**① What is Quaternion ?**

쿼터니온 (Quaternion)이란 사원수라고도 한다(알크 : [홈페이지](http://www.alc.co.jp/)）。

**숫자가 4개 모인것으로, 말하자면 4차원 벡터이다**.

3차원 벡터라고하면 가로,세로, 높이를 상상할수 있지만,

4차원이라 하면 도라에몽 밖에 생각 나지 않는다 =ㅁ=.

쿼터니온에 대해서 마이크로소프트 홈페이지에 설명이 있다.

<http://www.microsoft.com/japan/msdn/academic/Articles/DirectX/01/>

이해를 돕기 위해 한번 읽어보는것도 좋을 것이다

DirectX9의 매뉴얼 (Direct3D의 기초지식-> 3D좌표계와 지오메트리)부터

쿼터니온에 대해서 발췌해보자

|  |
| --- |
| 쿼터니온은 3성분 벡터를 정의하는 [x,y,z]값에 제4의 성분을 추가한 것이다. 쿼터니온은 3D회전에서 일반적으로 사용되는 행렬을 대신한다. **쿼터니온은 3D 공간내의 축과, 그 축을 중심으로 하는 회전을 나타낸다**. 예를들면 하나의쿼터니온 축(1,1,2)와 1라디안의 회전을 나타내는 것이 가능하다. 쿼터니온은 중요한 정보도 전해주지만, **그 진가는 쿼터니온상에서 실행가능한 2개의 처리, 합성과 보간에서 발휘된다**. |

　색칠되어있는 부분이 매우 중요하다. 여기에 의하면 쿼터니온을 사용하면 **원점으로부터 임의의 방향으로 늘린 회전축을 중심으로 하는 오브젝트의 회전을 표현할수 있다** 라는 것이다.

윗 문장에 빨갛게 칠해진 문장이 매력적이다.

어떤 벡터로부터 다른 벡터로 변환하는 것이 가능하다는 것이다.

예를 들면 어떤 방향으로 가고있는 비행기를 다른 방향으로 돌리고 싶을때 등에 사용할수 있다.

또 어떤 위치에 있는 어떤 물체의 방향을 보기위해 어떤 벡터를 변화시킬까 같은

고민의 문제를 해결하는데에도 쓰일수 있다.

**② 쿼터니온의 기초 중의 기초**

　DirectX의 매뉴얼에서 알 수 있는 것은 이 정도로 하고, 매력적이지만 사용법은 알려주지 않는다.

따라서 알기 쉬운 홈페이지를 하나 소개한다（<http://staff.aist.go.jp/toru-nakata/quaternion.html>）。

이 페이지에 의하면 쿼터니온Q는 1개의 실수부(실수)와 3개의 허수부(허수)로 구성되어 있으며

이 때의 3차원공간의 좌표는

**Q = (0; x, y, z)**

라고 나타낸다. 여기서 세미콜론을 기준으로 왼쪽이 실수부, x,y,z가 허수부이다.

쿼터니온인 점을 회전하려면, 쿼터니온끼리의 곱셈이 필요하다.

쿼터니온 Q1과 Q2를

**Q1 = (t1; x1, y1, z1)=(t1; V1)**

**Q2 = (t2; x2, y2, z2)=(t2; V2)**

라고 했을때 곱셈은

**Q1\*Q2 = (t1t2-V1・V2; t1V2+t2V1+V1×V2)**

가 된다. 여기서 V1, V2는 3D공간의 좌표로, t1이나 t2는 어떤 실수를 나타낸다.

이 곱셈이 실은 임의에 축에 의한 회전에 필요한 것이지만,

주목할것은 **쿼터니온끼리의 곱셈을 벡터의 내적과 외적만으로 계산이 가능하다는 점이다**

**③ 쿼터니온에 의한 임의 축 회전**

이정도만 알아도 임의의 축회전을 할수 있다!! 우선 회전하고 싶은 점을

**P = (0; xp, yp, zp)=(0; Vp)**

이라고 한다. 다음으로 회전축의 방향을 나타낼 벡터를

**v = (xv, yv, zv)**

라고 하고, 회전하고싶은 각도를 θ라고 하자。단|**v**|=1이며, **v**에서 다음 2개의 쿼터니온을 작성한다

**Q = (cos(θ/2); xv・sin(θ/2), yv・sin(θ/2), zv・sin(θ/2))**

**R = (cos(θ/2); -xv・sin(θ/2), -yv・sin(θ/2), -zv・sin(θ/2))**

그다음 곱셈 한다

**R\*P\*Q = (0; x, y, z)**

이것만으로 점 p를 벡터 v를 향한 축 주변으로 θ 회전시킨 점의 좌표를 얻을 수 있다!.

식을 보면 cos(θ/2) 와 sin(θ/2)밖에 없으며 , 그 다음은 3차원의 내적과 외적을 2번씩 하면 된다.

이것은 매우 간단하며 고속이다

**④ Direct3DX 헬퍼 함수로 쿼터니온에서 월드 변환행렬을 생성**

이 단계에서, 쿼터니온에 의한 회전을 실제로 사용하는 것을 생각했을때?? 라고 생각했다

Direct3D에서 점의 좌표변환을 수행하는 것은 월드 변환행렬이다.

D3D에 의한 렌더링은 월드 변환-> 뷰 변환 -> 프로젝션 변환 -> 이라는 걸 거쳐야 한다

그럼 위에서의 회전축에 의한 버텍스변환은 어떻게 결합하면 될까?

[ 버텍스를 회전시킨 후, 회전을 하지 않는 월드 변환 행렬에 남은

스케일링과 평행이동을 하면 되지 않음? ] 이라고 생각할수도 있다.

하지만 틀렸다. 왜냐하면 로컬 좌표에 있는 버텍스를 직접변경하는것이 되기 때문이다.

로컬 좌표의 버텍스는 애니메이션을 하지 않는다고 했을때 부동(不動) 이어야 하며,

이것이 변경되면 원래대로 돌아가는것이 매우 어려워지게 된다

실제로는 Direct3DX에는 쿼터니온에서 월드변환행렬을 생성하는

눈물이 나올 정도로 기쁜 헬퍼 함수들이 몇가지 존재한다.

첫번째로 **D3DXMatrixTransformation 함수**이다。이 함수의 내부를 살펴보자

|  |
| --- |
| **D3DXMatrixTransformation 함수** |
| *D3DXMATRIX \* D3DXMatrixTransformation( D3DXMATRIX \* pOut, CONST D3DXVECTOR3 \* pScalingCenter, CONST D3DXQUATERNION \* pScalingRotation, CONST D3DXVECTOR3 \* pScaling, CONST D3DXVECTOR3 \* pRotationCenter, CONST D3DXQUATERNION \* pRotation, CONST D3DXVECTOR3 \* pTranslation ) ;* |

인수가 조낸 많아서 눈돌아 가지만 하나하나 차근 차근 살펴보자

***pOut***에 얻어낸 월드 변환 행렬을 반환한다

***pScallingCenter***라는것은 스케일링의 중심점이다. 스케일링은 원점 이외에도 가능하다.

단 월드 변환행렬에서는 일반적으로 원점 중심의 스케일링이기 때문에 여기선 NULL을 사용한다

***pScalingRotation***은 스케일링의 회전을 지정한다. 스케일링의 회전이라는것도 이상한 표현이다.

일반적으로 월드 변환행렬은 XYZ축에 대해서 스케일 값이 설정되어 있다.

이 XYZ축 자체가 회전하고 있는 상태를 상상해보면 알기 쉬울 것이다.

이상한 것을 할때 말고는여기도 NULL이면 된다

***pScalling***는 축 의 스케일 값을 설정한다

**pRotationCenter**는 회전의 중심을 지정한다.회전의 중심이란 회전축의 원점의 평행이동을 의미하며, ③에서 설명한 회전은 어디까지나 원점을 지나는 직선이지만, 이 원점을 다른 점으로 하고 싶을때의 이 인수를 설정한다. NULL로 하면 원점이 사용된다

***pRotation***에는 회전축과 회전각도를 지정한다.

이부분이 중요한 곳이지만, 지정밥법이 독특하므로 추후 다시 언급한다

***pTranslation***는 스케일링과 회전이 끝난 버텍스를 오프셋(평행이동)할때 사용한다.

월드 좌표에서의 배치는 이부분에서 수행한다.

즉, 이 함수는 스케일링의 좌표, 회전 좌표, 오프셋의 좌표를 완전 별도로 취급하여 설정 가능하다는 것이다. 단, 일반적인 사용에 관해서 말한다면, 스케일링은 원점을 중심으로 축평행, 회전축의 중심도 원점일 경우가 대부분이므로 설정하는 곳은 ***pScalling*** 과 ***pRotation*** 과 ***pTranslation*** 만 쓰면 된다

쿼터니온에 의한 회전축과 회전각도의 지정은 조금 귀찮지만, 여기서는 헬퍼 함수에게 맡긴다

D3DXQuaternionRotationAxis함수를 사용하며 이 함수는 지정한 쿼터니온을 회전한 결과를 반환한다

|  |
| --- |
| **D3DXQuaternionRotationAxis 함수** |
| *D3DXQUATERNION \* D3DXQuaternionRotationAxis( D3DXQUATERNION \* pOut , CONST D3DXVECTOR3 \* pV , FLOAT Angle ) ;* |

***pOut***에는 회전시키고싶은 쿼터니온을 넣는다. 그러면 계산결과가 이곳으로 반환한다。

In 과 Out이 같이 되어있다

***pV***에는 회전축의 방향벡터를 넣는다  
***Angle***에는 회전축에 대한 회전각도를 넣는다

***pOut***에 단위 쿼터니온(x,y,z,w)=(0,0,0,1)을 넣고 축과 각도를 설정하면

원하는 쿼터니온을 얻을 수 있다

이걸로 쿼터니온에 의한 회전을 사용하여 월드 변환행렬을 한번에 생성 할 수 있다

　D3DXMatrixTransformation함수를 사용해도 좋지만, 명백히 계산량이 많아 보인다.

가장 심플하게 생성하고 싶을 경우에는 D3DXMatrixRotationQuaternion함수를 사용한다

|  |
| --- |
| **D3DXMatrixRotationQuaternion関数** |
| *D3DXMATRIX \* D3DXMatrixRotationQuaternion( D3DXMATRIX \* pOut , CONST D3DXQUATERNION \* pQ ) ;* |

매우 심플하다  
***pOut***에는 쿼터니온에 의해 정의된 회전행렬을 반환한다。  
***pQ***가 회전을 정의하는 쿼터니온이다

　우선***pQ***를 주면 회전행렬이 계산된다. 그 후 월드변환행렬을 생성할때의 이 행렬을 곱해주면 된다. 이쪽이 훨씬 간단하고 쓸데없는 계산을 피할수 있어 그만큼 빨라진다

**⑤ 축회전지정인가 임의축회전지정인가**

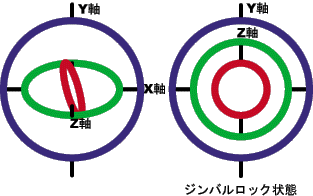
　④까지에서 일반적인 회전축과 쿼터니온에 의한 임의축회전이라는 2가지의 회전방법에서 월드변환행렬을 생성할 수 있다고 배웠다. 어느쪽이 쓰기 쉬운가? 이것은 상황에 따라 틀리다고 생각되지만 쿼터니온에 의한 임의축 회전이 좋다고 생각한다.

나는 땅에서 걷고 있고 새는 하늘에서 날고 있다. 이때 시선을 새쪽으로 바라보고 싶을때 어떤 축을 어떻게 회전 시키면 될까? 금방 생각나지는 않는다. 하지만 쿼터니온에 의한 임의 회전축을 사용한다면 새의 방향을 가리키는 벡터만 계산하면 하늘쪽을 바라보는 월드변환행렬을 바로 생성 할 수 있다.

또 하나 중요한것은 임의 회전축에는 [ 짐벌락 현상이 없다 ] 라는 이점이 있다.

짐벌락 현상이란 회전축이 겹쳐져서 특정 방향밖에 오브젝트를 돌릴수 없는 현상이다.

[ 회전축이 겹친다 ] 라는 의미는 [GAME CODING Vol.1 Direct3D/COM ] 에 의하면 짐벌락은 자이로 스코프의 짐벌이라는 부품이 락이 걸리는 현상으로 나왔던 것이다. 자이로 스코프란 아래와 비슷한 것이다



3개의 축이 축에서 회전가능한 형태로 되어있어 중앙의 빨간 축을 여러 방향으로 돌려보면 녹색이나 파란색 축이 같이 돌아주어 임의의 자세를 표현가능한 장치로 되어있다.

오른쪽 그림같이 X 축이 90도 회전해서 파란색 축과 녹색축이 동일평면상으로 오게 되면 Z축과 Y축이 겹쳐져 버리기 때문에 빨간축을 문의 손잡이 돌리는것 처럼 되지 않는다 .

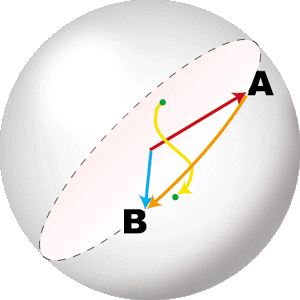
이것이 짐벌락이다. 축의 회전을 보존 하려고 하면 짐벌락이 되는 가능성이 생긴다

각도를 보존해도 임의의 다른 자세를 기묘하게 변화시킬 수 있는

쿼터니온은 회전축에는 없는 어드밴티지가 있다.

**⑥ 구면선형보간**

쿼터니온을 조사해보면 여기저기에 [ 구면선형보간 ] 이라는 용어가 나온다. 어떤 페이지에 의하면 이것은 「**떤 회전축 v1에서 θ1으로 돌고 있는 상태를 회전축 v2에서 θ2으로 돌고 있는 상태를 매끄럽게 변환하는 방법**」이라고 한다。이것만으로는 이미지를 상상하기 어렵기 때문에 아래에 그림을 보자



회전축 A 부터 회전축 B로 이동시키려고 할 때 가장 가까운것은 오렌지색의 화살표로 가리키고 있는 경로를 도달하는 것이다.

게다가 회전축 A에 의해 회전하고 있는 어떤 한 점(녹색 점)을 회전축 B의 점에 이동시키려면 A자체를 회전시켜서 B의 회전각도에 매끄럽게 맞출 필요가 있다.

이것을 회전축에서 수행하려고 해도 매우 어렵지만, 구면선형보간은 양쪽에 대해서 A->B간의 쿼터니온을 산출해 준다. 놀랄정도로 쓰기 좋은 보간 방법이다

[ 계산이 어렵진 않은가? ] 라고 생각 할 수도 있지만, 생각만큼 어렵지는 않다. 게다가 고맙게도 Direct3DX에는 이 계산을 구현해 놓은 헬퍼 함수인 D3DXQuaternionSlerp 함수가 있다.

|  |
| --- |
| **D3DXQuaternionSlerp 함수** |
| *D3DXQUATERNION \* D3DXQuaternionSlerp( D3DXQUATERNION \* pOut , CONST D3DXQUATERNION \* pQ1 , CONST D3DXQUATERNION \* pQ2 , FLOAT t ) ;* |

***pOut***에 보간된 쿼터니온이 반환된다  
***pQ1***이 시작점의 쿼터니온、***pQ2***가 마지막점의 쿼터니온이다  
***t***는 보간위치(간격)를 나타내는 수치로 0~1 값이다.

0일 수록***pQ1***에 가까우며 1에 가까울 수록***pQ2***에 가깝게 간다

이정도만 가지고도 훌륭한 보간법을 사용할 수 있다.

***pOut***으로 반환된 보간 쿼터니온은 그대로 D3DXMatrixTransformation 함수에 인자로 넣어 월드 변환좌표를 생성할수 있으므로 더욱 쓰기 좋다.

구면 선형보간의 사용 예로써, 예를 들면 액션 게임이나 3D의 STG에서 내쪽으로 적의 총구를 겨누게 할 경우가 있다. 이것은 좌표계에서도 가능할 수 있을지도 모르겠지만, 구면 선형보간을 사용하는 쪽이 훨씬 간단하다.

적이 지금 향하고 있는 방향벡터를 A, 적과 자신을 연결하는 방향 벡터를 B (물론 정규화 된것들)로써

쿼터니온을 생성한 뒤 D3DXQuaternionslerp 함수에 넣어 t를 0부터 1로 변화시켜주면 된다.

이때의 변화량이 문제가 되지만, 이것은 2개의 방향벡터의 각도 W를 내적으로 계산하여 적당히 정하는

변화량의 단위 (각속도) U에서 t = U / W로 계산한다.

t >1이라면 t = 1로 하면 된다.

**⑦ 쿼터니온에 의한 빌보드의 월드변환행렬 생성**

빌보드는 잘만 사용하면 적은 양의 폴리곤으로 화면에 여러가지 효과를 주는 것이 가능하다.

예를들어 파티클에 의한 불꽃이 있다.

점 폴리곤에서 불꽃을 생성하는 것도 가능하지만, 눈부신 효과등이 잘 표현되지 않는다.

그렇다고 해서 라이트를 사용할수도 없다 ( 애초에 라이트 자체는 발광하지 않는다 ).

이럴때에 작은 사각 폴리곤에 빛나는 [ 그림 ] 을 그려서 덧셈합성을 하면 예쁘게 표현이 가능하다

하지만 여기서 문제가 되는것은 카메라의 시점이 변할때이다. 정면에서 보면 예쁜 불꽃도 측면에서보면 사각형이 빤히 보인다. 여기서 카메라의 시선에 항상 빌보드가 바라볼수 있게 하면 동그란 불꽃같이 사람을 착각 시킨다.

카메라의 시선 벡터는 일반적으로 알 수 있으며

빌보드의 법선은 카메라의 시선과 정반대가 되면 된다.

빌보드의 법선을 vn = (nx, ny, nz), 카메라를 바라보고있는 방향을 vc = (cx, cy, cz)라고 하면

**vn = (nx, ny, nz) = (-cx, -cy, -cz)**

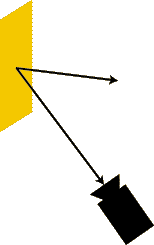
라고 카메라의 벡터에 마이너스를 붙이는것만으로 법선의 계산이 끝난다

[ 다음은 이 법선을 향해서 판자를 회전시키면 된다 ]

예를들어

vn = (-5, 8, 13)

일때의



오렌지 색의 사각 폴리곤을 카메라의 방향으로 바라보게 하려면 2개의 벡터가 포함된 평면에 수직인 벡터, 즉 법선 벡터를 축으로 하는 벡터가 이루는 각도만큼 회전시키면 된다.

2개의 벡터에 수직인 법선은 외적으로 이루는 각도는 내적으로 각각 계산할 수 있다.

다음은 쿼터니온을 생성해서 회전시키면 된다

|  |
| --- |
| D3DXVECTOR3 LookAt(-cx, -cy, -cz); // 카메라의 시선의 반대 벡터（정규화된것）  D3DXMATRIX TurnMat; // 회전행렬 D3DXQUATERNION q; // 회전 쿼터니온 D3DXVECTOR3 NAxis( 0, 1, 0); // 로컬 좌표에서 빌보드의 법선 D3DXVECTOR3 Normal;  // 회전축을 계산 D3DXVec3Cross(&Normal, &NAxis, &LookAt); // 법선 계산  // 회전각도 계산 float angle = acos(D3DXVec3Dot(&LookAt, &NAxis));  D3DXVec3Normalize(&Normal, &Normal); // 법선 정규화 q.x = q.y = q.z = 0.0f; q.w = 1.0f; **// 단위 쿼터니온으로 초기화** D3DXQuaternionRotationAxis( &q, &NAxis, angle); // 쿼터니온 생성 D3DXMatrixRotationQuaternion( &TurnMat, &q); |

단위 쿼터니온을 만드는것이 포인트이다.

이후에는 일반적으로 월드변환행렬을 생성하는것과 똑같이 행렬의 곱셈을 하면

항상 카메라의 방향으로 바라보는 빌보드가 완성된다.

단, 위의 구현대로라면 빌보드는 정면을 보고 있겠지만

카메라의 각도에 의해 빌보드가 시선축을 중심으로 회전하게 되버린다.

이것은 3차원 벡터만을 생각했기 때문에 [ 카메라의 업벡터 ] 와 [ 빌보드의 업벡터 ] 가 맞지 않기 때문이다. 따라서 위의 구현은 정확히 보이는 빌보드가 회전해도 문제 없을 경우에 사용 할 수 있다

쿼터니온은 나름 대로의 문제도 가지고 있지만 기본적인 사용법을 쓴다면 커다란 문제는 생기지 않을 것이다. 무엇보다 이점이 많으니 많이 사용하고 익숙해지면 좋을 것이다