاصة بعمليات التدوير.

لتحويل ال **Quaternion** إلى مصفوفة 3×3 نقوم باستخدام المعادلة التالية:

$$M = \begin{bmatrix} 1 - 2y^2 - 2x^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wz & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

و حيث أن ال **Quaternion** هى نظام أعداد مركب، لذا فهى تتكون من جزء خيالى و جزء حقيقى. و لكننا فى هذا المورس غير مطالبين بأن نكون على دراية بنظام الأعداد المركبة للتعامل مع ال **Quaternion**. المعادلة السابقة هى كل ما نحتاج لفهمه حتى يمكننا إنشاء و استخدام ال **Quaternion** داخل برامجنا، و حتى يمكننا تحويل ال **Quaternion** إلى مصفوفات لإستخدامها فى التحويلات.

وثائق المساعدة الخاصة بالـ **DirectX** - تسمى **DirectX SDK** - تأتي بمجموعة من الدوال التي تساعد في الرسومات الـ **3D** و تسمى **Direct3D Extension** أو **D3DX**. و هي تحتوي على مجموعة من الدوال و الكائنات التي تتعامل مع الرياضيات و التي ستكون ذات فائدة لنا. سنقوم باستخدام تلك الدوال عن طريق كتابة الكلمة `<d3dx9.h> #include` داخل أعلى ملف الكود. ستجد أن معظم كائنات **D3DX** قد قامت بإعادة تعريف العلامات الرياضية و استخدام الـ **constructors** حتى تسهل عملية التعامل معها. سنقوم باستخدام دوال الكائن **D3DX** و الكائنات الخاصة بالمتجهات و المصفوفات في أول مشروع لنا. و هذا لأن جميع تلك الدوال تستغل جميع مميزات المعالجات الجديدة مثل **MMX**.

الكائن D3DXMATRIX

تتعامل مكتبات الـ **DirectX** مع المصفوفات الـ 4×4 . أما عناصر المصفوفات فيتم التعامل معها باستخدام طريقتين، أولهما و نتعامل فيها مع المصفوفات كأننا نتعامل مع **Array** حجمها 4×4 ، والأخرى باستخدام أسلوب موقع العنصر داخل المصفوفة و يتم بالإستعانة بالشكل التالي: **row_column**. و هذا يعنى أنك قد تجعل العنصر الموجود في الصف الرابع و العمود الثانى يأخذ القيمة 9 كالتالى :

```
D3DXMATRIX mat;  
mat._42=9.0;
```

و باستخدام العلامات الرياضية التي تم إعادة تعريفها يمكن إتمام عمليات الجمع و الطرح و الضرب كالتالى :

```
D3DXMATRIX mat1, mat2, mat3;  
mat3 = mat1 * mat2; // multiplication  
mat3 = mat1 + mat2; // addition  
mat3 = mat1 - mat2; // subtraction
```

يوجد أيضاً مشيدان **Constructors** للكائن **D3DXMATRIX** أولهما يقوم بتحديد القيم المبدئية للمصفوفة عن طريق تمرير أسم مصفوفة أخرى كمعامل. و الثانى يسمح لنا بإدخال قيمه الـ 16 عنصر يدوياً. و شكل المشيدان كالتالى:

1. `D3DXMATRIX (const D3DXMATRIX&);`
2. `D3DXMATRIX (FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _44,
FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);`

هذا الكائن يتعامل فقط مع المتجهات الـ **3D**، و أيضاً يوجد كائن للتعامل مع المتجهات الـ **2D** و آخر للـ **4D**. و جميعهم يحتوا على العلامات الرياضية و المشيديات التى تتيح التعامل مع المتجهات. و هذا يعنى اننا نستطيع اجراء عمليات الجمع و الطرح و الضرب إلخ بإستخدام العلامات الرياضية كالتالى:

```
D3DXVECTOR3 MyVec (20.0f, 50.0f, -10.0f);
```

```
D3DXVECTOR3 TheVec (0.0f, -2.0f, -2.25f);
```

```
MyVec = MyVec + TheVec;           // addition
MyVec = MyVec * TheVec;           // multiplication
MyVec = MyVec - TheVec;           // subtraction
MyVec = (TheVec * 2) + MyVec;     // combination
```

يمكن تعديل قيم عناصر المتجه كالتالى:

```
MyVec.y += 10.0f;
```

```
MyVec.y = TheVec.x;
```

للمتجهات الـ **2D** يمكن إستخدام الكائن **D3DXVECTOR2**، و للمتجهات الـ **4D** يمكن إستخدام الكائن **D3DXVECTOR4**.

يوجد أيضاً داخل **D3DX** كائن يتعامل مع المسطحات ويحتوى على 4 عناصر. أول ثلاثة عناصر يستخدموا لحفظ بيانات إتجاه دوران المسطح (x, y, z). العنصر الرابع يستخدم فى حفظ المسافة بين نقطة الأصل و المسطح.

```
D3DXPLANE MyPlane;
MyPlane.a = Normal.x;
MyPlane.b = Normal.y;
MyPlane.c = Normal.z;
MyPlane.d = dist;
```

و للسهولة سنقوم باستخدام مشيد الكائن:

```
D3DXPLANE MyPlane (Normal.x, Normal.y, Normal.z, dist);
```

تقدم لنا مكتبات **D3DX** مجموعة من الدوال التى تساعد فى إنشاء المسطحات. الدالة **D3DXPlaneFromPointNormal** تستخدم فى إنشاء مسطح عن طريق تمرير متجه دوران المسطح و أى نقطة على هذا السطح كمعاملات للدالة، و فى حال إستخدام متجه إتجاه السطح، سنقوم بتمرير متجه دوران المسطح و أى رأس من رؤوس السطح.

```
D3DXPlaneFromPointNormal (D3DXPLANE* pOut,
                          D3DVECTOR3* pPoint,
                          const D3DVECTOR3* pNormal);
```

يتم تمرير مؤشر لكائن من نوع **D3DXPLANE** للدالة كأول معامل و هو الذى سيستقبل نتيجة الدالة و أيضاً يتم تمرير مؤشرين من نوع **D3DXVECTOR** و يمثلوا متجه دوران المسطح و أى نقطة معروفة عليه. فالنفترض أنك لم تستطع الحصول على متجه دوران السطح و تريد ان يتم حسابه نيابة عنك، فى تلك الحالة قم باستخدام الدالة **D3DXPlaneFromPoints**. هذه الدالة تستخدم فى إنشاء مسطح عن طريق تمرير أى 3 نقط موجودين به. فعلى سبيل المثال، إذا كنت تريد إنشاء مسطح لسطح ما، ستقوم بتمرير أى 3 نقط موجودة بذلك السطح للدالة. حينئذ ستقوم هذه الدالة بحساب متجه دوران المسطح و المسافة بينه و بين نقطة الأصل. القيمة العائدة من الدالة ستكون مؤشر لكائن **D3DXPLANE** عن طريق **pOut**.

```
D3DXPlaneFromPoints (D3DXPLANE* pOut,
                    const D3DXVECTOR3* pV1,
                    const D3DXVECTOR3* pV2,
                    const D3DXVECTOR3* pV3);
```

قد تجد أن الـ **Quaternion** صعبة الإستيعاب قليلاً حيث أنها نظام أعداد مركب، و لكن مع مكتبات الـ **D3DX** ستجد أنها أسهل كثيراً و أبسط .

الكائن **D3DXQUATERNION** يحجز مساحة 16 بايت لـ 4 عناصر هما **(x, y, z, w)**. و كما ذكرنا سابقاً، سنقوم بتحويل الـ **Quaternion** إلى مصفوفة لإستخدامها فى تدوير رؤوس الكائنات. فعلى سبيل المثال، لإنشاء **Quaternion** لحفظ زاوية أستدارة قدرها 45 درجة حول المحور **X** سنقوم بالآتى:

```
float angle = 0.785.0f;
D3DXVECTOR3 axis (1, 0, 0);
axis *= (float) sin (angle/2.0f);
D3DXQUATERNION q;
q.x = axis.x;
q.y = axis.y;
q.z = axis.z;
q.w = (float) cos (angle/2.0f);
```

قبل تحويل الـ **Quaternion** إلى مصفوفة لابد و أن نتأكد أولاً أنه تمت تسويته. مكتبات الـ **D3DX** تعطينا دالة تسمى **D3DXQuaternionNormalize** و هى تستخدم فى تسوية الـ **Quaternion**، لذا فإستكمالاً للمثال السابق:

```
D3DXQUATERNION normalized_q;
D3DXQuaternionNormalize (&normalized_q, &q);
```

لدينا الآن الـ **Quaternion** وحدة، سنبدأ فى عملية تحويله إلى مصفوفة. مكتبات الـ **D3DX** لديها دالة تسمى **D3DXMatrixRotationQuaternion** و هى تستخدم لهذا الغرض.

```
D3DXMATRIX rotation_matrix;
D3DXMatrixRotationQuaternion (&rotation_matrix, &normalized_q);
```

لدينا الآن مصفوفة تم صنعها من **Quaternion**. الرياضيات الخاصة بالـ **Quaternion** تتعامل مع نظام اليد اليمنى، و حيث أن مكتبات الـ **D3DX** تتعامل مع نظام اليد اليسرى فلابد من عكس المصفوفة **rotation_matrix** قبل إستخدامها، و لعكس المصفوفة سنقوم بإستخدام الدالة التالية:

```
D3DXMatrixTranspose (&rotation_matrix, &rotation_matrix);
```

البديل الأسرع من عكس المصفوفة هو عكس الـ **Quaternion** قبل تحويله لمصفوفة، هذه العملية ستقوم بعكس الـ **Quaternion** و تسويته فى نفس الوقت، لذا فسنقوم بالإستغناء عن عملية التسوية السابقة و أيضاً لن نستخدم دالة عكس المصفوفة:

```
D3DXQUATERNION normalized_q;
D3DXQuaternionInverse (&normalized_q, &q);
```

```
D3DXMATRIX rotation_matrix;
D3DXMatrixRotationQuaternion (&rotation_matrix, &normalized_q);
```

تعطينا **D3DX** دالة تقوم بضرب مصفوفتين:

```
D3DXMatrixMultiply (D3DMATRIX* pOut,  
                    const D3DMATRIX* pM1,  
                    const D3DMATRIX* pM2);
```

تأخذ هذه الدالة مؤشرات المصفوفتين ثم تقوم بضربهما و تعود بالمصفوفة الناتجة فى **pOut**. عملية الضرب السابقة ستستخدم أى إمكانيات متاحة داخل الـ **CPU** لتسريع عملية الضرب.

تعطينا أيضاً مكتبات الـ **D3DX** مجموعة من الدوال لبناء مصفوفة تدوير معينة:

```
D3DMATRIX* D3DXMatrixRotationX (D3DMATRIX* pOut, float angle);  
D3DMATRIX* D3DXMatrixRotationY (D3DMATRIX* pOut, float angle);  
D3DMATRIX* D3DXMatrixRotationZ (D3DMATRIX* pOut, float angle);
```

الدوال الثلاث السابقة تأخذ مؤشر لكائن **D3DMATRIX** و زاوية **angle** بالنظام العشري و تمثل زاوية التدوير حول محور معين. فعلى سبيل المثال، إذا أردت بناء مصفوفة تقوم بتدوير المتجهات بزاوية عشرية مقدارها 1.3 حول المحور **Y** سنقوم بالآتى:

```
D3DMATRIX RotMatY;  
D3DXMatrixRotationY (&RotMatY, 1.3);
```

توجد أيضاً دالة تقوم بتغيير الإحداثيات من مكان لآخر:

```
D3DXMatrixTranslation (D3DMATRIX* pOut, float x, float y, float z);
```

فمثلاً إذا أردنا تحريك مجسم ليصل للنقطة (10,40,50) داخل اللعبة سنفعل الآتى:

```
D3DXMATRIX TraMat;  
D3DXMatrixTranslation (&TraMat, 10, 40, 50);
```

باستخدام الدوال السابقة نستطيع إعطاء كل مجسم مصفوفة خاصة به، و عند إكساء ذلك المجسم نقوم بضرب كل رؤوسه فى تلك المصفوفة ليتم تحويلهم جميعاً للمجال العام للعبة. فعلى سبيل المثال، لنقول أنه لدينا مجسم نريد تدويره بمقدار 2.0 بالنظام العشري حول المحور **Z**، و وضعه فى الإحداثى (10,50,2) داخل المجال العام للعبة. لنفعل ذلك سنقوم أولاً ببناء مصفوفة نقل الإحداثيات أولاً، و بعد ذلك إنشاء مصفوفة التدوير، و بعد ذلك نقوم بدمج مصفوفة التدوير مع نقل الإحداثيات لإحداث التأثير الذى نريده:

```
CObject Obj; // we assume that we have a class to represent our objects.  
D3DXMATRIX RotMat, TraMat;
```

```
D3DXMatrixTranslation (&TraMat, 10, 50, 2);  
D3DXMatrixRotationZ (&RotMat, 2);
```

```
Obj.WorldMatrix = RotMat * TraMat;
```

يحتوى المجسم الآن على مصفوفة تمثل مكانه و اتجاه دورانه داخل اللعبة، هذه المصفوفة سنقوم باستخدامها لتحويل رؤوس المجسم للمجال العام للعبة. و أيضاً عن طريق تعديل قيمها نستطيع تحريك المجسم. فعلى سبيل المثال، قد نضع دالة نقوم بإستدعائها فى كل مرة نقوم فيها بالضغط على زر الإتجاه لليمين، هذه الدالة ستقوم ببناء مصفوفة تدوير حول المحور **Y** بمقدار 0.2 بالنظام العشري و بعد ذلك ستقوم بضرب تلك المصفوفة بمصفوفة المجال العام للمجسم. و هذا يجعل الكائن يدور فى كل مرة يتم فيها الضغط على الزر بمقدار 0.2 وحدة عشرية.

فى الكود السابق لاحظ أن ترتيب عملية الضرب مهم. حيث يتم فى هذا الكود التدوير أولاً حول نقطة أصل المجسم يليها تحريكه إلى المكان الجديد داخل اللعبة.

الدالة التالية تسمح لك بتحديد زاوية التدوير حول المحاور **X** و **Y** و **Z** دفعة واحدة. لتعرف قيمة تلك الدالة سنفترض عدم وجودها و أننا نريد تدوير مجسم حول المحاور الثلاثة فماذا سنفعل، سنقوم ببناء 3 مصفوفات للثلاث محاور، مصفوفة لكل محور و بعد ذلك نضربهم جميعاً بنفس ترتيب كتابة أسمهم بالأعلى:

```
D3DXMatrixRotationYawPitchRoll (D3DXMATRIX* pOut, float Yaw, float Pitch, float Roll);
```

لنقوم بتدوير مجسم حول المحور **X** بمقدار 1 وحدة عشرية، و حول المحور **Y** بمقدار 2 وحدة عشرية، و حول المحور **Z** بمقدار 0.5 وحدة عشرية، يليها تحريك المجسم للإحداثى (100,50,-20) سنستخدم الكود التالى:

```
D3DXMATRIX RotMat, TraMat;
```

```
D3DXMatrixRotationYawPitchRoll (&RotMat, 1, 2, 0.5);
```

```
D3DXMatrixTranslation (&TraMat, 100, 50, -20);
```

```
Obj.WorldMatrix = RotMat * TraMat;
```

عن طريق تغيير قيم **Yaw** و **Pitch** و **Roll** لكل مجسم، سيتم تدوير المجسم حول المحاور الثلاثة. بعد بناء مصفوفة التدوير سنقوم بإعادة رسم الكائنات على الشاشة لتظهر فى موضعها الجديد.

تحتوى أيضاً مكتبات الـ **D3DX** على دوال تستخدم لضرب المتجهات بالمصفوفات. سنحتاجها عند ضرب رؤوس المجسم بمصفوفته العامة. توجد 3 دوال تستخدم لذلك الغرض و كل واحدة مختلفة عن الأخرى:

```
1. D3DXVec3TransformCoord (D3DXVECTOR3* pOut,  
                           const D3DXVECTOR3* pV,  
                           const D3DXMATRIX* pM);
```

الدالة السابقة تضرب متجه **3D** فى مصفوفة **4x4**. و كما عرفنا سابقاً فإن هذه الدالة تتعامل مع المتجه الـ **3D** كأنه متجه **4D** محايد و يأخذ الشكل (x, y, z, 1). تقوم الدالة بضرب المصفوفة بالمتجه و بعد ذلك تقوم بمحايدة الناتج إلى نظام **3D** ثم تعود به كناتج عملية الضرب. الدالة السابقة سنستخدمها فى ضرب رؤوس المجسم فى مصفوفته العامة.

```
2. D3DXVec3TransformNormal (D3DXVECTOR3* pOut,  
                            const D3DXVECTOR3* pV,  
                            const D3DXMATRIX* pM);
```

تستخدم الدالة السابقة فى تحويل المتجهات العادية إلى متجهات وحدة. فعلى سبيل المثال، لنفترض أن لدينا سطح وجهه لأسفل فى الاتجاه الموجب للمحور **Z**. متجه اتجاه وجه ذلك السطح يساوى (1,0,0)، فإذا تم تدوير ذلك السطح بمقدار 45 درجة فى اتجاه المحور **Y**، فسيتم تحديث قيم متجه اتجاه ذلك السطح. الاختلاف بين تحديث قيم متجه وحدة و متجه عادى هى أنه متجه الوحدة يظل كما هو - متجه وحدة - بعد ضربه فى المصفوفة.

عندما تحتوى المصفوفة على بيانات تحريك المجسم - و هى فى الأغلب ستكون كذلك بالنسبة لمصفوفة الكائنات - فلا بد حينها أن يتم تحديث بيانات المتجه. و كنتيجة لذلك، فلا داعى لأن تكون رأس المتجه على بعد وحدة واحدة من نقطة الأصل. لذلك سنقوم بتجاهل الصف و العمود الرابع بالمصفوفة و بعدها سنضرب المتجه بها بعد أن أصبحت أبعادها **3x3**. مكتبات **D3DX** لديها دالة تقوم بذلك:

3. D3DXVec3Transform (D3DXVECTOR4* pOut, **const** D3DXVECTOR3* pV,
const D3DXMATRIX* pM);

الدالة السابقة تأخذ متجه 3D و مصفوفة و المتجه الناتج أبعاده 4D. المتجه المدخل للدالة يتم التعامل معه كأنه 4D و يكون على الشكل (x, y, z, 1). أما المتجه الناتج فيأخذ الشكل (x, y, z, w) حيث w لا تساوى 1. و على عكس الدالة D3DXVec3TransformCoord، هذه الدالة لا تقوم بتوحيد المتجه الناتج بقسمة x و y و z على w. الدالة السابقة قد نحتاجها فى أن العمود الرابع بالمصفوفة ليس بعمود وحدة.

مكتبات D3DX تحتوى أيضاً على مجموعة من الدوال التى تقوم بتسوية المتجهات و تطبيق الضرب القياسى و الإتجاهى عليهم و أيضاً الحصول على أطوالهم.

1. الضرب الإتجاهى لمتجهين 3D

D3DXVec3Cross (D3DXVECTOR3* pOut, **const** D3DXVECTOR3* pV1,
const D3DXVECTOR3* pV2);

ناتج الدالة هو متجه عمودى على المتجهين pV1 و pV2 و يتم الحصول عليه من pOut.

D3DXVECTOR3 A, B, Result;
D3DXVec3Cross (&Result, &A, &B);

2. الضرب القياسى لمتجهين 3D

float D3DXVec3Dot (**const** D3DXVECTOR3* pV1, **const** D3DXVECTOR3* pV2);

ناتج الدالة هو جيب تمام الزاوية Cos بين المتجهين.

D3DXVECTOR3 A, B;
float angle = D3DXVec3Dot (&A, &B);

3. طول المتجهات

float D3DXVec3Length (**const** D3DXVECTOR3* pV);

تعود هذه الدالة بطول المتجه المدخل.

D3DXVECTOR3 A;
float Length = D3DXVec3Length (&A);

4. تسوية المتجهات

D3DXVec3Normalize (D3DXVECTOR3* pOut, **const** D3DXVECTOR3* pV);

تأخذ الدالة السابقة متجه عادى و تقوم بتحويله إلى متجه وحدة.

D3DXVECTOR3 A;
D3DXVec3Normalize (&A, &A);

الدالة السابقة تأخذ متجهين أولهما هو المتجه الناتج من عملية التسوية و الآخر هو المتجه المراد تسويته. فإذا كنت تريد تسوية المتجه فقد تفعل كما فعلنا فى المثال السابق، حيث حددنا أن المتجه المدخل و المتجه الناتج هما A، و هنا سيتم تسوية المتجه A و العودة بالمتجه الجديد داخله.

داخل مشروعا الأول، توجد ثلاثة مصفوفات سيتم إستخدامهم بشكل رئيسى. تلك المصفوفات مجتمعين يظهرها الشكل المبدئى للتحويلات من فضاء المجسم إلى فضاء العرض **Projection Space**. وكما عرفنا سابقاً، فإنه بعدما يتم تمرير سطح ما خلال تلك المصفوفات ستكون رؤوسه جاهزة للتحويل من فضاء العرض إلى نظام العرض على الشاشة. أما الآن سنقوم بدراسة تلك المصفوفات الثلاثة دراسة عميقة، لنفهم كيفية أستخدامهم و لنعرف بعض الحقائق الهامة عنهم.

مصفوفة اللعبة The World Matrix

كل كائن داخل اللعبة لديه مصفوفة خاصة به تسمى مصفوفة اللعبة **World Matrix** - توجد أسماء أخرى لهذه المصفوفة مثل مصفوفة المجال العام أو مصفوفة الكائن، هذه المصفوفة هى المسئولة عن تحريك و تحجيم و تدوير الكائن داخل اللعبة. أول خطوة سنفعلها داخل اللعبة هى ضرب رؤوس الأسطح فى هذه المصفوفة، و بهذا يتم تحويل السطح من فضاءه الخاص إلى فضاء اللعبة. و بإستخدامها سنقوم بتدوير و تحريك الكائن و بهذا نقوم بصنع حركة **Animation**. الكائن المسئول عن تمثيل المجسمات داخل اللعبة شكله كالتالى:

```
class CObject
{
Public:
    CMesh* pMesh;
    D3DXMATRIX WorldMatrix;
};
```

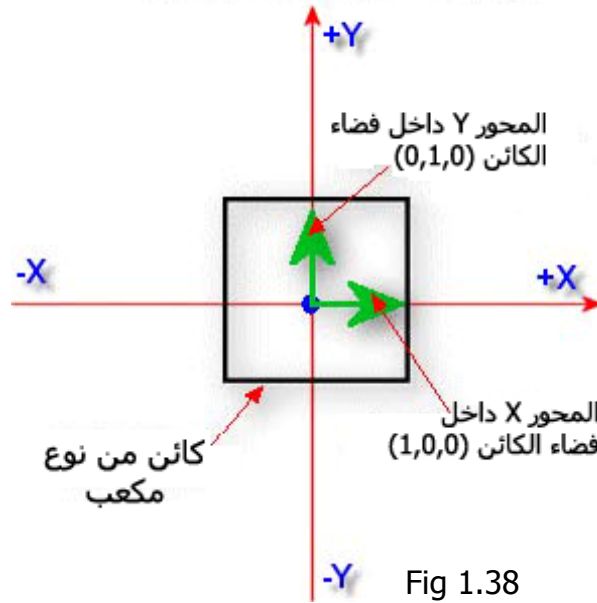
إذا قمنا بجعل مصفوفة كائن ما مصفوفة وحدة، نعرف بهذا انه لن يتم تدويره أو تحريكه و أنه يتمركز عند نقطة أصل اللعبة أى عند النقطة (0,0,0)، و يفترض أن الكائن حينها يواجه الإتجاه السفلى الموجب للمحور **Z**. فالننظر للمصفوفة القادمة:

$$\begin{matrix} \text{المحور X} \\ \text{المحور Y} \\ \text{المحور Z} \\ \text{موقع الكائن} \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

فالنتجاهل العمود و الصف الرابع للحظة، نستطيع أن نرى أن أول 3 متجهات يماثلوا تماماً المحاور **X** و **Y** و **Z** على التوالي داخل اللعبة. فعلى سبيل المثال، الصف الثالث هو متجه وحدة و قيمته (0,0,1) و يماثل المحور **Z** داخل اللعبة. لذا قد تقول أن الصفوف الثلاثة تمثل النظام الإحداثى الخاص بالكائن داخل فضاءه الخاص، و أنهم يمثلوا اتجاه دوران المحاور **X** و **Y** و **Z** الخاصة بالكائن و علاقتهم بالمحاور **X** و **Y** و **Z** الخاصة بنظام إحداثيات اللعبة. ملاحظة أخرى و هى انه نظام الإحداثيات الخاص بالكائن يطابق النظام الإحداثى العام للعبة فى حالة أستخدامنا لمصفوفة وحدة.

فى الصورة القادمة الصفان **X** و **Y** داخل مصفوفة الوحدة هما متجهى وحدة و يشيروا فى نفس إتجاه محاور النظام الإحداثى للعبة. نحن نعلم مسبقاً أن أى متجهات سيتم إدخالها لتلك المصفوفة لن تتغير لأن الإعمدة الموجودة بها هى أعمدة وحدة.

مصفوفة اللعبة (مصفوفة وحدة)



لذا فسواء كانت مصفوفة الكائن مصفوفة وحدة أو غير ذلك، فإننا سنتعامل مع أول 3 صفوف داخل مصفوفة الكائن كأنهم متجهات وحدة و أنهم يصفوا إحداثيات الكائن داخل فضاءه الخاص.

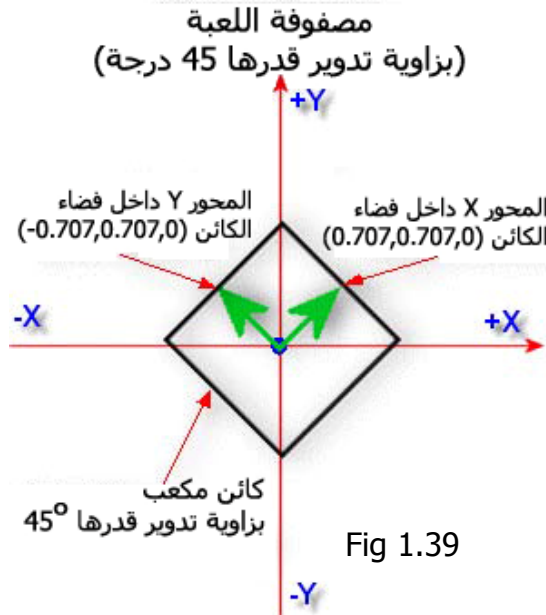
لنرى ما سيحدث عندما نقوم بدمج مصفوفة الكائن (و هي مصفوفة وحدة) بمصفوفة تدوير حول المحور **Z**. فى المثال القادم، سنقوم ببناء مصفوفة تدوير بمقدار 45 درجة أى 0.78539 عشرية.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos(45) & \sin(45) & 0 & 0 \\ -\sin(45) & \cos(45) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{array}{l} \text{المحور X داخل فضاء الكائن الخاص} \\ \text{المحور Y داخل فضاء الكائن الخاص} \\ \text{المحور Z داخل فضاء الكائن الخاص} \\ \text{موقع الكائن} \end{array} \begin{bmatrix} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

فى الصورة القادمة نرى أن محاور النظام الإحداثى الخاص بالمكعب قد دورت بمقدار 45 درجة.



إذا كان لدينا متجه وحدة يصف الإتجاه الذى يشير إليه كائن ما، فإننا نستطيع إستخدام ذلك المتجه لتحديث موقع ذلك الكائن داخل اللعبة. مصفوفة الكائن **World Matrix** تحتوى على كل المعلومات اللازمة لتحريك الكائن بمحاذاة المحاور الخاصة به أو حتى محاور اللعبة. فعلى سبيل المثال، الصف الثالث فى المصفوفة يسمى بمتجه الرؤية **Look Vector**. و هو متجه وحدة يصف الإتجاه الذى يشير إليه الكائن. و هذا الصف فعلياً هو المحور **Z** الخاص بإحداثيات الكائن الخاصة. و علاقته بالكائن داخل اللعبة تماثل تماماً علاقته بالكائن داخل فضاءه الخاص. فإذا أردنا أن نحرك الكائن مسافة معينة للأمام سنقوم بإستخدام الصف الثالث بالمصفوفة لفعل ذلك:

```
void MoveForward (float Dist)
```

```
{
    D3DXVECTOR3 LookVector;

    LookVector.x = SpaceShip.WorldMatrix._31;
    LookVector.y = SpaceShip.WorldMatrix._32;
    LookVector.z = SpaceShip.WorldMatrix._33;

    SpaceShip.WorldMatrix._41 += LookVector.x * Dist;
    SpaceShip.WorldMatrix._42 += LookVector.y * Dist;
    SpaceShip.WorldMatrix._43 += LookVector.z * Dist;
}
```

الدالة السابقة تقوم بإستخلاص قيمة متجه الرؤية من مصفوفة الكائن و تقوم بتعديلها بإستخدام المسافة المدخلة. و بعد ذلك تقوم الدالة بإضافة الناتج للموقع الحالى للدالة (الصف الرابع). و حيث أن متجه الرؤية هو متجه وحدة، فسيتم توزيع المسافة على المحاور **X** و **Y** و **Z** بنسب متساوية و بذلك تحتوى مصفوفة الكائن على الإحداثيات الجديدة. و الموقع الجديد هو عبارة عن المسافة الجديدة المدخلة **Dist** بدءاً من آخر موقع كان فيه و فى إتجاه متجه الرؤية **Look Vector**. لذا أياً كانت زاوية دوران الكائن داخل اللعبة، نستطيع تحريكه للأمام و فى اتجاهه الحالى. إذا قمنا بإدخال قيمة سالبة للدالة السابقة، فإن الكائن سيتحرك للخلف داخل اللعبة.

لدينا أيضاً المحور **Y** داخل إحداثيات الكائن الخاصة و الذى يسمى بمتجه الارتفاع **Up Vector** و موقعه الصف الثانى بالمصفوفة. أيضاً لدينا المحور **X** داخل إحداثيات الكائن الخاصة و يسمى بمتجه الجانب **Right Vector** و موقعه الصف الأول بالمصفوفة. و هذا يعنى على سبيل المثال أننا نستطيع تحريك الكائن لليمين و اليسار عن طريق صنع دالة تتعامل مع الصف الأول. فإذا قمنا بإدخال قيمة سالبة لتلك الدالة فسيتحرك الكائن ليسار بدلاً من اليمين.

```
void MoveStraf (float Dist)
```

```
{  
    D3DXVECTOR3 RightVector;  
  
    RightVector.x = SpaceShip.WorldMatrix._11;  
    RightVector.y = SpaceShip.WorldMatrix._12;  
    RightVector.z = SpaceShip.WorldMatrix._13;  
  
    SpaceShip.WorldMatrix._41 += RightVector.x * Dist;  
    SpaceShip.WorldMatrix._42 += RightVector.y * Dist;  
    SpaceShip.WorldMatrix._43 += RightVector.z * Dist;  
}
```

و بالمثالان السابقان أعتقد أنه لا توجد لديك مشكلة فى صنع دالة تسمى **MoveDown**.

سنقوم بتلخيص مصفوفة اللعبة فى الشكل التالى:

مصفوفة اللعبة

RightVector.x	RightVector.y	RightVector.z	0
UpVector.x	UpVector.y	UpVector.z	0
LookVector.x	LookVector.y	LookVector.z	0
Position.x	Position.y	Position.z	1

المصفوفة السابقة بها كل ما نحتاج لإتمام كل التحويلات التى نريدها. حيث سنقوم بضرب كل رؤوس الكائن بها ليتم تحويله من النظام الإحداثى الخاص به للنظام الإحداثى للعبة، و هذا سيتم بناءً على المعادلة التالية:

$WorldSpaceVertex = ModelSpaceVertex * WorldMatrix$

الخطوة التالية هي تحويل الرؤوس الموجودة بفضاء اللعبة لمجال الرؤية (تبعاً للكاميرا معينه). يمكن حفظ زاوية تدوير الكاميرا و موقعها داخل مصفوفة، هذه المصفوفة يطلق عليها أسم مصفوفة الرؤية **The View Matrix**.

التعامل مع مصفوفة الرؤية يختلف قليلاً عن التعامل مع مصفوفة اللعبة. و كما عرفنا سابقاً، لتحويل الرؤوس لمجال الرؤية لابد من عكس العملية المطلوبة عليهم. فعندما يتم تدوير الكاميرا لليمين، يتم تدوير الرؤوس لليسار. و إذا تم تحريك الكاميرا للأمام، يتم تحريك الرؤوس للخلف وهكذا. و لنقوم بتنفيذ تلك العمليات نقوم بإستخدام معكوس المصفوفة **The Inverse Matrix**.

فالفترض أنه لدينا 3 متجهات يمثلوا المتجه العلوى و متجه الرؤية و متجه الجانب للكاميرا و أيضاً متجه موقع الكاميرا داخل اللعبة و قمنا بإسناد قيم رقمية لهم:

```
D3DXVECTOR3 CLook; // Camera Look Vector
D3DXVECTOR3 CRight; // Camera Right Vector
D3DXVECTOR3 CUp; // Camera Up Vector
D3DXVECTOR3 CPos; // Camera World Space Position
```

إذا أردنا بناء مصفوفة للتعامل مع الكاميرا داخل فضاء اللعبة سيكون شكلها كالتالى:

$$\begin{bmatrix} \text{CRight.x} & \text{CRight.y} & \text{CRight.z} & 0 \\ \text{CUp.x} & \text{CUp.y} & \text{CUp.z} & 0 \\ \text{CLook.x} & \text{CLook.y} & \text{CLook.z} & 0 \\ \text{CPos.x} & \text{CPos.y} & \text{CPos.z} & 1 \end{bmatrix}$$

المصفوفة السابقة لا تعطينا النتيجة المرجوة منها. فى الواقع، هذه المصفوفة إذا أدخلنا لها رأس موجودة بمجال الرؤية فسيتم تحويلها لفضاء اللعبة. و هذا قد يسبب لنا مشاكل كثيرة سنقابل بعضها لاحقاً. و لكن ليس هذا ما نريده فى الوقت الحالى، كل ما نريده الآن هو إستخدام معكوس تلك المصفوفة:

$$\begin{bmatrix} \text{CRight.x} & \text{CUp.x} & \text{CLook.x} & 0 \\ \text{CRight.y} & \text{CUp.y} & \text{CLook.y} & 0 \\ \text{CRight.z} & \text{CUp.z} & \text{CLook.z} & 0 \\ -(\text{CPos} \bullet \text{CRight}) & -(\text{CPos} \bullet \text{CUp}) & -(\text{CPos} \bullet \text{CLook}) & 1 \end{bmatrix}$$

المصفوفة السابقة تقوم بتحويل الرؤوس من فضاء اللعبة لفضاء الرؤية. الكاميرا داخل اللعبة سيتم تمثيلها بمصفوفة 4x4. مكتبات الـ **D3DX** لديها دالة تقوم بعكس مصفوفة 4x4:

```
D3DXMatrixInverse (D3DXMATRIX* pOut, float* pDeterminant, const D3DXMATRIX* pM);
```

الدالة السابقة ندخل إليها المصفوفة المراد عكسها فى المعامل **pM** و نضع بالمعامل **pDeterminant** القيمة **NULL** حيث أننا لن نحتاجه. المعامل **pOut** سيحمل المصفوفة الناتجة من الدالة و لاحظ أن المصفوفة الناتجة لا يمكن أن تكون المصفوفة المدخلة.

توجد بعض الحالات التى سنحتاج فيها أستدعاء الدالة السابقة لعكس المصفوفات و لكنها بالتأكيد ليست الطريقة المثلى لإنشاء مصفوفة الرؤية كلما تحركت الكاميرا. لذلك فعملية حفظ موقع الكاميرا و زاوية دورانها داخل مصفوفة وحدة و تحديث قيم تلك المصفوفة ثم عكسها فى كل مرة تتحرك الكاميرا فيها هى عملية بطيئة و طويلة جداً. و لكن بدلاً من ذلك، نحن نعرف أن الكاميرا يتم التعبير عنها بأربع متجهات و هم **Look** ، **Up** ، **Right** ، **Position** لذلك سنقوم ببناء مصفوفة الرؤية بأنفسنا عن طريق إدخال قيم المتجهات للمصفوفة، و شكل المصفوفة خاصتنا سيكون شكلها مثل التى بالأعلى. وبالتالي سنقوم بإعادة بناء المصفوفة كلما تحركت الكاميرا أو تم تدويرها. المتجهات الأربعة السابقين لن نتعامل معهم و هم على شكلهم الحالى و بدلاً من ذلك سنقوم بتعديل **Class** الكاميرا التى صنعناها سابقاً لتلائم إحتياجاتنا الحالية.

```
class CCamera
{
public:
    D3DXVECTOR3 LookVector;
    D3DXVECTOR3 UpVector;
    D3DXVECTOR3 RightVector;
    D3DXVECTOR3 Position;
};
```

عندما تبدأ اللعبة فى العمل سنضع الكاميرا فى مكان مبدئى داخل اللعبة و سنجعل قيمة متجه الرؤية و متجه الارتفاع و متجه الجانب يساوى (0,0,1) و (0,1,0) و (1,0,0) على التوالى. بعد ذلك عندما يقوم اللاعب بالضغط على الأسهم، نقوم بتدوير المتجهات باستخدام مصفوفة تدوير فى اتجاه الزر المضغوط حتى نواجه الاتجاه الصحيح. و فى النهاية تقوم بإنشاء مصفوفة الرؤية من جديد عن طريق إدخال تلك المتجهات يدوياً للمصفوفة.

داخل مشروعنا الأول لن نقوم بتحريك الكاميرا و سنترك مصفوفة الرؤية مصفوفة وحدة. و هذا يعنى أننا سنرى اللعبة من مركزها أى من النقطة (0,0,0).

مصفوفة الرؤية

RightVector.x	UpVector.x	LookVector.x	0
RightVector.y	UpVector.y	LookVector.y	0
RightVector.z	UpVector.z	LookVector.z	0
-(Position • RightVector)	-(Position • UpVector)	-(Position • LookVector)	1

توجد العديد من دوال الـ **D3DX** التى تساعد فى بناء مصفوفة الرؤية. فى الفصل الرابع سنقوم بشرح مصفوفة الرؤية بشكل مفصل أكثر من ذلك و سنعرف تفاصيل أدق عنها، و ستعرف كيف تنشئ كاميرات معروفة مثل **Spaceship** و **First-Person Shooter**.

الدالة التى نحتاج لمعرفتها الآن هى **D3DXMatrixLookAtLH**، هذه الدالة تستخدم فى بناء مصفوفة الرؤية و تتطلب تفاصيل أكثر من السابقة. فهى تأخذ موقع الكاميرا الحالى داخل اللعبة و نقطة إلى حيث نريد الكاميرا أن تنظر و متجه الارتفاع للكاميرا (و هو فى الأغلب يكون <0,1,0> عند بداية اللعبة).

```
D3DXMatrixLookAtLH (D3DXMATRIX* pOut, const D3DXVECTOR3* pEye,
                    const D3DXVECTOR3* pAt, const D3DXVECTOR3* pUp);
```

معاملات الدالة السابقة كالتالى:

pOut: مؤشر لكائن **D3DXMATRIX** الذى سيستقبل مصفوفة الرؤية الناتجة.

pEye: موقع الكاميرا الحالى داخل فضاء اللعبة.

pAt: مكان داخل فضاء اللعبة حيثما نريد الكاميرا أن تنظر.

pUp: متجه ارتفاع الكاميرا.

لابد أن نحذر من أن الحرفين **LH** فى آخر الدالة السابقة يعنوا اليد اليسرى **Left Hand**. و حيث أن هذه الدالة تبني مصفوفة رؤية تعمل فى نظام اليد اليسرى، و حيث أن الـ **DirectX** و نحن سنقوم باستخدام ذلك النظام. فلا بد أن نعرف أن الـ **D3DX** لا تحتوى على دالة تسمى **D3DXMatrixLookAtRH** لذا فكن حذراً فى التعامل مع هذه الدالة.

معكوس المصفوفة يعطى تأثير عكس تأثير المصفوفة العادية. و بالتالى، عن طريق عكس مصفوفة كائن ما، نقوم بالحصول على مصفوفة تستطيع تحويل رؤوس الكائن من فضاء اللعبة رجوعاً إلى فضاء المجسم. هذا الأسلوب يستخدم كثيراً داخل دوال إكتشاف التصادمات **Collision Detection Methods** و التى تقوم بالتحقق من كل رأس ضد مكعب حدود اللعبة **World Space Bounding Box** أو كرة حدود اللعبة **World Space Bounding Sphere**. فى هذه الحالة الأفضل و الأسرع هو تحويل مصفوفة الكائن إلى مصفوفته الخاصة ثم نقوم بالإختبار بدلاً من تحويل كل رأس داخل المجسم إلى فضاء اللعبة ثم إجراء الإختبار.

إذا كنت تعرف أن مصفوفة التحويل تقوم بتحويل نقطة معينة من فضاء - سنسميه الخارجى - إلى فضاء آخر - سنسميه الداخلى، فإن معكوس مصفوفة التحويل يقوم بتحويل النقطة من الفضاء الداخلى إلى الفضاء الخارجى.

عرفنا من قبل أننا نستطيع ضرب المصفوفات لإنشاء مصفوفة تحويلات واحدة تقوم بتحويل أى متجه، و هذا كان سيحدث عند ضرب المتجه بالمصفوفات قبل ضربهم. سنقوم الآن بدمج مصفوفة الكائن بمصفوفة الرؤية تجهيزاً لرسم الكائن. و بهذه العملية نقوم بتحويل كل رؤوس الكائن من فضاءه إلى فضاء الرؤية، باستخدام عملية ضرب واحدة. هذه العملية توفر علينا وقت و جهد كثير حيث أنه كان لابد من ضرب كل رؤوس الكائن بمصفوفته حتى يتم تحويل الرؤوس لفضاء اللعبة، و بعد ذلك نقوم بضرب كل رؤوسه مره أخرى بمصفوفة الرؤية لتجهيزهم للرسم.

```
D3DXMATRIX ComboMat = Object.WorldMatrix * ViewMatrix;
D3DXVECTOR ViewSpaceVertex = ModelSpaceVertex * ComboMat;
```

داخل مشروعتنا الأول لن نفعل ذلك، حيث و عند تلك النقطة سنفضل أن نجعل كل مصفوفة على حدى حتى نستوعبهم جميعاً بشكل جيد.

أستنفذنا وقت جيد في الدرس السابق في شرح كيفية تحويل نقطة من نظام 3D إلى نظام 2D دون أن تفقد شكلها الأصلي. قد تتذكر حينها أن النتيجة دائماً كانت محصورة في المدى $[-1, +1]$ للمحورين x و y . و أيضاً تكلمنا عن المعادلات الخاصة بتلك العملية و عرفنا أن أسمها معادلات الرؤية على الشاشة **Projection Formulas** و كانت كالتالى:

$$2D \text{ ProjectionPoint } X = \frac{ViewSpace.X}{ViewSpace.Z}$$

$$2D \text{ ProjectionPoint } Y = \frac{ViewSpace.Y}{ViewSpace.Z}$$

أيضاً نتذكر أنه كلما أبتعدت النقطة عن الكاميرا يصغر حجمها و العكس بالعكس و هذا يعطينا الإحساس بالمنظور. أيضاً تذكر أن أهم خصائص تلك المعادلات كانت الرؤية بزاوية 90 درجة. و هذا يعنى أن الكاميرا تستطيع الرؤية بزاوية قدرها 45 درجة لليمين و 45 درجة لليسار و 45 درجة للأعلى و 45 درجة للأسفل. و قد تخيلنا هذا المشهد كأنه مخروط يزيد كلما أبتعدنا عن الكاميرا و بزاوية قدرها 90 درجة.

و هذا بالضبط ما تفعله مكتبات الـ **Direct3D** و أيضاً ما سنفعله نحن لتحويل مشهد 3D ليظهر على الشاشة. و لكن أن تصبح زاوية الرؤية 90 درجة و لا يمكن تغييرها فهذا أمر لايمكننا تقبله. لذا و كبداية، فزاوية الرؤية بمقدار 90 درجة ليست جيدة على الإطلاق. و أغلب المبرمجين يفضلوا استخدام زاوية رؤية ما بين 45 درجة و 65 درجة. و لكن هذا ليس صحيح، فالبشر يستطيعوا الرؤية بزاوية عرضاً أكبر من ذلك، و لكن داخل عالم الألعاب تكون هذه الزاوية مناسبة تماماً و سليمة. أما ثانياً، فشاشة الكمبيوتر ليست مربعة و فى أغلب الأحوال يكون عرضها أكبر من طولها. و هذا يعنى أنه لابد و أن تكون زاوية الرؤية بالعرض أكبر منها بالطول. و لكن إذا لم نفعل ذلك ستعمل اللعبة و بها خطين أسودين سميكين على الجوانب لإننا نقوم بتشغيل لعبة مربعة على شاشة مستطيلة. فإذا كانت اللعبة تعمل على شاشة مربعة فلا توجد مشكلة. و لكن فى الأغلب، نفضل كمبرمجين أن تعمل ألعابنا على الشاشة بأكملها **Fullscreen** و الأغلب تكون أبعاد الشاشات 800x600 أو 1024x768 وهكذا و كما تلاحظ فهى أبعاد مستطيلة.

و لكن إذا كانت زاوية الرؤية 90 درجة فعلاً فنحن فى مشكلة. فمكتبات الـ **Direct3D** تقوم بتجهيز نافذة العرض باستخدام معادلات الرؤية و بزاوية قدرها 90 درجة. و لكننا نفضل استخدام الرؤية الحرة **Arbitrary FOV**.

و لنحل تلك المشكلة سنقوم بضرب المتجهات الموجودة بفضاء الرؤية بمصفوفة جديدة قبل تطبيق معادلات الرؤية. هذه المصفوفة تسمى بمصفوفة الرؤية على الشاشة **Perspective Projection Matrix** تقوم بتشويه شكل المجسم - نعم أنت لم تخطئ قراءة الكلمة - داخل اللعبة بنسبة معينة و هذا لإصلاح الإيحاء بالرؤية الحرة. و لكن و حتى الآن لازالت الرؤية تتم بزاوية 90 درجة. و لكن و بسبب أن رؤوس المجسم تم إعادة تحجيمها داخل اللعبة، نستطيع أن نتحكم ما إذا كانت الرؤوس ستظهر داخل فضاء الرؤية أم لا.

سيتم ضرب رؤوس المجسم فى المصفوفة الجديدة بعد أن يتم تحويلها لفضاء الرؤية و هذا يعنى أننا نستطيع كتابة الشكل الكامل لمعادلة التحويل من فضاء المجسم إلى فضاء الرؤية على الشاشة كالتالى:

$$ProjectedVertex = ModelVertex + (WorldMatrix + ViewMatrix + ProjectionMatrix);$$

عند هذه النقطة، سنقوم برسم الرؤوس الموجودة بالمدى $[-1, +1]$ للمحورين x و y على الشاشة كما سنرى لاحقاً فى هذا الدرس.

إنه لمن الغريب أن تسمى تلك المصفوفة بمصفوفة الرؤية على الشاشة. حيث أنها لا تقوم بتحويل الرؤوس من فضاء 3D إلى فضاء 2D. على العكس فكل ما تفعله هو إبعاد أو تقريب مكان الرؤوس من الكاميرا تمهيداً لعملية القسمة (القسمة على z). لذا قد نقول أن مصفوفة الرؤية على الشاشة تجهز الرؤوس الـ 3D للتحويل إلى فضاء 2D.

قم بقراءة الجزء الخاص بالرؤية من منظور طبيعي إذا كنت تحتاج، لأنه من الهام أن تعرف لماذا القسمة على المحور z تنتج رؤية بزاوية قدرها 90 درجة، و هذا لتكون جاهز لفهم الجزء التالي.

لنبدأ في تحليل مصفوفة الرؤية على الشاشة و سنتعامل معها و هى مصفوفة وحدة. نحن نعرف أن تلك المصفوفة هى مصفوفة 4x4. لذا فالمتجه المدخل سيكون متجه محايد رباعى الأبعاد و سيكون على الشكل (x,y,z,1) كما عرفنا سابقاً. و كما عرفنا أيضاً من المصفوفات التى تعاملنا معها من قبل، فإن العمود W فى المصفوفة هو عمود وحدة. لذا فالمتجه الناتج سيكون أيضاً على الشكل (x,y,z,1) و بالتالى نستطيع تجاهل العمود W أثناء الضرب.

$$V[x \ y \ z \ 1] \times M \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = P[X \ Y \ Z \ W = 1]$$

المعادلة السابقة لا تفعل شيئاً. وحيث أن المصفوفة هى مصفوفة وحدة، لذا فالمتجه الناتج P سيكون مطابق للمتجه المدخل V. و بمجرد حساب قيمة المتجه P نستطيع تطبيق معادلات الرؤية عليه لمعرفة مكانه الجديد داخل الفضاء الـ 2D. و بهذا نقوم بتجسيم المتجه بزاوية رؤية مقدارها 90 درجة.

لاحظ أن مصفوفة الرؤية على الشاشة هى آخر مرحلة نستطيع عندها التحكم فى رؤوس المجسمات باستخدام دوال التحويل الخاصة بالـ Direct3D. حيث أننا سنقوم بتمرير مصفوفة اللعبة و مصفوفة الرؤية و مصفوفة الرؤية على الشاشة و سنقوم بعدها بإستدعاء الدالة DrawPrimitive لتقوم برسم الرؤوس على الشاشة. أما مكتبات الـ Direct3D ستقوم بضرب الرؤوس بتلك المصفوفات الثلاثة و بعدها ستقوم بتطبيق معادلات الرؤية على المتجه الناتج من مصفوفة الرؤية على الشاشة. و فى النهاية ستقوم مكتبات الـ Direct3D بإعادة رسم الرؤوس داخل نظام إحداثيات الشاشة. داخل مشروعا الاول محرك الإكساء الموجود به سيقوم بمحاكاة الأساليب التى تطبقها الـ Direct3D و لكن بشكل محدود. و لهذا سنقوم بتجهيز مصفوفة الرؤية على الشاشة بنفس الطريقة التى كنا سنتبعها عند التعامل مع الـ Direct3D.

المشكلة الأولى التى لابد و أن نعرفها هى أن الـ Direct3D يتطلب منا أن يكون المحور W بالمتجه الناتج يساوى فى القيمة المحور z فى المتجه المدخل بعد إتمام عملية الضرب فى المصفوفة بمعنى أن يكون W=z بعد عملية الضرب. و السبب فى هذا هو أن الـ Direct3D يقوم المحور W فى حسابات أخرى سنعرفها فى حينها. قد تقول أن الأسهل هو نسخ قيمة z من المتجه المدخل إلى Z فى المتجه الناتج. و لكن و كما سنعرف لاحقاً أننا نريد المحور Z فى المتجه الناتج ليحمل لنا قيمة معينة تمهيداً لإستخدامها فى شئ يسمى بمخزن العمق Depth Buffer. عملية نسخ المحور z من المتجه الناتج إلى المحور W فى المتجه الناتج ليست عملية صعبة أو شاقة و ستتم بتعديل بسيط سنقوم به داخل المصفوفة كالتالى:

$$V[x \ y \ z \ 1] \times M \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = P[X \ Y \ Z \ W = z]$$

بتعديل العمود الرابع بحيث يكون الصف الثالث يحتوى على الرقم 1 بدلاً من الصف الرابع، سيصبح المحور W فى المتجه الناتج يساوى المحور z فى المتجه المدخل و يمكن حساب قيمة المحور W كما يلى:

$$W = x*0 + y*0 + z*1 + w*0 = z$$

بهذا نجد أن المصفوفة نجحت فى نسخ القيمة بشكل صحيح. و كمثال، إذا كان $V=(20,40,60,1)$ فإنه بعد عملية الضرب سيكون $P=(20,40,60,60)$. عرفنا من قبل أنه لتحويل رأس من فضاء الرؤية إلى فضاء 2D و أثناء القسمة على z نستخدم المعادلات التالية:

$$2D\ ProjectionPoint\ X = \frac{ViewSpace.X}{ViewSpace.Z}$$

$$2D\ ProjectionPoint\ Y = \frac{ViewSpace.Y}{ViewSpace.Z}$$

عند هذه النقطة، يصبح لدينا نظام إحداثى جديد. هذا النظام يوجد عندما تكون الرأس قد ضربت فى مصفوفة الرؤية على الشاشة و لكن قبل تطبيق معادلات الرؤية. النظام الإحداثى الجديد يسمى بنظام الرؤية المختصر **Homogenous Clip Space**. و الفرق الوحيد بين فضاء الرؤية المختصر و فضاء الرؤية هو أنه فى فضاء الرؤية المختصر يكون المحور W يساوى المحور z كالتالى:

$$View\ Space\ Vector = (x, y, z, 1)$$

$$Homogenous\ Clip\ Space\ Vector = (x, y, z, z)$$

و حيث أن المحور w يحتوى على قيمة المحور z و المحور z سيحتوى فيما بعد على قيمة أخرى. لذا تقوم مكتبات الـ **Direct3D** و كذلك محرك اللعبة خاصتنا باستخدام معادلات الرؤية التالية:

$$2D\ ProjectionSpace.X = \frac{Homogenous\ ClipSpace.X}{Homogenous\ ClipSpace.W}$$

$$2D\ ProjectionSpace.Y = \frac{Homogenous\ ClipSpace.Y}{Homogenous\ ClipSpace.W}$$

معادلات الرؤية لم تتغير، و كل ما حدث هو أننا أستخدمنا w و نحن نعرف أن قيمة z موجودة بـ w . لذا فبتعديل مصفوفة الرؤية على الشاشة، سيكون المتجه الناتج منها يأخذ الشكل التالى:

$$(X, Y, Z=Depth\ Buffer, W=Z)$$

سنتجاهل مخزن العمق الآن، لإننا سنعرفه فى الدرس التالى. و لكن و حتى الآن كل ما يهمنا هو كيف سيقوم المحورين x و y بالمصفوفة بغطائنا الرؤية بشكل حر. لقد عرفنا حل مشكلة المحور w ، أما الآن فسندرس ماذا سنفعل مع العمودين الأول و الثانى بالمصفوفة.