Quaternionen voor proper rigid body transformations in computer graphics

Bram Arends, s4055586
Begeleider: Theo Schouten

Inhoudsopgave

HOOFDSTUK 1

Inleiding

Deze scriptie gaat over *proper rigid body transformations* in computer graphics. Deze transformaties bestaan uit een rotatie, gevolgd door een translatie. In deze scriptie worden meerdere methoden besproken om deze transformaties uit te voeren. Daarna wordt gekeken naar de efficiëntie en eventuele bijkomende problemen van die algoritmen. Verder zal ook kort gekeken worden naar normale *rigid body transformations*, die bestaan niet alleen uit rotaties en translaties, maar ook reflecties. *Non-rigid body transformations* zullen ook besproken worden, dit zijn transformaties die de vorm van het object veranderen, zoals scaling.

Bij de analyse van efficiëntie zal niet gekeken worden naar geheugengebruik, de algoritmen die besproken worden gebruiken immers weinig geheugen, en voor de huidige computers zal dit niet veel moeten uitmaken. De O-notatie zal ook niet gebruikt worden om de efficiëntie aan te geven:

$$O(g(n)) = \{f(n) : \exists_{c\in\mathbb{R}>0, n_0\in\mathbb{R}>0} \forall_{n\geq n_0} 0 \leq f(n) \leq cg(n)\}$$

Dit is omdat in dit geval n constant en klein blijft. In computer graphics wordt meestal gewerkt in drie dimensies, dus $n = 3$. In sommige gevallen is dat kleiner dan n_0, dus dan zegt de O-notatie niks over de efficiëntie en het aantal operaties dat uitgevoerd moeten worden. Om de efficiëntie te bepalen zal ik daarom van een aantal operaties bijhouden hoe vaak die gebruikt moeten worden bij een algoritme, zoals vermenigvuldiging en additie van twee getallen.

Het meest gebruikte algoritme om *proper rigid body transformations* te bewerkstelligen is met gebruik van rotatiematrices, deze zullen in het volgende hoofdstuk besproken worden. Nieuwere algoritmen om te roteren en transleren zijn in opkomst. In de hoofdstuk 3 en 4 zullen quaternionen en dual-quaternionen besproken worden. Dit zijn getallen met de gewenste eigenschappen om goed te roteren. Quaternionen worden in de praktijk al gebruikt, maar dual-quaternionen nog niet.

Ook zal gekeken worden naar manieren om *proper rigid body transformations* te interpoleren (keyframing), zodat het animeren van objecten ook efficiënt kan verlopen. Voor rotatiematrices wordt er één algoritme besproken. Voor quaternionen zijn er meerdere goede interpolatie-algoritmen, zoals SLERP, NLERP en LERP. Ze komen allemaal met hun voor- en nadelen en hebben allemaal andere eigenschappen. SLERP is de populairste van deze drie, en wordt daarom behandeld in deze scriptie.

Verder zal ik onderzoek doen naar het gebruik van hyper-dual-quaternionen bij proper rigid body transformaties. Deze zouden misschien efficiënt gebruikt kunnen worden om objecten op meerdere manieren te transformeren, zoals shearing en scaling. Deze getallen worden nog niet in de praktijk gebruikt, er is ook nog geen onderzoek naar gedaan (in de context van computer graphics). Dit gedeelte van het onderzoek is dus vooral theoretisch relevant.

Verder is er nog een implementatie van quaternionen, dual-quaternionen, hyper-dual-quaternionen en SLERP. Hiermee heb ik wat probleempjes ontdekt, waar ik in eerste instantie niet aan gedacht had en ook niet duidelijk worden vermeldt in de literatuur. Ook had ik wat observaties gedaan van de snelheden van de verschillende algoritmen. Uiteindelijk zal de conclusie volgens in het laatste hoofdstuk.

HOOFDSTUK 2

Rotatiematrices

De traditionele manier om objecten te roteren is met behulp van rotatiematrices. Dit is ook de meest intuïtieve manier om rotaties te realiseren. In de volgende drie secties wordt de theorie besproken, vervolgens zijn er twee secties over het fenomeen gimbal lock en over keyframing, en de laatste sectie geeft een analyse van het algoritme.

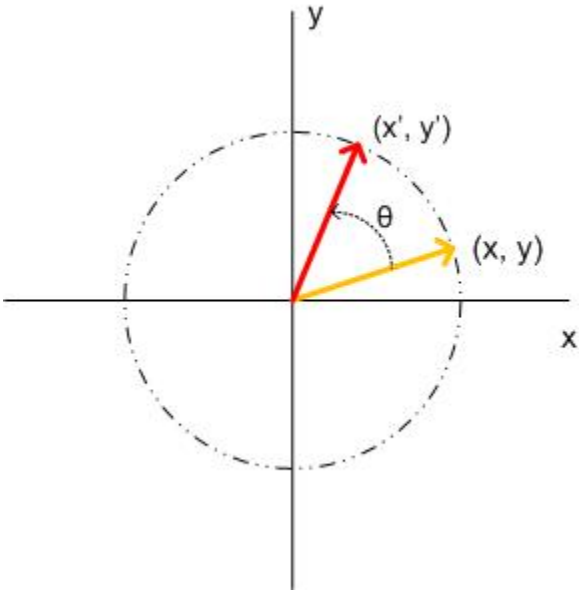
1. Rotatie in $\mathbb{R}^2$

Voor het roteren in twee dimensies (dus alleen op de x en de y-as worden de volgende twee formules gebruikt:

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

Waarbij (x, y) de originele coördinaten zijn van het object dat je wilt roteren en (x', y') de nieuwe coördinaten van het object na een rotatie van θ graden om de z-as. Als $\theta > 0$ dan is de rotatie tegen de klok in en als $\theta < 0$ dan is de rotatie met de klok mee.

FIGUUR 2.1.



Deze formules zijn te schrijven als één matrixmultiplicatie met rotatiematrix R [**Wat00**]:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = R(\theta) \begin{pmatrix} x \\ y \end{pmatrix}$$

Waarbij $R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$.

Het is niet al te moeilijk om R af te leiden. Stel, hebben een vector $(1, 0)$ en we roteren deze θ graden om de z-as, dan komen we uit op $(\cos(\theta), \sin(\theta))$. Als we nu een vector $(0, 1)$ θ graden om de z-as roteren, dan kom je uit op $(-\sin(\theta), \cos(\theta))$, want $(0, 1)$ heeft een hoek van 90 graden ten opzichte van $(1, 0)$, θ blijft gelijk, dus $(0, 1)$ wordt afgebeeld op $(\cos(\theta + \frac{\pi}{2}), \sin(\theta + \frac{\pi}{2})) = (-\sin(\theta), \cos(\theta))$. Dus nu hebben we een matrix nodig waarvoor geldt:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\sin(\theta) \\ \cos(\theta) \end{pmatrix} \tag{2}$$

Uit (1) valt meteen af te lezen dat $a = \cos(\theta)$ en $c = \sin(\theta)$. Uit (2) ook, dus $b = -\sin(\theta)$ en $d = \cos(\theta)$, nu krijg je de rotatiematrix R.

2. Rotatie in $\mathbb{R}^3$

Rotatie in drie dimensies komt op hetzelfde neer als in twee dimensies, alleen nu heb je twee extra *degrees of freedom*. Nu is het roteren om de x-as en y-as ook mogelijk, daarom zijn er nu drie rotatiematrices in plaats van één.

De rotatiematrix die een vector draait om de z-as heeft nu een dimensie erbij gekregen [**Wat00**]:

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Zoals te zien is, veranderd de z-coördinaat niet, en dat is natuurlijk ook de bedoeling als je alleen om de z-as roteert. De rotatiematrix die roteert om de x-as is analoog aan R_z. Als we de x-as met de y-as vervangen, de y-as met de z-as en de z-as met de x-as, dan roteert hij om de x-as:

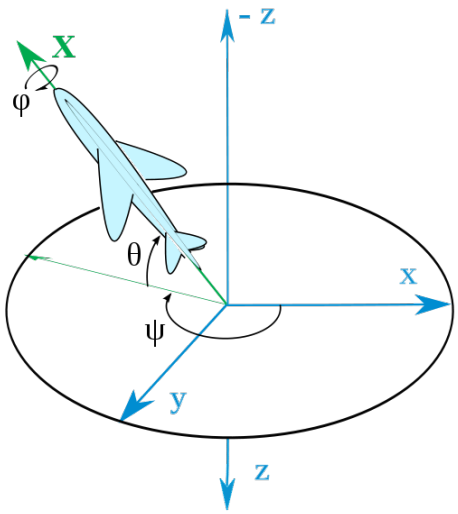
$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Als we de x-as vervangen met de z-as, de y-as met de x-as en de z-as met de y-as krijgen we de rotatiematrix die om de y-as draait:

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

Met deze drie rotatiematrices kan je alle kanten op roteren als je ze vermenigvuldigd met elkaar en met een vector. De volgorde waarin dat gebeurt maakt uit, want matrixvermenigvuldigen is niet commutatief. Er zijn verschillende conventies die gebruikt worden in de praktijk, zoals de zxz-conventie, maar ik zal de conventie gebruiken die gebruikelijk is in computer graphics en dat is de xyz-conventie, eerst roteren om de x-as, dan de y-as en dan de z-as: $(x', y', z') = R_x(\phi)R_y(\theta)R_z(\psi)(x, y, z)$. De hoeken ϕ, θ en ψ worden de Euler hoeken genoemd [**Gra06**].

FIGUUR 2.2. De hoeken bij het roteren van een vliegtuig. Bron: `en.wikipedia.org/wiki/File:Plane.svg`



3. Translatie

Met Euler hoeken kan je tegelijkertijd translaties uit voeren. Wel is daar een vier bij vier matrix voor nodig [**Wat00**]:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Het is gemakkelijk in te zien dat dit hetzelfde is als:

$$x' = x + T_x$$
$$y' = y + T_y$$
$$z' = z + T_z$$

Deze translatiematrix is gemakkelijk te combineren met één van de bovengenoemde rotatiematrixen, bijvoorbeeld:

$$R_x(\theta, T_x, T_y, T_z) = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & \cos(\theta) & -\sin(\theta) & T_y \\ 0 & \sin(\theta) & \cos(\theta) & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Keyframing

In de praktijk is het algoritme om te roteren met Euler hoeken niet genoeg. In de meeste gevallen willen we wel dat de rotatie geanimeerd wordt. Het bovenstaand algoritme heeft in de praktijk te veel rekentijd nodig om vloeiend te kunnen animeren, daarom worden het begin- en eindpunt van een rotatie berekend met Euler hoeken en worden de punten die er tussen liggen geïnterpoleerd.

Er zijn verschillende methodes om te interpoleren, maar ik ga hier een methode gebruiken uit 2002 uit [**Ale02**]. Het algoritme heeft geen naam, dus ik zal het verder gewoon Keyframing noemen. In het artikel worden twee nieuwe operatoren geïntroduceert: $\odot$ en $\oplus$.

Waarvan de definities als volgt zijn:

$$r \odot A = e^{r \ln(A)}$$

$$A \oplus B = e^{\ln(A)+\ln(B)}$$

A en B zijn hier matrices en r is hier een scalair.

Om te interpoleren wordt de volgende formule gebruikt:

$$C(t) = ((1-t) \odot A) \oplus (t \odot B),\ t \in [0,1]$$

A is hier de beginpunt van de beweging en B is de eindpunt daarvan. $C(0.5)$ geeft dan de vector die precies op de cirkelboog tussen A en B ligt.

Het logaritme en de het machtsverheffen zijn als volgt voor vectoren en matrices gedefinieerd:

$$e^{(x,y,z)} = (e^x, e^y, e^z)$$

$$\ln((x,y,z)) = (\ln(x), \ln(y), \ln(z))$$

Voor vectoren met een hogere dimensie en matrices komt het op hetzelfde neer, elementsgewijs worden dan de operaties uitgevoerd. Hierbij moet worden opgemerkt dat een elk element van de vector of matrix groter dan 0 moet zijn, vanwege het logaritme. Als dit niet zo is moet er een translatie worden uitgevoerd op A en B, zodat alle elementen daarvan groter dan 0 worden. Vervolgens moet dan de interpolatie berekend worden en dan kan de uitkomst met dezelfde translatie (maar dan negatief) weer terug gezet worden.

Natuurlijk kan hiervoor ook SLERP gebruikt worden (zie het volgend hoofdstuk), door de vectoren om te zetten naar quaternionen. Dit wordt meestal in de praktijk ook gedaan (de redenen daarvoor worden duidelijker in het volgend hoofdstuk), maar ik wil voor nu Euler hoeken en quaternionen strikt gescheiden houden. Ik kom hier nog wel op terug in de conclusie.

5. Gimbal Lock

Euler hoeken lijken een goede oplossing om te roteren. Theoretisch klopt dit, maar in de praktijk kan er een ongewenst verschijnsel optreden, en dat is gimbal lock. Gimbal lock is een bekend probleem en niet alleen in computer graphics. Volgens [**Vas09**] was het fenomeen al bij NASA bekend, omdat het een probleem was voor de navigatiesystemen van space shuttles. Door gimbal lock verliest er een *degree of freedom*, wat natuurlijk niet gewenst is in computer graphics en andere applicaties waar Euler hoeken gebruikt worden. Door gimbal lock kunnen er ongewenste bewegingen worden gemaakt in animatie. Hieronder staat wiskundig beschreven wat er gebeurd als gimbal lock optreedt [**Gra06**].

We hebben dus het volgende:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Stel $\theta = \frac{\pi}{2}$:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 1 \\ \sin(\phi) & \cos(\phi) & 0 \\ -\cos(\phi) & \sin(\phi) & 0 \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 1 \\ \sin(\phi)\cos(\psi) + \cos(\phi)\sin(\psi) & -\sin(\phi)\sin(\psi) + \cos(\phi)\cos(\psi) & 0 \\ -\cos(\phi)\cos(\psi) + \sin(\phi)\sin(\psi) & \cos(\phi)\sin(\psi) + \sin(\phi)\cos(\psi) & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 1 \\ \sin(\phi+\psi) & \cos(\phi+\psi) & 0 \\ -\cos(\phi+\psi) & \sin(\phi+\psi) & 0 \end{pmatrix}$$

Je zou als antwoord twee rotatiematrices verwachten, één om over de x-as te draaien en één om over de z-as te draaien. Dit is niet het geval, dus er is één *degree of freedom* verloren. Als je dus met een hoek van 90 of -90 graden gaat draaien om de y-as, en je vervolgens met ϕ graden om de x-as en ψ graden om de z-as wilt roteren, dan zal hij ψ en ϕ bij elkaar optellen en met die nieuwe hoek alleen om de x-as roteren. Bij andere conventies komt dit probleem ook voor. Bijvoorbeeld bij de zxz-conventie:

$$R = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Stel $\theta = 0$:

$$R = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\phi)\cos(\psi) - \sin(\phi)\sin(\psi) & -\cos(\phi)\sin(\psi) - \sin(\phi)\cos(\psi) & 0 \\ \sin(\phi)\cos(\psi) + \cos(\phi)\sin(\psi) & -\sin(\phi)\sin(\psi) + \cos(\phi)\cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\phi+\psi) & -\sin(\phi+\psi) & 0 \\ \sin(\phi+\psi) & \cos(\phi+\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Het probleem is op te lossen [**Vas09**], maar daar ga ik niet verder op in. De techniek om te roteren waarbij gimbal lock sowieso niet voorkomt ga ik bespreken in het volgend hoofdstuk.

6. Analyse

Matrix multiplicatie zit in $O(n^3)$. Maar omdat in alle rotatie-algoritmes de n heel klein is, kijk ik naar het aantal addities, multiplicaties en andere operaties die uitgevoerd moeten worden. Omdat er bij alle methoden weinig geheugen wordt gebruikt zal ik geheugengebruik weg laten uit de analyse, omdat het maar een zeer kleine invloed heeft op de rekentijd.

Om een 4×4 matrix met een 4×1 matrix te vermenigvuldigen zijn er 16 multiplicaties en 12 addities nodig. Bij een multiplicatie met twee 4×4 matrices is dat vier keer zo veel, en dus 64 multiplicaties en 48 addities. Om in drie dimensies te transformeren worden er drie 4×4 matrices met elkaar vermenigvuldigd. En de uitkomst daarvan wordt vermenigvuldigd met een 4×1 matrix, dus in totaal worden er $2 \cdot 64 + 16 = 144$ multiplicaties uitgevoerd en $2 \cdot 48 + 12 = 108$ addities. Verder zijn er nog 12 sinus en cosinus operaties.

Bij de $\odot$-operatie in Keyframing worden 3 e-machten, 3 logaritmes en 3 multiplicaties gebruikt, en bij de $\oplus$-operatie 6 logaritmes, 3 addities en 3 e-machten. Voor de gehele interpolatie kan je de aantallen in het volgend tabel zien:

	mult.	add.	cos/sin	e-machten	log	delingen
Euler hoeken	144	108	12	0	0	0
Keyframing	6	4	0	9	12	0

Problemen waarmee rekening gehouden moet worden als men Euler hoeken en deze keyframing methode gebruikt, zijn als volgend: Gimbal lock is nog niet opgelost, als men hier rekening mee wilt houden om goede animaties te krijgen moeten er extra stappen ondernomen worden en dat zal weer ten koste gaan van de efficiëntie. Ook geeft het keyframing nog wat problemen, doordat componenten van vectoren kleiner dan 0 niet gebruikt kunnen worden door het logaritme wat gebruikt wordt. Dit kan opgelost

worden door net zo lang translaties uit te voeren totdat alle componenten positief zijn, het algoritme uit te voeren, en dan de translatie terug uit te voeren, maar dat gaat ook weer ten koste van de efficiëntie.

7. Overige transformaties

Met deze vier bij vier matrices kunnen ook andere transformaties uitgevoerd worden. De volgende transformaties worden zo vaak gebruikt dat ik deze ter volledigheid vermeldt.

7.1. Scaling. Met scaling kunnen objecten simpelweg groter of kleiner gemaakt worden:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

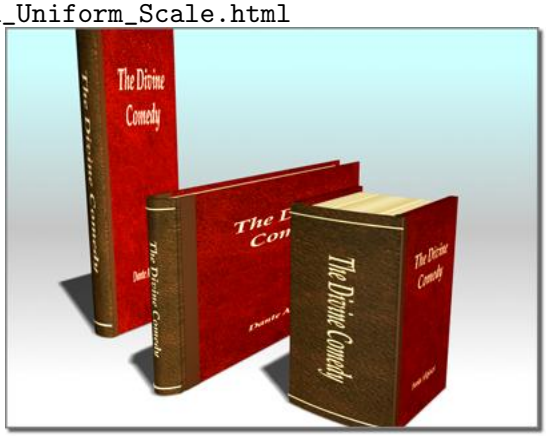
Dit komt neer op:

$$x' = \beta_x x$$
$$y' = \beta_y y$$
$$z' = \beta_z z$$

Als $\beta_x = \beta_y = \beta_z$ dan is het scalen uniform (de verhoudingen van het object blijven hetzelfde). Als $\beta > 0$, dan wordt het object groter en als $0 \leq \beta < 1$, dan wordt het object kleiner.

Als niet alle β's gelijk zijn, is het scalen non-uniform en wordt het object uitgerekt, zoals in figuur 2.3.

FIGUUR 2.3. Bron:`www.3dmax-tutorials.com/Select_and_Non_Uniform_Scale.html`

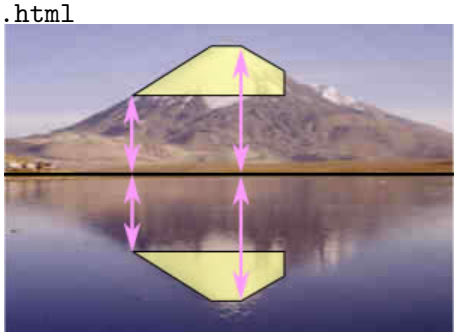


Scaling wordt vooral gebruikt om het 3D-effect te vergroten, een object in de verte is immers kleiner dan hetzelfde object in de voorgrond.

7.2. Reflecties. Als in de vorige subsectie $\beta < 0$ geldt, dan wordt er een reflectie uitgevoerd.

Deze transformaties worden bijvoorbeeld gebruikt voor reflecties van een object in het water, dan heb je een reflectie in de x-as nodig, zoals in figuur 2.4.

FIGUUR 2.4. Bron:`www.mathisfun.com/geometry/reflection.html`



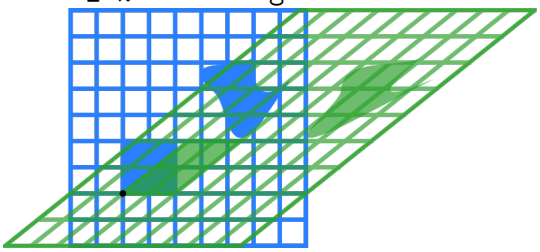
Een reflectie in een willekeurige as is iets ingewikkelder:

$$\text{Ref}_l(v) = 2\frac{v \cdot l}{l \cdot l} l$$

Waarbij v de vector is die moet worden gereflecteerd in de vector l. Omdat dit niets met matrices te maken heeft zal ik hier niet verder op in gaan.

7.3. Shearing. Met behulp van shearing kan je objecten 'kantelen', zoals te zien is in afbeelding 2.3.

FIGUUR 2.5. Bron:`en.wikipedia.org/wiki/File:VerticalShear_m%3D1.25.svg`



Je kan langs de x-as, y-as of de z-as shearen in 3D. Dat kan op de volgende manier, bijvoorbeeld langs de z-as:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Dit komt dus neer op:

$$x' = x + az$$
$$y' = y + bz$$
$$z' = z$$

Langs andere assen komt dit op hetzelfde neer. Shearing in 2D kan bijvoorbeeld gebruikt worden om fonts schuingedrukt (*italic*) af te drukken.

HOOFDSTUK 3

Quaternionen

Quaternionen, aangeduid met het symbool $\mathbb{H}$, zijn een uitbreiding op de complexe getallen. Ze zijn bedacht in 1843 bedacht door de wiskundige William Rowan Hamilton. Ze waren eigenlijk bedoelt voor gebruik in de mechanica, maar later bleek dat ze ook handig waren om vectoren te roteren in drie dimensies [**CS03**].

1. Aritmetiek

Een getal $q \in \mathbb{H}$ is een getal met de volgende vorm: $q = q_0 + q_1 i + q_2 j + q_3 k$ waarbij q_0, q_1, q_2, $q_3 \in \mathbb{R}$ en i, j en k verschillende imaginaire getallen zijn. Het getal q is ook te schrijven als een vector $q = (q_0, q_1, q_2, q_3) = (q_0, \vec{q})$, deze twee notaties ga ik in het vervolg aanhouden. Ik ga de rekenregels niet verklaren of bewijzen, maar hier is een opsomming [**CS03**]:

De rekenregels voor de imaginaire getallen zijn als volgt:

$$\boxed{i^2 = j^2 = k^2 = -1, \quad ij = k, \quad jk = i, \quad ki = j, \quad ji = -k, \quad kj = -i, \quad ik = -j}$$

Optellen:

$$p + q = (p_0 + q_0, \vec{p} + \vec{q})$$

Vermenigvuldigen:

$$\begin{aligned} pq &= (p_0, \vec{p})(q_0, \vec{q}) \\ &= (p_0 q_0 - \vec{p} \cdot \vec{q}, p_0 \vec{q} + q_0 \vec{p} + \vec{p} \times \vec{q}) \\ &= \begin{pmatrix} p_0 q_0 - p_1 q_1 - p_2 q_2 - p_3 q_3 \\ p_1 q_0 + p_0 q_1 + p_2 q_3 - p_3 q_2 \\ p_2 q_0 + p_0 q_2 + p_3 q_1 - p_1 q_3 \\ p_3 q_0 + p_0 q_3 + p_1 q_2 - p_2 q_1 \end{pmatrix} \end{aligned} \tag{3}$$

De geconjugeerde:

$$\overline{q} = (q_0, -\vec{q})$$

De inverse:

$$q^{-1} = \frac{(q_0, -q_1, -q_2, -q_3)}{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \frac{\overline{q}}{\|q\|^2} \tag{4}$$

2. Rotatie

De imaginaire eenheden i, j, en k van een quaternion kunnen gezien worden als de coördinaten van de x, y en z-as [**Que83**]. Dus als het reële deel van een quaternion 0 is, dan hebben we een vector in $\mathbb{R}^3$:

$$q = 0 + xi + yj + zk$$

De absolute waarde hiervan is $|q| = x^2 + y^2 + z^2$.
De lengte van q is $\|q\| = \sqrt{|q|}$.
Uit (3) volgt dat het product van twee vectoren het volgende is:

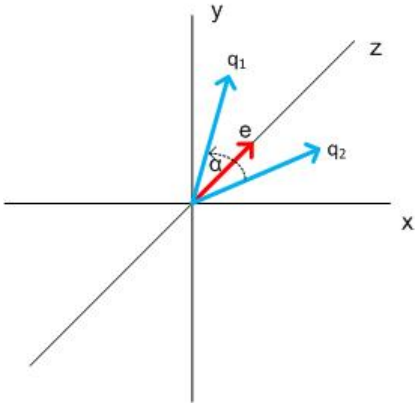
$$\begin{aligned} q_1 q_2 &= -x_1x_2 - y_1y_2 - z_1z_2 \\ &\quad + (y_1z_2 - z_1y_2)i \\ &\quad + (z_1x_2 - x_1z_2)j \\ &\quad + (x_1y_2 - y_1x_2)k \\ &= -x_1x_2 - y_1y_2 - z_1z_2 + \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} \end{aligned}$$

Omdat we nu eigenlijk met vectoren bezig zijn kunnen we nu het inproduct en het kruisproduct definiëren:

$$x_1x_2 + y_1y_2 + z_1z_2 = q_1 \cdot q_2 = \|q_1\|\|q_2\| \cos(\alpha)$$

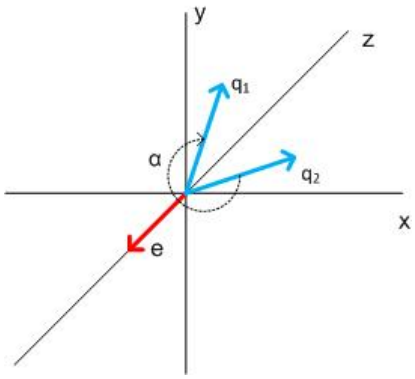
$$\begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = q_1 \times q_2 = e\|q_1\|\|q_2\| \sin(\alpha)$$

FIGUUR 3.1.



Waarbij α de hoek tussen de twee vectoren q_1 en q_2 is. En e is de eenheidsvector die loodrecht op beide vectoren staat. Als q_1 en q_2 dezelfde lengte hebben is er al een soort van rotatie (met de klok mee en met de klok

FIGUUR 3.2.



tegen) te zien om de z-as, zie figuur 3.1 em 3.2. De volgende afleiding komt in zijn geheel uit [**Que83**].

Het product is dus te schrijven als:

$$q_1 q_2 = -\|q_1\|\|q_2\|\cos(\alpha) + e\|q_1\|\|q_2\|\sin(\alpha) \tag{5}$$

Waarbij het niet uit maakt of de grote, of de kleine hoek gekozen wordt.

Als we nu q_2^{-1} nemen in plaats van q_2, dan volgt nu uit (4) en (5):

$$q_1 q_2^{-1} = \frac{\|q_1\|}{\|q_2\|}\cos(\alpha) - e\frac{\|q_1\|}{\|q_2\|}\sin(\alpha) \tag{6}$$

Als we nu e vermenigvuldigen met q_2^{-1}, dan volgt uit (6):

$$eq_2^{-1} = \frac{1}{\|q\|}(\cos(\alpha) + f\sin(\alpha)) \tag{7}$$

Waarbij f de vector is die loodrecht op e en q_2^{-1} staat. Omdat f een eenheidsvector is geldt $f^{-1} = -f$. Uit (7) volgt:

$$\begin{aligned}(eq_2^{-1})^{-1} &= \|q_2\|(\cos(\alpha) - f\sin(\alpha)) \\ e(eq_2^{-1})^{-1} &= \|q_2\|(e\cos(\alpha) - ef\sin(\alpha)) \\ &= \|q_2\|(e\cos(\alpha) + ef^{-1}\sin(\alpha))\end{aligned}$$

Uit (6) volgt:

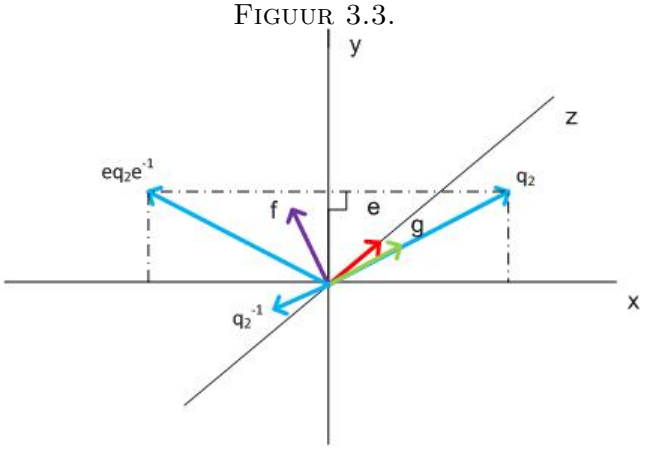
$$ef^{-1} = \cos(\frac{\pi}{2}) + g\sin(\frac{\pi}{2}) = g$$

Waarbij g de eenheidsvector loodrecht op e en f is, dus:

$$e(eq_2^{-1})^{-1} = eq_2e^{-1} = \|q_2\|(e\cos(\alpha) + g\sin(\alpha))$$

eq_2e^{-1} is nu het spiegelbeeld van q_2 in de lijn e, zoals te zien is in figuur 3.3.

FIGUUR 3.3.

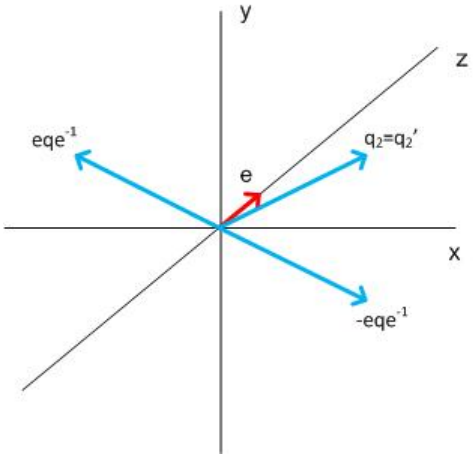


Nu weten we:

$$
\begin{aligned}
q_2' &= \frac{1}{2}(q_2 - eq_2e^{-1}) \\
q_2'' &= \frac{1}{2}(q_2 + eq_2e^{-1})
\end{aligned}
\tag{8}
$$

q_2' is hier de component van q_2 loodrecht op e en q_2'' is hier de componenten parallel aan e, zie figuur 3.4. Merk op dat de q_2'' in het voorbeeld de nul-quaternion is.

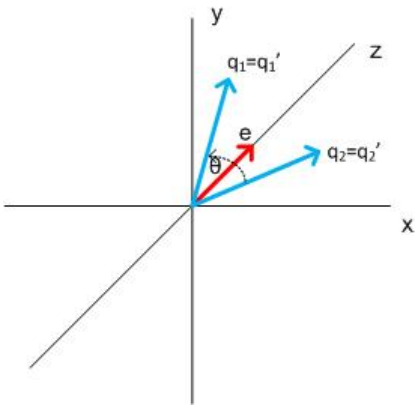
FIGUUR 3.4.



Nu gaan we q_2 roteren om e, met een hoek van θ zodat we q_1 weer krijgen, zie figuur 3.5. We weten van (8) dat:

$$q_1' = \frac{1}{2}(q_1 - eq_1e^{-1})$$
$$q_2' = \frac{1}{2}(q_2 - eq_2e^{-1})$$

FIGUUR 3.5.



Nu kunnen we de formule om te roteren afleiden. Uit (6) en $\|q_1'\| = \|q_2'\|$ volgt:

$$q_1'q_2'^{-1} = \cos(\theta) + e\sin(\theta)$$
$$q_1' = (\cos(\theta) + e\sin(\theta))q_2'$$

$$\frac{1}{2}(q_1 - eq_1e^{-1}) = (\cos(\theta) + e\sin(\theta))(\frac{1}{2}(q_2 - eq_2e^{-1})) \tag{9}$$

We weten:

$$\frac{1}{2}(q_1 + eq_1e^{-1}) = \frac{1}{2}(q_2 + eq_2e^{-1}) \tag{10}$$

Het optellen van (9) en (10) geeft:

$$q_1 = (\cos(\frac{\theta}{2}) + e\sin(\frac{\theta}{2}))q_2(\cos(\frac{\theta}{2}) - e\sin(\frac{\theta}{2}))$$

Hieruit volgt de formule om met quaternionen te roteren:

$$\boxed{p' = qpq^{-1}}$$

Waarbij p' de geroteerde vector is, p de originele vector en q de rotatie-quaternion: $q = (\cos(\frac{\theta}{2}), n\sin(\frac{\theta}{2}))$, waarbij θ de hoek is waarmee geroteerd wordt en n de as waarover geroteerd wordt.

3. SLERP

SLERP staat voor *spherical linear interpolation* [**Wat00**] en wordt gebruikt om rotatie te animeren. Stel, we hebben twee 2D eenheidsvectoren A en B die het begin- en eindpunt zijn van een rotatie om de z-as met hoek Ω, en we willen een vector P hebben die een hoek maakt met A van θ, die tussen A en B op de cirkel ligt (zie figuur 3.6), dan kan P berekent worden door de volgende lineaire combinatie te gebruiken:

$$P = \alpha A + \beta B$$

Nu zijn α en β als volgt te berekenen:

$$\left(\begin{array}{cc|c} a_1 & b_1 & p_1 \\ a_2 & b_2 & p_2 \end{array}\right)$$

$$R_1 := R_1 - \frac{a_1}{a_2} R_2$$

$$\left(\begin{array}{cc|c} 0 & b_1 - \frac{a_1 b_2}{a_2} & p_1 - \frac{a_1 p_2}{a_2} \\ a_2 & b_2 & p_2 \end{array}\right).$$

$$(b_1 - \frac{a_1 b_2}{a_2})\beta = p_1 - \frac{a_1 p_2}{a_2}$$

$$\frac{a_2 b_1 - a_1 b_2}{a_2}\beta = p_1 - \frac{a_1 p_2}{a_2}$$

$$(a_2 b_1 - a_1 b_2)\beta = a_2 p_1 - a_1 p_2$$

$$\beta = \frac{a_2 p_1 - a_1 p_2}{a_2 b_1 - a_1 b_2}$$

$$= \frac{A \times P}{A \times B}$$

$$= \frac{\sin(\theta)}{\sin(\Omega)}$$

$$a_2 \alpha = p_2 - \frac{b_1(a_2 p_1 - a_1 p_2)}{a_2 b_1 - a_1 b_2}$$

$$\alpha = \frac{p_2(a_2 b_1 - a_1 b_2) - b_1(a_2 p_1 - a_1 p_2)}{a_2(a_2 b_1 - a_1 b_2)}$$

$$= \frac{a_2 b_1 p_2 - b_2 a_2 p_1}{a_2 a_2 b_1 - a_2 a_1 b_2}$$

$$= \frac{b_1 p_2 - b_2 p_1}{a_2 b_1 - a_1 b_2}$$

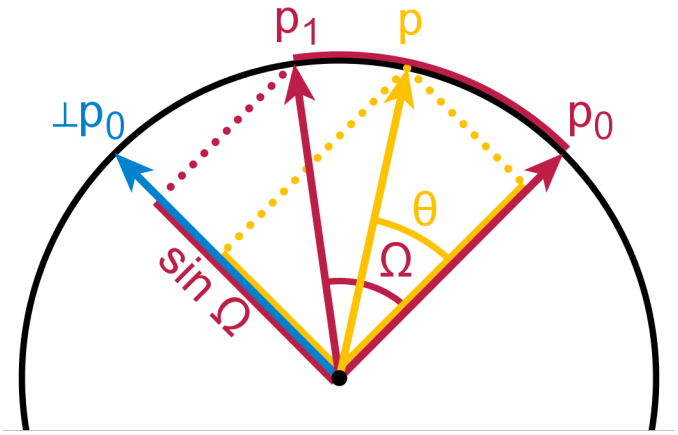
$$= \frac{B \times P}{A \times B}$$

$$= \frac{\sin(\Omega - \theta)}{\sin(\Omega)}$$

Door θ te schrijven als Ωt, waarbij $t \in [0, 1]$, en de lineaire combinatie te generalizeren naar vier dimensies hebben we de interpolatie te pakken:

$$slerp(q_1, q_2, t) = \frac{\sin((1-t)\Omega)}{\sin(\Omega)} q_1 + \frac{\sin(\Omega t)}{\sin(\Omega)} q_2$$

FIGUUR 3.6. Bron: `en.wikipedia.org/wiki/File:Slerp_factor_explanation.png`



4. Analyse

Voor een rotatie met quaternionen worden twee quaternion multiplicaties gebruikt. Een multiplicatie met quaternionen heeft even veel addities en multiplicaties als een matrix multiplicatie met een 4×4 matrix en een 4×1 matrix, dus 16 multiplicaties en 12 addities. Dit maakt een transformatie uitreken veel efficiënter dan met rotatiematrices, namelijk $2 \cdot 16 + 3 = 35$ multiplicaties en $2 \cdot 12 = 24$ addities. De drie extra multiplicaties komen door het uitrekenen van $\overline{p}$ in $q' = pq\overline{p}$, waar er drie keer met -1 moet worden vermenigvuldigd om de complex geconjugeerde uit te rekenen. Verder moet er voor p 4 keer een sinus of cosinus uitgerekend worden, en voor $\overline{p}$ ook 4 keer.

Dus:

	mult.	add.	sin/cos	e-machten	log	delingen
Euler hoeken	144	108	12	0	0	0
Keyframing	6	4	0	9	12	0
Quaternionen	35	24	8	0	0	8
SLERP	10	5	4	0	0	2

Zoals te zien is in de tabel zijn quaternionen veel efficiënter dan Euler hoeken qua addities en multiplicaties, gimbal lock is nu ook niet meer aan de orde. Ook is SLERP een stuk efficiënter en heeft geen last van logaritmen.

Een nadeel van quaternionen is dat rotaties gevolgd door een translatie (nog) niet uitgevoerd kunnen worden. Verder zijn quaternionen veel lastiger

te begrijpen en zijn ze een stuk minder intutief dan Euler hoeken. Quaternionen zijn ook wat gevoeliger voor afrondfouten als een quaternion niet genormaliseerd wordt. Dit wil zeggen dat het reële deel weer nul moet worden als het niet nul is. Als bijvoorbeeld de vector $(1, 2, 3)$ na een aantal berekeningen wordt gerepresenteerd wordt door het quaternion $(10, 1, 2, 3)$, moet dat quaternion weer genormaliseerd worden: $(0, 1, 2, 3)$. Het reële deel kan anders te veel effect hebben op verdere berekeningen.

HOOFDSTUK 4

Dual-Quaternionen

1. Dual getallen

Dual getallen zijn geïntroduceert door de wiskundige William Clifford in 1882 [**Ken12**] , ze lijken heel erg op complexe getallen. Ze hebben een reëel deel en een dual deel:

$$z = r + d\epsilon, \quad \epsilon^2 = 0 \text{ en } \epsilon \neq 0$$

De aritmetische operaties lijken op de operaties van de complexe getallen [**Ken12**]:

Additie:

$$(r_1 + d_1\epsilon) + (r_2 + d_2\epsilon) = (r_1 + r_2) + (d_1 + d_2)\epsilon$$

Multiplicatie:

$$(r_1 + d_1\epsilon)(r_2 + d_2\epsilon) = r_1r_2 + r_1d_2\epsilon + r_2d_1\epsilon + d_1d_2\epsilon^2 = r_1r_2 + (r_1d_2 + r_2d_1)\epsilon$$

2. Dual-Quaternionen

Een dual-quaternion is een combinatie van een quaternion en een dual getal:

$$q = q_r + q_d\epsilon, \text{ waarbij } q_r, q_d \in \mathbb{H}$$

De aritmetiek komt op hetzelfde neer als bij een normaal dual getal.

Scalair vermenigvuldigen:

$$sq = sq_r + sq_d\epsilon$$

Additie:

$$q_1 + q_2 = (q_{r1} + q_{r2}) + (q_{d1} + q_{d2})\epsilon$$

Multiplicatie:

$$q_1q_2 = q_{r1}q_{r2} + (q_{r1}q_{d2} + q_{d1}q_{r2})\epsilon$$

Er zijn drie types van de geconjugeerde [**KCOZ06**] , de versie die we voor translatie nodig hebben is:

$$q^\dagger = \overline{q_r} - \overline{q_d}\epsilon$$

Een andere versie van de geconjugeerde die gebruikt wordt in de eigenschappen die hierna komen is $\overline{q} = \overline{q_r} + \overline{q_d}\epsilon$.

Lengte:

$$\|q\| = q\overline{q}$$

Eigenschappen eenheidsdual-quaternion:

$$\|q\| = 1$$
$$\overline{q_r} q_d + \overline{q_d} q_r = 0$$

3. Rotatie

Rotatie werkt hetzelfde als bij normale quaternionen, om alleen rotatie uit te voeren wordt de volgende dual-quaternion gebruikt [**KCOZ06**]:

$$q_r = (\cos(\frac{\theta}{2}), n\sin(\frac{\theta}{2})) + (0,0,0,0)\epsilon$$

Als vervolgens dezelfde formule als voor normale quaternion rotatie wordt gebruikt, krijgen we dezelfde rotatie als in het vorig hoofdstuk:

$$p' = q_r p q_r^\dagger$$

Merk op dat de inverse van een eenheidsdual-quaternion gelijk is aan de geconjugeerde van die eenheidsdual-quaternion.

4. Translatie

Een voordeel van het gebruik van dual-quaternionen is dat translatie er ook mee uitgevoerd kan worden. Hiervoor gebruiken we het dualgedeelte van het dualquaternion, waarbij het reëele gedeelte de identiteitsquaternion is [**Ken12**]:

$$q_t = (1,0,0,0) + (0, \frac{T_x}{2}, \frac{T_y}{2}, \frac{T_z}{2})\epsilon$$

Dan kunnen we weer de bekende formule uitvoeren:

$$p' = q_t p q_t^\dagger$$

Een uitwerking:

$$p = (1,0,0,0) + (0,x,y,z)\epsilon$$
$$q_t = (1,0,0,0) + (0, \frac{T_x}{2}, \frac{T_y}{2}, \frac{T_z}{2})\epsilon$$
$$q_t^\dagger = (1,0,0,0) - (0, -\frac{T_x}{2}, -\frac{T_y}{2}, -\frac{T_z}{2})\epsilon$$
$$q_t p = (1,0,0,0) + (0, x + \frac{T_x}{2}, y + \frac{T_y}{2}, z + \frac{T_z}{2})\epsilon$$
$$((1,0,0,0) + (0, x + \frac{T_x}{2}, y + \frac{T_y}{2}, z + \frac{T_z}{2})\epsilon)q_t^\dagger = (1,0,0,0) + (0, x + T_x, y + T_y, z + T_z)\epsilon$$

Om nu rotatie en translatie te combineren moeten q_r en q_t vermenigvuldigd worden:

$$q = q_r q_t = (\cos(\frac{\theta}{2}), n\sin(\frac{\theta}{2})) + (0, \frac{T_x}{2}, \frac{T_y}{2}, \frac{T_z}{2})\epsilon$$

Om vervolgens weer de formule $p' = qpq^\dagger$ te gebruiken.

5. Analyse

Er worden bij de berekening van rotatie en translatie $3 \cdot 2$ quaternion-multiplicaties uitgevoerd. Dus in totaal $3 \cdot 2 \cdot 16 = 96$ multiplicaties. Plus nog 10 multiplicaties om de complex geconjugeerde uit te rekenen geeft 110 multiplicaties. Hetzelfde geldt voor addities: $3 \cdot 2 \cdot 12 = 72$ addities. Er zijn drie delingen nodig voor de translatie, plus nog de acht delingen in q en $q^\dagger$ geeft 11 delingen. En dan hebben we nog $2 \cdot 4 = 8$ sinus en cosinus berekeningen. EN dan krijgen we het volgend tabel:

	mult.	add.	sin/cos	e-machten	log	delingen
Euler hoeken	144	108	12	0	0	0
Keyframing	6	4	0	9	12	0
Quaternionen	35	24	8	0	0	8
Dual-Quaternionen	110	72	8	0	0	11
SLERP	10	5	4	0	0	2

Dual-quaternionen zijn dus nog steeds efficiënter dan Euler hoeken, en anders dan quaternionen kunnen we er nu ook translaties mee uitvoeren.

HOOFDSTUK 5

Hyper-dual-quaternionen

In dit hoofdstuk zal ik het gebruik van hyper-dual-quaternionen bij transformaties onderzoeken.

1. Hyper-dual getallen

Hyperdual getallen zijn getallen van de volgende vorm [**FA11**]:

$$x = a + b\epsilon_1 + c\epsilon_2 + d\epsilon_1\epsilon_2, \text{ waarbij}$$
$$\epsilon_1^2 = \epsilon_2^2 = (\epsilon_1\epsilon_2)^2 = 0 \text{ en } \epsilon_1 \neq \epsilon_2 \neq \epsilon_1\epsilon_2 \neq 0$$

We nemen de volgende twee hyper-dual getallen:

$$a = a_1 + a_2\epsilon_1 + a_3\epsilon_2 + a_4\epsilon_1\epsilon_2 \text{ en } b = b_1 + b_2\epsilon_1 + b_3\epsilon_2 + b_4\epsilon_1\epsilon_2$$

Met onder andere de volgende rekenregels:
Additie:

$$a + b = (a_1 + b_1) + (a_2 + b_2)\epsilon_1 + (a_3 + b_3)\epsilon_2 + (a_4 + b_4)\epsilon_1\epsilon_2$$

Multiplicatie:

$$ab = (a_1b_1) + (a_1b_2 + a_2b_1)\epsilon_1 + (a_1b_3 + a_3b_1)\epsilon_2 + (a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1)\epsilon_1\epsilon_2$$

Inverse:

$$a^{-1} = \frac{1}{a_1} - \frac{a_2}{a_1^2}\epsilon_1 - \frac{a_3}{a_1^2}\epsilon_2 + \left(\frac{2a_2a_3}{a_1^3} - \frac{a_4}{a_1^2}\right)\epsilon_1\epsilon_2$$

2. Hyper-dual-quaternionen

Hyper-dual-quaternionen hebben dezelfde vorm als hyper-dual getallen, alleen zijn a, b, c en d geen rele getallen, maar quaternionen. Dus [**FA11**]:

$$x = a + b\epsilon_1 + c\epsilon_2 + d\epsilon_1\epsilon_2, \text{ waarbij } a, b, c, d \in \mathbb{H}$$

De complex geconjugeerde van een hyper-dual-quaternion wordt gebruikt voor de rotatie:

$$x^\dagger = \overline{a} - \overline{b}\epsilon_1 - \overline{c}\epsilon_2 - \overline{d}\epsilon_1\epsilon_2$$

3. Rotatie

Rotatie komt op hetzelfde neer als bij dual-quaternionen, dus een vector wordt gerepresenteerd als:

$$p = (1,0,0,0) + (0,0,0,0)\epsilon_1 + (0,0,0,0)\epsilon_2 + (0,x,y,z)\epsilon_1\epsilon_2$$

En de rotatiehyper-dual-quaternion:

$$q_r = (\cos(\frac{\theta}{2}), n\sin(\frac{\theta}{2})) + (0,0,0,0)\epsilon_1 + (0,0,0,0)\epsilon_2 + (0,0,0,0)\epsilon_1\epsilon_2$$

4. Translatie

Translatie werkt ook precies hetzelfde als bij dual-quaternionen, samen met rotatie wordt q het volgende:

$$q = (\cos(\frac{\theta}{2}), n\sin(\frac{\theta}{2})) + (0,0,0,0)\epsilon_1 + (0,0,0,0)\epsilon_2 + (0, \frac{T_x}{2}, \frac{T_y}{2}, \frac{T_z}{2})\epsilon_1\epsilon_2$$

Weer kan dan $p' = qpq^\dagger$ gebruikt worden.

5. Analyse

Er worden 9 quaternion-multiplicaties uitgevoerd voor 1 hyper-dual-quaternion-multiplicatie, en 24 multiplicaties voor de complex geconjugeerde, dus $9 \cdot 2 \cdot 16 + 24 = 312$ multiplicaties. Ook zijn er 9 maal zo veel additions, dus 216 additions. De rest van de operaties blijven op hetzelfde aantal als dual-quaternionen.

	mult.	add.	sin/cos	e-machten	log	delingen
Euler hoeken	144	108	12	0	0	0
Keyframing	6	4	0	9	12	0
Quaternionen	35	24	8	0	0	8
Dual-Quaternionen	103	72	8	0	0	11
Hyper-dual-quaternionen	312	216	8	0	0	11
SLERP	10	5	4	0	0	2

Natuurlijk hebben hyper-dual-quaternionen alleen nut als de andere twee quaternionen in q ook gebruikt worden (die nu allebei nul zijn), zodat meerdere transformaties uitgevoerd kunnen worden. Meer daarover in de volgende sectie.

6. Overige transformaties

6.1. Scaling. Uniform scaling hoeft niet per se met hyper-dual-quaternionen, dat kan veel efficiënter met normale quaternionen, en dat is triviaal met scalair vermenigvuldigen:

$$(0, \beta x, \beta y, \beta z) = (\beta,0,0,0)(0,x,y,z) = \beta(0,x,y,z)$$

Uniform scaling kan tegelijkertijd met rotatie, in plaats van $p' = qpq^{-1}$ hebben we $p' = \beta qpq^{-1}$ met β de scalingsfactor. Als p en q dual-quaternionen

zijn, dan kan scaling ook met rotatie en translatie. Hier moet worden opgemerkt dat er eerst geroteerd en getransleerd wordt, en dan pas gescaled.

Non-uniform scaling ligt wat moeilijker, dat kan niet met quaternionen, in ieder geval niet met de normale quaternion-operatoren. Met de inverse, complex geconjugeerde en additie kun je sowieso niets, omdat je in ieder geval twee getallen wilt vermenigvuldigen. Maar met vermenigvuldiging lukt het ook niet, omdat alle componenten invloed op elkaar hebben en dat wil je niet bij scaling:

$$pq = \begin{pmatrix} p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3 \\ p_1q_0 + p_0q_1 + p_2q_3 - p_3q_2 \\ p_2q_0 + p_0q_2 + p_3q_1 - p_1q_3 \\ p_3q_0 + p_0q_3 + p_1q_2 - p_2q_1 \end{pmatrix}$$

Als je dit toch met quaternionen wilt gaan doen, zul je een nieuwe operator moeten definiëren. In dit geval componentsgewijs vermenigvuldigen met de $\times$-operator. Die wordt dan als volgt gedefinieerd:

$$p \times q = (p_0, p_1, p_2, p_3) \times (q_0, q_1, q_2, q_3) = (p_0q_0, p_1q_1, p_2q_2, p_3q_3)$$

Dan kunnen we non-uniform scalen met quaternionen als volgt:

$$(0, \beta_x x, \beta_y y, \beta_z z) = (0, \beta_x, \beta_y, \beta_z) \times (0, x, y, z)$$

Deze operator is wiskundig echter niet mogelijk, want $i^2 = j^2 = k^2 = -1$, dus uit deze operator zou eigenlijk een reëel getal moeten komen. Voor scaling zou het heel handig zijn geweest. Als we het wiskundig correct willen houden, zouden we hiervoor dus een andere oplossing moeten vinden.

Met dual-quaternionen lukt het non-uniform scalen ook niet. Hieronder nog even de vermenigvuldiging:

$$q_1q_2 = q_{r1}q_{r2} + (q_{r1}q_{d2} + q_{d1}q_{r2})\epsilon$$

We hebben hierboven al gezien dat we niks met $q_{r1}q_{r2}$ kunnen. Dus dan zou de uitkomst in het duale gedeelte van de dual-quaternion moeten komen. We moeten het reële gedeelte van de quaternionen 0 laten, anders krijgen we uniform scaling. Dus dan komt $(q_{r1}q_{d2} + q_{d1}q_{r2})\epsilon$ op het volgende neer, als we q_{r1} a noemen, q_{d2} b noemen, q_{d1} c noemen en q_{r2} d noemen:

$$\begin{pmatrix} 0 \\ a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} + \begin{pmatrix} 0 \\ c_2d_3 - c_3d_2 \\ c_3d_1 - c_1d_3 \\ c_1d_2 - c_2d_1 \end{pmatrix} = \begin{pmatrix} 0 \\ \beta_x x \\ \beta_y y \\ \beta_z z \end{pmatrix}$$

Hieraan is te zien dat je nooit de correcte uitkomst kan krijgen, omdat er altijd iets van de uitkomst wordt afgetrokken of opgeteld. Als je bijvoorbeeld neemt $a_2b_3 = c_2d_3 = \frac{1}{2}\beta_x x$, $a_3b_1 = c_3d_1 = \frac{1}{2}\beta_y y$ en $a_1b_2 = c_1d_2 = \frac{1}{2}\beta_z z$ dan

krijg je:

$$\begin{pmatrix} 0 \\ \frac{1}{2}\beta_x x - \frac{1}{2}\beta_y z \\ \frac{1}{2}\beta_y y - \frac{1}{2}\beta_z x \\ \frac{1}{2}\beta_z z - \frac{1}{2}\beta_x y \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2}\beta_x x - \frac{1}{2}\beta_y z \\ \frac{1}{2}\beta_y y - \frac{1}{2}\beta_z x \\ \frac{1}{2}\beta_z z - \frac{1}{2}\beta_x y \end{pmatrix} = \begin{pmatrix} 0 \\ \beta_x x - \beta_y z \\ \beta_y y - \beta_z x \\ \beta_z z - \beta_x y \end{pmatrix}$$

Als je deze fouten tegen elkaar weg wilt laten vallen lukt dat ook niet, want dan vallen de juiste uitkomsten ook automatisch tegen elkaar weg:

$$\begin{pmatrix} 0 \\ \frac{1}{2}\beta_x x - \frac{1}{2}\beta_y z \\ \frac{1}{2}\beta_y y - \frac{1}{2}\beta_z x \\ \frac{1}{2}\beta_z z - \frac{1}{2}\beta_x y \end{pmatrix} + \begin{pmatrix} 0 \\ -\frac{1}{2}\beta_x x + \frac{1}{2}\beta_y z \\ -\frac{1}{2}\beta_y y + \frac{1}{2}\beta_z x \\ -\frac{1}{2}\beta_z z + \frac{1}{2}\beta_x y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Dit probleem blijf je houden bij hyper-dual-quaternionen, ook al heb je de beschikking over meer quaternionen. Non-uniform scaling is daarom niet mogelijk met quaternionen.

6.2. Reflecties. Reflecties met $\beta < 0$ lukken daarom ook niet. De formule voor een reflectie in een willekeurige as kunnen we ook niet gebruiken, omdat het inproduct voor quaternionen niet gedefinieerd is. Er is wel een andere formule voor reflecties van quaternionen, die op een iets andere manier werkt [**CS03**]:

$$p' = qpq$$

Met p de originele quaternion, p' de gereflecteerde quaternion en q het normaal van de het vlak van reflectie. Als we bijvoorbeeld $(0, 1, 1, 1)$ in de x-as willen reflecteren, dan nemen we $q = (0, 0, 1, 0)$ en dan krijgen we $p' = (0, 0, 1, 0)(0, 1, 1, 1)(0, 0, 1, 0) = (0, 1, -1, 1)$.

Ook nu kunnen we tegelijkertijd uniform scalen, en dat gaat op dezelfde manier als bij scalen tijdens rotatie. Als we in het voorbeeld ook nog met factor β willen scalen, dan nemen we $p' = \beta qpq$.

Met dual-quaternionen en hyper-dual-quaternionen kan dit dus ook, maar verder kun je er niks speciaals mee.

6.3. Shearing. Bij shearing met quaternionen komt hetzelfde probleem als bij non-uniform scaling om de hoek kijken. Geen enkele quaternion-operatie kan deze transformatie bewerkstelligen. Zelfs met hyper-dual-quaternionen niet, omdat je nog steeds met dezelfde operaties werkt, alleen dan met meer quaternionen. Hyper-dual-quaternionen vallen dus een beetje tegen, qua efficiëntie en gebruik.

HOOFDSTUK 6

Implementatie

Voor deze scriptie is er ook een implementatie van quaternionen, dual-quaternionen en hyper-dual-quaternionen gemaakt. Deze is gemaakt met openGL in C++ en is te zien in de appendix. Omdat je geen eigen transformaties kan maken in openGL (zo ver ik weet in ieder geval) kon ik helaas geen complexe objecten roteren en transleren, zoals de bekende theepot. Mijn rotatiefuncties worden daardoor ook uitgevoerd door de processor, in plaats van de grafische kaart. Dit maakt uitspraken doen over de snelheid van de transformaties wat lastiger. Verder moet worden vermeldt dat de regel `#include<windows.h>` bovenaan de code toegevoegd moet worden, als de code in Windows gecompileerd wordt.

Ten eerste waren er me een aantal dingen opgevallen tijdens het implementeren, het belangrijkste daarvan is dat SLERP niet altijd blijkt te werken. Wiskundig klopt de formule, maar in de praktijk kan er een situatie voorkomen waar SLERP niet op berekend is. Stel we willen de quaternion $(0, 1, 0, 0)$ met een hoek van 90 graden roteren over de x-as (ook $(0, 1, 0, 0)$). Het is gemakkelijk in te zien dat de vector niet van de plaats af komt, maar als we SLERP het beginpunt en eindpunt van de rotatie meegeven met de bijbehorende 90 graden, met $t = \frac{1}{2}$ zal SLERP met een totaal verkeerd antwoord komen:

$$\begin{aligned} slerp((0,1,0,0),(0,1,0,0),\frac{1}{2}) &= \frac{\sin((1-\frac{1}{2})\frac{1}{2}\pi)}{\sin(\frac{1}{2}\pi)}(0,1,0,0) + \frac{\sin((\frac{1}{2}\pi)\frac{1}{2})}{\sin(\frac{1}{2}\pi)}(0,1,0,0) \\ &= \frac{1}{\sqrt{2}}(0,1,0,0) + \frac{1}{\sqrt{2}}(0,1,0,0) \\ &= (0, 2\sqrt{2}, 0, 0) \end{aligned}$$

De vector is dus langer geworden, iets wat bij *proper rigid body transformations* uit den boze is. Een gemakkelijke oplossing hiervoor is om een if-statement toe te voegen waarin je kijkt of q_1 en q_2 gelijk zijn, zo ja dan return je q_1, zo nee, dan voer je SLERP uit. Bij het vergelijken van q_1 en q_2 stuitte ik op een ander probleem. De floats die ik gebruikte bleken afrondfouten te bevatten, hier had ik in eerste instantie niet aan gedacht. Dit heb ik opgelost door een foutmarge ϵ te gebruiken. Dus een float x is gelijk aan een float y als $y - \epsilon \leq x \leq y + \epsilon$. In mijn implementatie heb ik de foutmarge $\epsilon = 0.00001$ gebruikt.

Een ander probleem met SLERP ontstaat als $\Omega = \pi$ of $\Omega = 2\pi$, dan wordt $\sin(\Omega) = 0$, en dan deel je dus door 0, wat natuurlijk niet mogelijk is. Maar vanwege die afrondfouten van floats wordt $\sin(\Omega)$ geen 0, maar komt het heel dicht bij 0, waardoor je deelt door een heel klein getal en uiteindelijk dus een heel groot getal krijgt. Hierdoor kan het komen dat het roterende object ineens heel erg uitgerekt wordt, zoals bij non-uniform scaling. Dit is natuurlijk niet de bedoeling. Ook hier moet dus rekening mee worden gehouden.

Een tweede belangrijke observatie die ik maakte is dat de standaard rotatiefunctie van openGL blijkbaar niet gimbal lock voorkomt. Als we bijvoorbeeld het volgende fragment code hebben:

```
glRotatef(90, 0, 0, 1);
glRotatef(45, 0, 1, 0);
drawLine(0, 0, 0, 0, 1, 0);
```

Dus we roteren de vector $(0, 1, 0)$ eerst met 90 graden om de z-as, vervolgens 45 graden om de y-as en vervolgens wordt de resulterende vector getekend op het scherm. Wat meteen opvalt is dat de tweede rotatie niet uitgevoerd wordt, dit komt door gimbal lock, wat wordt uitgelegd in hoofdstuk 2. Hier moet dus serieus rekening mee worden gehouden door de programmeurs.

Verder heb ik geprobeerd om zoveel mogelijk tussenstappen op te schrijven, zodat de code duidelijker is. Ook heb ik de

Als de code uitgevoerd wordt, dan is te zien dat de vectoren die gebruik maken van dual-quaternionen en hyper-dual-quaternionen allebei even snel roteren en transleren, ondanks het feit dat hyper-dual-quaternionen veel meer operaties moet uitvoeren. Als er al een snelheidsverschil is, dan is deze nihil en niet te zien. Dit zou kunnen komen doordat de extra operaties die uitgevoerd moeten worden, vaak 0 vermenigvuldigd met 0 is, en voor de processor niet veel moeite kosten.

Verder is te zien dat de vector die roteert met hulp van SLERP veel sneller roteert dan alle andere vectoren, SLERP is dus zeker bruikbaar als er een hogere snelheid gehaald moet worden, zonder dat de "vloeiendheid"van de animatie achteruit gaat.

De vector die gebruik maakt van de standaard rotatiefunctie van openGL roteert iets sneller dan de degenen die gebruik maken van dual-quaternionen en hyper-dual-quaternionen, ook al heeft het dezelfde rotatiesnelheid. Dit komt ongeveer overeen met het aantal operaties die uitgevoerd moeten worden, die zijn immers gelijk. Dat de rotatiefunctie van openGL toch iets sneller is dan die met dual-quaternionen en hyper-dual-quaternionen zou kunnen komen doordat deze gebruik maakt van de grafische kaart in plaats van de processor en dus een voordeel heeft. Verder is te zien dat de translaties even snel gaan, ook als er verschillende variabelen worden gebruikt in plaats van alleen `translate`. Dit was ook de verwachten omdat translaties geen moeilijke transformaties zijn om uit te voeren.

HOOFDSTUK 7

Conclusie

Hier nog even de resultaten van de analyses:

	mult.	add.	sin/cos	e-machten	log	delingen
Euler hoeken	144	108	12	0	0	0
Keyframing	6	4	0	9	12	0
Quaternionen	35	24	8	0	0	8
Dual-Quaternionen	103	72	8	0	0	11
Hyper-dual-quaternionen	312	216	8	0	0	11
SLERP	10	5	4	0	0	2

Als het puur over efficiëntie van rotaties gaat, dan komen normale quaternionen als beste uit de bus. Zeker als je ook de andere voordelen meerekent, zoals het geen last hebben van gimbal lock en het mindere geheugengebruik. Een quaternion heeft immers maar 4 floats nodig en een vier bij vier rotatiematrix 16 floats.

Rotatiematrices hebben ook het nadeel dat je de rotaties moeilijk kan interpoleren, doordat je op verschillende manieren op het eindpunt kan komen. Je kan bijvoorbeeld de vector $(1, 0, 0)$ met 90 graden over de z-as draaien om op $(0, 1, 0)$ uit te komen, maar ook door eerst 90 graden om de y-as te draaien en vervolgens 90 graden om de x-as. Dit probleem heb je bij quaternionen niet. En als je SLERP wilt gebruiken bij rotatiematrices moet je eerst het aantal graden tussen het begin- en eindpunt uitrekenen. Als je dan alleen de afzonderlijke Euler hoeken weet kost dat weer wat extra rekentijd.

Rotatiematrices hebben dan weer het voordeel dat ze relatief simpel te begrijpen zijn vergeleken met quaternionen en SLERP. Deze zijn voor animators die niet veel van wiskunde af weten misschien onbegrijpbaar en niet intuïtief.

Rotaties met dual-quaternionen hebben iets minder operaties nodig dan rotatiematrices. Je kan dan ook meteen translaties en uniform scaling uitvoeren. SLERP werkt ook bij deze transformaties, ook al zijn het geen pure rotaties. Dual-quaternionen hebben het voordeel dat ze net als quaternionen geen last hebben van gimbal lock.

SLERP zelf werkt zelf ook behoorlijk goed, gezien de snelheid die de van SLERP gebruikmakende vector haalt, vergeleken met de rotatiesnelheid van de andere vectoren in mijn implementatie. Het algoritme voor keyframing

uit [**Ale02**] kan hier niet tegen op. Vooral omdat het logaritmen en worteltrekken bevat, deze operaties zijn niet voor alle getallen gedefinieerd en kunnen dus in de praktijk veel problemen opleveren. Ook zijn er meerdere goede interpolatie-algoritmen voor quaternionen te vinden, zoals LERP en NLERP, die allebei hun voor- en nadelen hebben vergeleken met SLERP.

De grafische kaarten zijn nu nog gespecialiseerd in matrixvermenigvuldiging. Maar als de GPU's in de toekomst gespecialiseerd worden in quaternionen en dual-quaternionen kan er nog meer snelheid uit gehaald worden, zodat er meer rekentijd beschikbaar kan worden voor gedetailleerdere graphics en nog vloeiendere animaties. Dus vooral voor de gamesindustrie zijn deze quaternionen en dual-quaternionen van belang.

Het enige probleem is het feit dat non-uniform scaling en shearing nog niet met quaternionen of dual-quaternionen bewerkstelligd kan worden. Hier zou nog een oplossing voor gevonden moeten worden. Of men mijn wiskundig incorrecte "componentsgewijze vermenigvuldiging"zou willen toevoegen is nog maar de vraag.

In het tabel en de analyses is te lezen dat hyper-dual-quaternionen veel operaties vereisen, zonder dat ze extra transformaties of iets anders toevoegen aan wat dual-quaternionen ook al kunnen. Het gebruik hiervan is dus af te raden.

Bibliografie

[Ale02] Marc Alexa, *Linear combinations of transformations*, vol. 21, 2002, pp. 380–387.

[CS03] John H. Conway and Derek A. Smith, *On quaternions and octonions*, A K Peters, Ltd., 2003.

[FA11] Jeffrey A. Fike and Juan J. Alonso, *The development of hyper-dual numbers for exact second-derivative calculations*, 2011.

[Gra06] Matt Gravelle, *Quaternions and their applications to rotation in 3d space*, 2006.

[KCOZ06] Ladislav Kavan, Steven Collins, Carol O'Sullivan, and Jiri Zara, *Dual quaternions for rigid transformation blending*, 2006.

[Ken12] Ben Kenwright, *A beginners guide to dual-quaternions*, 2012.

[Muk02] R. Mukundan, *Quaternions: From classical mechanics to computer graphics, and beyond*, 2002.

[Que83] H. Quee, *Quaternion algebra applied to polygon theory in three dimensional space*, Publications on Geodesy **7** (1983), no. 2.

[Vas09] Gergely Vass, *Avoiding gimbal lock*, Computer Graphics World **32** (2009), no. 6, 10–11.

[Wat00] Alan Watt, *3d computer graphics*, 3 ed., Addison-Wesley, 2000.

BIJLAGE A

De code

Sommige regels zijn over meerdere regels verdeeld, omdat het anders niet op de pagina paste.

```
//Bram Arends, 2013

#include <GL/glut.h>
#include <GL/gl.h>
#include <stdlib.h>
#include <iostream>
#include <stdio.h>
#include <stdarg.h>
#include <math.h>
#include <string.h>

//Size of window
#define WINSIZE_X     500
#define WINSIZE_Y     500
//Position of window
#define WINPOS_X      100
#define WINPOS_Y      100

#define FRAME_TITLE "Quaternions"

using namespace std;

//Rotational angle for rotation matrix,
//dual-quaternion and hyper-dual-quaternion
static GLfloat rotate_angle = 0.0;
//Translation
static GLfloat translate = 0.0;
//Constant rotational angle for quaternion with SLERP
static GLfloat qrotate_angle = 0.5*M_PI;
//The t for SLERP (between 0 and 1)
static GLfloat t = 0;

//Degrees to radians
const float DEG2RAD = 3.14159/180;
//error
const float epsilon = 0.00001;
```

```
//Auxiliary variable for begin position of quaternion
int beginposition = 0;
//Am I moving up?
bool movingUp = true;

//Print text s on position (x,y)
void printText (float x, float y, char s[])
{
    int  i;
    char c;
    glRasterPos2f(x, y);
    int length = strlen(s);
    for (i=0; i<length; i++)
    {
        c = s[i];
        glutBitmapCharacter (GLUT_BITMAP_8_BY_13, c);
    }
}

//Quaternion class:
class Quaternion
{
    //The four components
    float r, x, y, z;


    public:
    //Constructor
    Quaternion(float a, float b, float c, float d)
    {
        r=a; x=b; y=c; z=d;
    }

    //Constructor
    Quaternion()
    {
        r=0; x=0; y=0; z=0;
    }

    //Getters
    float getR(){return r;}
    float getX(){return x;}
    float getY(){return y;}
    float getZ(){return z;}

    //Multiplication with scalar s
    Quaternion scalar(float s)
```

```
    {
        return Quaternion(r*s, x*s, y*s, z*s);
    }

    //Addition with another quaternion
    Quaternion add(Quaternion q)
    {
        float newr = r+q.getR();
        float newx = x+q.getX();
        float newy = y+q.getY();
        float newz = z+q.getZ();
        return Quaternion(newr, newx, newy, newz);
    }

    //Print this quaternion
    void printQuaternion()
    {
        cout << "(" << r << ", ";
        cout << x << ", " << y << ", " << z << ")";
    }

    //Computes magnitude of this quaternion
    float magnitude()
    {
        return sqrt(pow(r, 2)+pow(x, 2)+pow(y, 2)+pow(z, 2));
    }

    //Computes multiplication with another quaternion
    Quaternion multiply(Quaternion q)
    {
        if(r==0 && q.getR()==0)
        {
            float real = -x*q.getX()-y*q.getY()-z*q.getZ();
            float x2 = y*q.getZ()-z*q.getY();
            float y2 = z*q.getX()-x*q.getZ();
            float z2 = x*q.getY()-y*q.getX();
            return Quaternion(real, x2, y2, z2);
        }
        else
        {
            float real =
                r*q.getR()-x*q.getX()-y*q.getY()-z*q.getZ();
            float x2 =
                x*q.getR()+r*q.getX()+y*q.getZ()-z*q.getY();
            float y2 =
                y*q.getR()+r*q.getY()+z*q.getX()-x*q.getZ();
            float z2 =
                z*q.getR()+r*q.getZ()+x*q.getY()-y*q.getX();
```

```
            return Quaternion(real, x2, y2, z2);
        }
    }

    //Computes complex conjugate
    Quaternion conjugate()
    {
        return Quaternion(r, -x, -y, -z);
    }
};

//Identity quaternion
const Quaternion id = Quaternion(1,0,0,0);
//Zero quaternion
const Quaternion zero = Quaternion(0,0,0,0);

//Dualquaternion class:
class Dualquaternion
{
    //The two components
    Quaternion real, dual;

    public:
    //constructor
    Dualquaternion(Quaternion a, Quaternion b)
    {

        real = Quaternion(a.getR(),a.getX(),a.getY(),a.getZ());
        dual = Quaternion(b.getR(),b.getX(),b.getY(),b.getZ());
    }

    //Getters
    Quaternion getReal(){return real;}
    Quaternion getDual(){return dual;}

    //Print this dual-quaternion
    void printDualquaternion()
    {
        cout << "(";
        real.printQuaternion();
        cout << ",";
        dual.printQuaternion();
        cout << ")" <<endl;
    }

    //Computes addition with another quaternion
    Dualquaternion add(Dualquaternion a)
    {
```

```
        Quaternion newreal = getReal().add(a.getReal());
        Quaternion newdual = getDual().add(a.getDual());
        return Dualquaternion(newreal, newdual);
    }

    //Multiply with another quaternion
    Dualquaternion multiply(Dualquaternion a)
    {
        Quaternion r = getReal().multiply(a.getReal());
        Quaternion d =
                getReal().multiply(a.getDual()).
                        add(getDual().multiply(a.getReal()));
        return Dualquaternion(r, d);
    }

    //Computes the complex conjugate
    Dualquaternion conjugate()
    {
        Quaternion cr = getReal().conjugate();
        Quaternion cd = getDual().conjugate();
        return Dualquaternion(cr, cd.scalar(-1));
    }
};

//Hyper-dual-quaternion class
class Hyperdualquaternion
{
    Quaternion real;
    Quaternion dual1;
    Quaternion dual2;
    Quaternion dual3;

    public:
    //Constructor
    Hyperdualquaternion
    (Quaternion a, Quaternion b, Quaternion c, Quaternion d)
    {
        real = Quaternion(a.getR(),a.getX(),a.getY(),a.getZ());
        dual1 = Quaternion(b.getR(),b.getX(),b.getY(),b.getZ());
        dual2 = Quaternion(c.getR(),c.getX(),c.getY(),c.getZ());
        dual3 = Quaternion(d.getR(),d.getX(),d.getY(),d.getZ());
    }

    //Getter
    Quaternion getReal()
    {
        return real;
    }
```

```
    //Getter
    Quaternion getDual(int x)
    {
        switch(x)
        {
            case 1: return dual1;
            case 2: return dual2;
            default: return dual3;
        }
    }

    //Print this hyper-dual-quaternion
    void printHyperdualquaternion()
    {
        cout << "(";
        real.printQuaternion();
        cout << ", ";
        dual1.printQuaternion();
        cout << ", ";
        dual2.printQuaternion();
        cout << ", ";
        dual3.printQuaternion();
        cout << ")" <<endl;
    }

    //Add with another hyper-dual-quaternion
    Hyperdualquaternion add(Hyperdualquaternion h)
    {
        Quaternion a = real.add(h.getReal());
        Quaternion b = dual1.add(h.getDual(1));
        Quaternion c = dual2.add(h.getDual(2));
        Quaternion d = dual3.add(h.getDual(3));
        return Hyperdualquaternion(a, b, c, d);
    }

    //Multiply with another hyper-dual-quaternion
    Hyperdualquaternion multiply(Hyperdualquaternion h)
    {
        Quaternion a = real.multiply(h.getReal());
        Quaternion b =
        real.multiply(h.getDual(1)).
        add(dual1.multiply(h.getReal()));
        Quaternion c =
                real.multiply(h.getDual(2)).
                        add(dual2.multiply(h.getReal()));
        Quaternion d1 = real.multiply(h.getDual(3));
        Quaternion d2 = dual1.multiply(h.getDual(2));
```

```
        Quaternion d3 = dual2.multiply(h.getDual(1));
        Quaternion d4 = dual3.multiply(h.getReal());
        Quaternion d = d1.add(d2.add(d3.add(d4)));
        return Hyperdualquaternion(a, b, c, d);
    }

    //Computes the complex conjugate
    Hyperdualquaternion conjugate()
    {
        Quaternion a = real.conjugate();
        Quaternion b = dual1.conjugate();
        Quaternion c = dual2.conjugate();
        Quaternion d = dual3.conjugate();
        return Hyperdualquaternion
                (a, b.scalar(-1), c.scalar(-1), d.scalar(-1));
    }
};

//Computes rotation with quaternions
Quaternion quaternionRotation
        (Quaternion original, Quaternion axis, float angle)
{
    axis = axis.scalar(1/axis.magnitude());
    float a = cos(0.5*angle);
    float b = sin(0.5*angle);
    Quaternion rotation =
        Quaternion(a, b*axis.getX(),
                b*axis.getY(), b*axis.getZ());
    Quaternion rotationinv = rotation.conjugate();
    Quaternion x = rotation.multiply(original);
    return x.multiply(rotationinv);
}

//Computes rotation with translation with dual-quaternions
Dualquaternion dqtransform
        (Quaternion original, Quaternion axis,
                float angle, float tx, float ty, float tz)
{
    axis = axis.scalar(1/axis.magnitude());
    float a = cos(0.5*angle);
    float b = sin(0.5*angle);
    Quaternion rotation =
        Quaternion(a, b*axis.getX(),
                b*axis.getY(), b*axis.getZ());
    Quaternion translation =
        Quaternion(0, tx/2.0, ty/2.0, tz/2.0).multiply(rotation);
    Dualquaternion both = Dualquaternion(rotation, translation);
    Dualquaternion bothcon = both.conjugate();
```

```
    Dualquaternion start = Dualquaternion(id, original);
    Dualquaternion result = both.multiply(start);
    return result.multiply(bothcon);
}

//Computes rotation with translation with hyper-dual-quaternions
Hyperdualquaternion hdqtransform
        (Quaternion original, Quaternion axis,
                float angle, float tx, float ty, float tz)
{
    axis = axis.scalar(1/axis.magnitude());
    float a = cos(0.5*angle);
    float b = sin(0.5*angle);
    Quaternion rotation =
        Quaternion(a, b*axis.getX(),
                b*axis.getY(), b*axis.getZ());
    Quaternion translation =
        Quaternion(0, tx/2.0, ty/2.0, tz/2.0).
                multiply(rotation);
    Hyperdualquaternion start =
        Hyperdualquaternion(id, zero, zero, original);
    Hyperdualquaternion both =
        Hyperdualquaternion(rotation, zero, zero, translation);
    Hyperdualquaternion bothcon = both.conjugate();
    Hyperdualquaternion result = both.multiply(start);
    return result.multiply(bothcon);
}

//Computes SLERP, but when begin==end it returns begin
Quaternion SLERP
        (Quaternion begin, Quaternion end, float t, float angle)
{
    if(begin.getX()<=end.getX()+epsilon &&
        begin.getY()<=end.getY()+epsilon &&
        begin.getZ()<=end.getZ()+epsilon &&
        begin.getX()>=end.getX()-epsilon &&
        begin.getY()>=end.getY()-epsilon &&
        begin.getZ()>=end.getZ()-epsilon)
        return begin;
    Quaternion first =
        begin.scalar((sin((1-t)*angle))/sin(angle));
    Quaternion second = end.scalar(sin(t*angle)/sin(angle));
    return first.add(second);
}

//Draws circle with origin on (x,y,z) with radius
void drawCircle(float radius, float x, float y, float z)
{
```

```
    glBegin(GL_LINE_LOOP);
    for (int i=0; i < 360; i++)
    {
        float degInRad = i*DEG2RAD;
        glVertex3f(cos(degInRad)*radius+x,
                   sin(degInRad)*radius+y, 0+z);
    }
    glEnd();
}

//Draws vector from (x1, y1, z1) to (x2, y2, z2)
void drawVector
        (float x1, float y1, float z1,
                float x2, float y2, float z2)
{
    glLineWidth(3);
    glBegin (GL_LINES);
        glVertex3f (x1, y1, z1);
        glVertex3f (x2, y2, z2);
    drawCircle(0.015, x2, y2, z2);
}

//Begin position of different vectors
Quaternion begin = Quaternion(0,0.9,0,0);
Quaternion begin2 = Quaternion(0,0.9,0,0);
Quaternion begin3 = Quaternion(0,0.9,0,0);

//Displays rotating vectors on screen
void display()
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);

    glLoadIdentity();

    glPointSize(6);

    //Get new beginposition when rotation finishes
    switch(beginposition)
    {
        case 0 : begin = Quaternion(0,0.9,0,0); break;
        case 1 : begin = Quaternion(0,0,0.9,0); break;
        case 2 : begin = Quaternion(0,-0.9,0,0); break;
        default: begin = Quaternion(0,0,-0.9,0); break;
    }

   //Upper left
```

```
    glPushMatrix();
    glViewport(0, WINSIZE_Y/2, WINSIZE_X/2, WINSIZE_Y/2);
    printText(-0.9, 0.9, (char*)"Rotation matrix");
        glBegin(GL_POINTS);
                glColor3f(0.5, 0.5, 0.5);
                glVertex3f(0,translate,0);
        glEnd();
        glColor3f(1,0,0);
        glTranslatef(0, translate, 0);
        glRotatef(rotate_angle, 0, 0, 1);
        drawVector(0, 0, 0, 0, 0.9, 0);
    glPopMatrix();

    //Lower left
    glViewport(0,0,WINSIZE_X/2, WINSIZE_Y/2);
    printText(-0.9, 0.9, (char*)"Quaternion with SLERP");
    glPushMatrix();
        glBegin(GL_POINTS);
            glColor3f(0.5, 0.5, 0.5);
            glVertex3f(0,0,0);
        glEnd();
        glColor3f(0,1,0);
        Quaternion end =
                quaternionRotation(begin,
                        Quaternion(0,0,0,1), qrotate_angle);
        Quaternion interpolated =
                SLERP(begin, end, t, qrotate_angle);
        drawVector(0,0,0, interpolated.getX(),
                interpolated.getY(), interpolated.getZ());
    glPopMatrix();

    //Upper right
    glViewport(WINSIZE_X/2, WINSIZE_Y/2,
        WINSIZE_X/2, WINSIZE_Y/2);
    printText(-0.9, 0.9, (char*)"Dual-quaternion w/o SLERP");
    glPushMatrix();
        glBegin(GL_POINTS);
            glColor3f(0.5, 0.5, 0.5);
            glVertex3f(0,translate,0);
        glEnd();
        glColor3f(0,0,1);
        Dualquaternion end2 =
                dqtransform(begin2, Quaternion(0,0,0,1),
                        rotate_angle*DEG2RAD, 0, translate, 0);
        drawVector(0, translate, 0,
                end2.getDual().getX(), end2.getDual().getY(),
                        end2.getDual().getZ());
    glPopMatrix();
```

```
    //Lower right
    glViewport(WINSIZE_X/2, 0, WINSIZE_X/2, WINSIZE_Y/2);
    printText(-1.0, 0.9,
        (char*)"Hyper-dual-quaternion w/o SLERP");
    glPushMatrix();
        glBegin(GL_POINTS);
            glColor3f(0.5, 0.5, 0.5);
            glVertex3f(0,translate,0);
        glEnd();
        glColor3f(1,1,0);
        Hyperdualquaternion end3 =
                hdqtransform(begin3, Quaternion(0,0,0,1),
                        rotate_angle*DEG2RAD, 0, translate, 0);
        drawVector(0,translate,0,end3.getDual(3).getX()
                ,end3.getDual(3).getY(),end3.getDual(3).getZ());
    glPopMatrix();

    glutSwapBuffers();
}

//Animation function
void animate()
{
    rotate_angle+=1.0;
    if (rotate_angle>360) rotate_angle=0;
    t+=0.0175;
    if(t>1)
    {
       t=0;
       beginposition = (beginposition + 1) % 4 ;
       //Start a new rotation
    }
    if(movingUp)
        translate+=0.01;
    else
        translate-=0.01;
    if(translate>1) movingUp = false;
    if(translate<-1) movingUp = true;
    glutPostRedisplay();
}

//Main function
int main(int argc, char* argv[])
{
    glutInit(&argc,argv);

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

```
    glutInitWindowSize (WINSIZE_X, WINSIZE_Y);
    glutInitWindowPosition (WINPOS_X, WINPOS_Y);
    glutCreateWindow (FRAME_TITLE);

    glEnable(GL_DEPTH_TEST);

    glutDisplayFunc(display);
    glutIdleFunc(animate);

    glutMainLoop();

    return 0;
}
```