MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-00000-XXX/00

# SIMULAÇÃO E VISUALIZAÇÃO DE ATITUDE DE SATÉLITES COM PAINÉIS ARTICULADOS

Carlos Henrique Gustavo Hassmann

Dissertação de Mestrado do Curso de Pós-Graduação em Engenharia e Tecnologia Espaciais / Mecânica Espacial e Controle, orientada pelo Dr. Valdemir Carrara aprovada em 11/fevereiro/2008.

INPE
São José dos Campos
2008

**PUBLICADO POR:**

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6911/6923
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-00000-XXX/00

# SIMULAÇÃO E VISUALIZAÇÃO DE ATITUDE DE SATÉLITES COM PAINÉIS ARTICULADOS

Carlos Henrique Gustavo Hassmann

Dissertação de Mestrado do Curso de Pós-Graduação em Engenharia e Tecnologia Espaciais / Mecânica Espacial e Controle, orientada pelo Dr. Valdemir Carrara aprovada em 11/fevereiro/2008.

INPE
São José dos Campos
2008

00.000.00(000.0)

Hassmann. C.H.G..

/ Hassmann. C.H.G.. – São José dos Campos: INPE, 2008.
97p.; (INPE-00000-XXX/00)

1. atitude de satélites/satellite attitude. 2. simulação/simulation. 3. simulação de atitude/attitude simulation. 4. visualização de simulação/simulation visualization. 5. visualização de satélites/satellite visualization. I. Simulação e visualização de atitude de satélites com painéis articulados.

**Simulação e visualização de atitude de satélites com painéis articulados Curso de Pós-Graduação em Engenharia e Tecnologia Espaciais**.

## RESUMO

Este trabalho apresenta a descrição de um conjunto de funções para visualização gráfica e simulação numérica do movimento de atitude de satélites artificiais terrestres que apresentem painéis articulados. O conjunto de funções apresenta a possibilidade da visualização da dinâmica destes satélites. Tem-se, portanto, um ambiente de simulação de atitude de satélites com a respectiva animação gráfica da atitude. A atitude é propagada integrando-se numericamente suas equações e a visualização da dinâmica do corpo é efetuada simultaneamente como resultado desta integração por intermédio da matriz de atitude. Esta matriz de atitude é obtida na forma de quatérnios. Com esta matriz, obtida a cada instante conforme o passo de integração, e uma geometria inicial é possível ter-se uma animação. Para a visualização foi desenvolvido o conjunto de funções para animação gráfica que permite a visualização do movimento de atitude de forma síncrona ao computador. A cada instante, como já foi dito, uma nova matriz de atitude é criada dentro de um laço de integração. Várias atitudes em seqüência proporcionam ilusão de movimento. A visualização é viabilizada com a utilização da biblioteca OpenGL. Essa biblioteca permite não somente a manipulação de elementos gráficos como, também, o interfaceamento com o usuário e a manipulação de matrizes. Sendo o OpenGL também um conjunto de funções é necessária uma linguagem de programação a ele aliada para se fazer as chamadas dessas funções. A linguagem escolhida é o C/C++ , pois há uma grande tradição no uso desta linguagem com as funções OpenGL, além de uma vasta biblioteca de funções que facilitam, por exemplo, as manipulações matemáticas das equações necessárias para o trabalho. As funções foram validadas por intermédio da visualização do fenômeno de nutação para corpos rígidos, pois os resultados para este caso podem ser obtidos analiticamente, logo as funções podem ser testadas. É possível, além disso, ler-se a geometria de satélites por intermédio de arquivos. As coordenadas de um satélite ficam armazenadas em um arquivo que é lido por uma função específica para ser reproduzido na simulação. Tem-se, portanto, uma ferramenta genérica.

# SIMULATION AND VISUALIZATION SATELLITES WITH JOINED PANELS ATTITUDE


## ABSTRACT

This work presents the description of a function set for graphical visualization and numerical simulation of the attitude movement for Earth artificial satellites carrying articulated panels. The function set also presents, also the possibility of dynamic visualization of these satellites. Then exist thus a simulation environment for satellites attitude with the corresponding graphical animation. The attitude propagation will be implemented by the numerical integration of the equations of motion and the respective body visualization will occur in a synchronic time with the computer as a result of this integration by the attitude matrix. This attitude matrix is obtained in terms of quaternions. With this matrix obtained at any instant according to integration step and an initial satellite geometry is possible to do the animation. For the visualization was made a function set that allows to see the attitude movement in synchronic time. At each time step, as said, a new attitude matrix is created into a integration loop. By working on a sequence of matrices obtained for program several integration time step the graphical visualization creates the illusion of the object motion. The visualization is possible by using the OpenGl graphical library. This library allows not only graphical manipulation, but also user interface and matrix manipulation. As any library it needs a programming language for the functions calls. The choose language was C/C++, because exists an old tradition between OpenGl and C/C++ programming language and a large functions library that makes easier, for example, the mathematical manipulation of the equations used in this work. The functions were tested by visualization of the nutation phenomenon for rigid bodies because the results for this case can be obtained in a analytical way allowing the test of the functions. It is possible too read the satellites geometry from files. The body coordinates are stored in a file that is read with a special function for the drawing at screen during the simulations. The simulation toll can be used with many kinds of files.

**SUMÁRIO**

**Pág.**

## LISTA DE FIGURAS

**Pág.**

## LISTA DE ABREVIATURAS E SIGLAS

| | | |
|---|---|---|
| AC3D | – | Tipo de arquivo originado pelo programa "Blender 3D"e AC3D |
| AOC | – | Sistema de controle de atitude e órbita |
| Clock | – | Relógio Interno do Computador |
| CPU | – | Unidade Central de Processamento |
| MECB | – | Missão Espacial Completa Brasileira |
| PMM | – | Plataforma Multi Missão |
| RGB | – | Padrão universal de cores usados pelos sistemas computacionais |
| Sisop | – | Sistema Operacional |
| Time Step | – | Intervalo de Tempo |
| 3DS | – | Tipo de arquivo originado pelo programa "3D Studio" |

# 1 INTRODUÇÃO

A atitude de um satélite é uma medida da forma com que ele orienta-se no espaço. O movimento de um satélite é especificado por sua posição, velocidade, atitude e movimentação de atitude. As duas primeiras quantidades descrevem o movimento de transalção do centro de massa do satélite e as duas últimas descrevem o movimento rotacional do satélite em torno de seu centro de massa (Wertz, 2002). Um dos tópicos da análise de atitude para projetos de satélites é a predição da atitude e tem como objetivo permitir que se analise a orientação futura do satélite (Wertz, 2002). Esta predição é possível por intermédio, entre outras técnicas, da simulação numérica de atitude. Os principais limitadores de uma simulação numérica são o conhecimento a respeito dos torques aplicados e a acurácia do modelo matemático que descreve a dinâmica e os equipamentos do satélite. Segundo Wertz (Wertz, 2002) a atitude é regida pelas equações da dinâmica. Faz-se necessário o equacionamento da dinâmica de atitude para a simulação de um satélite seja este simulado como um corpo rígido ou que tenha apêndices articulados fixados ao seu corpo. Uma simulação desse tipo é de máxima importância, pois é uma forma de prever-se o comportamento do satélite, quando em missão, com relação à atitude. Ao obter-se o modelo matemático (equações do movimento relativas ao satélite em questão) é necessário integrá-las e verificar se os resultados são coerentes com o que se espera em relação a especificação da missão.

Atualmente o Instituto Nacional de Pesquisas Espaciais (INPE) está executando o projeto de uma Plataforma Multi Missão (PMM), que visa construir uma plataforma que seja adaptável a várias missões científicas, e cujo sistema de controle de atitude deverá ser integrado no Brasil. Desta forma se terá sempre pronta uma base de um satélite e não será necessário projetar-se um novo satélite a cada missão que surja no futuro. O controle para um satélite deste tipo ou qualquer outro é regido por um programa de controle (programa computacional) que tem como uma das fases de desenvolvimento a simulação de atitude para poder-se validar todo o sistema de controle. A avaliação do sistema de controle da PMM será feita por intermédio da análise dos resultados numéricos, da análise de gráficos e pode ser feita também pela visualização do movimento de atitude. A visualização da atitude é útil, pois ela permite que o projetista tenha uma idéia clara de como será o comportamento do satélite quando em órbita. Nem sempre é possível ao projetista ter noção de como será o movimento de atitude do satélite somente por intermédio de resultados numéricos e gráficos bidimensionais. Em resumo a visualização do movimento de atitude é uma poderosa ferramenta adicional para a análise de projeto.

O presente trabalho tem como objetivo desenvolver um pacote de visualização gráfica que seja modular, adaptável a várias missões, reconfigurável, isto é, que possa ser modificado

em tempo de execução, com geometria e texturas fornecidas pelo usuário, com portabilidade, e com possibilidade de se construir a geometria a partir de editores gráficos não comerciais. O conjunto de funções deve permitir ainda a visualização da simulação de atitude de satélites artificiais com apêndices articulados. Com relação ao equacionamento da dinâmica do satélite com painéis articulados será utilizada uma biblioteca desenvolvida dentro do próprio INPE (Carrara, Hassmann, 2007). Objetiva-se também implementar funções específicas para lidar com geometrias geradas por programas de edição gráfica e com texturas. Essas acrescentam um realismo maior à simulação, pois o projetista verá o satélite não só no formato, mas também em suas cores e texturas reais. Para isso pretende-se construir funções que permitam que está seja lida de arquivos binários ou do tipo texto e reproduzidos em tela para a análise do projetista. Além disso, estas funções devem suportar formatos gerados por editores gráficos livres ("freeware"), como Blender 3D (Blender, 2008) ou 3D Canvas (Amabilis, 2008), e também editores profissionais de amplo uso como 3D Max (Autodesk, 2008).

Tem-se também como objetivo que a representação gráfica dos movimentos de um corpo seja feita em sincronia com o computador em que está sendo executado a simulação. O conjunto de funções de visualização desenvolvido deverá possuir caráter genérico, isto é, não se prenderá a um dado satélite, mas permitirá a configuração completa da geometria. Uma vez que é bastante usual a utilização de painéis solares giratórios nos satélites atuais, deverão ser construídas funções para permitir o suporte de painéis articulados com movimento relativo ao corpo do satélite.

O satélite, portanto, não será encarado, exatamente, um corpo rígido, mas terá um comportamento dinâmico variável em função de painéis articulados. As relações dinâmicas e cinemáticas do movimento de um corpo no espaço com painéis articulados, levando-se em conta a posição do centro de massa e a variação do momento de inércia do conjunto, necessários para efetuar a simulação do movimento do satélite são levadas em consideração (Carrara e Hassmann, 2007).

Um corpo com painéis articulados consiste em uma estrutura considerada rígida por partes, contendo um corpo principal, unido a um ou mais painéis também considerados rígidos por meio de juntas ou articulações rotativas. Juntas prismáticas ou de deslocamento linear não serão consideradas neste trabalho, embora, conforme Hughes (Hughes, 1986), também tenham uma grande aplicabilidade em problemas que envolvam a dinâmica de atitude de satélites, como os satélites providos de mastro extensível ("boom").

Há vários tipos de simuladores computacionais de visualização, tanto comerciais quanto gratuitos existentes atualmente. É possível perceber, porém, que há exemplos de simu-

ladores que não levam em consideração a correção dinâmica do movimento. Simuladores onde não há preocupação com a dinâmica executam movimentos com dinâmica simplificada.

Simuladores visuais (com a animação do fenômeno físico) dinamicamente corretos talvez sejam difíceis de se encontrar, pois há poucos trabalhos desse tipo relatados. Sabe-se que a Agência Espacial Norte Americana (NASA) (Nasa, 2007 ou Nasa_Simlabs, 2007) utiliza cada vez mais de simuladores para a geração de vídeos usados na divulgação de seus trabalhos, além de simuladores para suas missões. Para o público leigo é muito mais confortável ver uma cena animada do que gráficos e relatórios, o que dá credibilidade ao trabalho. A visualização permite, por exemplo, que erros de projeto ou na dinâmica sejam detectados quando da simulação.

O trabalho aqui relatado tem afinidade com simuladores dinâmicos. Nesta categoria, como simuladores para aeronaves, pode-se citar como por exemplo o "Fligh Simulator"(Microsoft_Games, 2007) que mesmo não sendo um simulador profissional, apresenta bastante correção física e também o "X-Plane"(Laminar_Research, 2007). Na área de simuladores espaciais há o "Space Simulator"e o "Orbiter Simulator"(Schweiger, 2006). O fabricante deste último adverte inclusive para a necessidade de conhecimento sobre mecânica celeste, física e outros tópicos para se poder iniciar a sua utilização.

Em meio às funções de simulação, com base na matriz de inércia do corpo e equações do movimento, calcula-se a matriz de atitude por intermédio de quatérnios. Conforme Coutinho (Coutinho, 2001) o uso de quatérnios permite eliminar a matriz de co-senos para a obtenção da matriz de atitude, eliminando também o problema de singularidades e a limitação de pequenos ângulos. Os quatérnios, por sua vez, serão calculados a intervalos discretos de tempo, por meio de um integrador numérico de equações diferenciais.

Quando da injeção em órbita muitos veículos espaciais assumem uma configuração final diferente da inicial. Durante essa fase há movimentos relativos entre as partes do veículo. Satélites com painéis solares que são abertos após a injeção em órbita, ou que giram para acompanharem o movimento do sol em relação ao satélite, ou ainda que possuam braços robóticos, são exemplos de aplicações do modelo de satélite a ser desenvolvido aqui.

Neste trabalho será suposto que o satélite seja formado por diversas estruturas rígidas. Estas estruturas estarão unidas por juntas articuladas com um corpo principal, ou seja, devido aos seus movimentos em relação ao corpo principal, tem-se um satélite com geometria variável.

A visualização gráfica será realizada com a utilização da biblioteca OpenGL. O conjunto

de funções a ser desenvolvido deverá oferecer uma interface entre uma dinâmica qualquer de atitude aliada a uma geometria fornecida de um satélite, com a biblioteca OpenGL. Isto irá desobrigar o usuário das funções de conhecer profundamente o OpenGL.

Com a matriz de atitude do dispositivo (corpo rígido) pode-se visualizar o comportamento deste via um conjunto de funções de visualização gráfica, tendo como tarefa transcrever as coordenadas do satélite no espaço via matriz de atitude conforme intervalos de tempo determinados previamente (time steps) para coordenadas de tela onde será possível a visualização do movimento de forma síncrona ao "clock"do computador.

A linguagem escolhida para o desenvolvimento do simulador e da visualização será o C/C++, pois há uma grande tradição no uso desta linguagem com as funções OpenGL, além de uma vasta biblioteca de funções matemáticas. Considera-se ainda uma característica importante da linguagem C que é uma grande velocidade de execução, sendo ainda uma linguagem, basicamente, independente de arquitetura computacional, logo muito versátil (Kernighan e Ritchie, 1988).

Deve-se lembrar ainda que a motivação para o desenvolvimento desse conjunto de funções é a mesma da biblioteca de funções desenvolvidas para o simulador de atitude de apoio a missões espaciais para suporte à plataforma multi missão (PMM) da MECB (Carrara e Hassmann, 2007). Estas funções fazem parte do conjunto de programas destinados à qualificação do sistema de controle de atitude e órbita (AOC) do satélite.

O trabalho está organizado da seguinte forma: no segundo capítulo deste trabalho serão abordados os fundamentos teóricos como as notações que serão utilizadas, a álgebra de quatérnios, as equações do movimento e a integração numérica dessas equações. Ainda no segundo capítulo será vista a parte de visualização em tempo real e a descrição dos algorítmos utilizados para que a simulação tenha uma visualização síncrona ao processador do computador. Esse capítulo também é reservado para a explanação de uma interface computacional e comparações entre os métodos de sincronização e o terceiro capítulo apresenta os resultados obtidos, sendo o quarto a conclusão.

# 2 FUNDAMENTOS TEÓRICOS

O estado de um satélite é representado pela sua órbita e atitude, ou seja a posição e velocidades lineares e angulares em relação a um sistema de referência conhecido. Neste trabalho, porém, se fará referência somente à atitude. Bastam três parâmetros para a descrever a atitude e sua variação que são as três velocidades angulares (uma para cada eixo coordenado) ou os três ângulos de Euler, por exemplo, a cada instante. A posição pode ser representada de diversas formas: ângulos de Euler, ângulo e vetor de Euler, vetor de Gibbs, quatérnios e matriz de atitude (Wertz 2002).

A velocidade angular pode também assumir formas equivalentes, mas o mais comum é representá-la por meio de três componentes num sistema cartesiano fixado ao corpo do satélite. Neste trabalho serão utilizados quatérnios e velocidades angulares cartesianas.

Neste capítulo abordar-se-á, particularmente, a teoria geral de quatérnios, a aplicação desses às equações da dinâmica, a forma de integração das equações e as estratégias de visualização.

## 2.1 Notações

Neste item apresentar-se-á a notação referente a vetores e matrizes que será usada neste trabalho. Serão também apresentadas as notações relativas às operações matemáticas entre esses elementos.

A notação utilizada na representação da cinemática e dinâmica de sólidos é fundamental quando se deseja manter o formalismo matemático, conforme Hughes assinalou (Hughes, 1986). Vetores são representados em bases de coordenadas, e transformações entre bases são realizadas por matrizes cujos elementos são vetores da base (ou suas componentes). Percebe-se, com isso, que uma notação baseada exclusivamente em matrizes, onde um vetor passa a ser uma matriz coluna, é mais conveniente e de fácil compreensão. Isto facilita também, como demonstrado por Hughes, toda a álgebra de vetores. Torna-se então fundamental estabelecer uma representação matricial das diversas operações algébricas entre vetores. Obviamente a soma e a subtração são triviais. Mais importantes, porém, serão os produtos escalar e vetorial.

Um produto vetorial entre $\vec{v}$ e $\vec{u}$ é denotado por $\vec{v} \times \vec{u}$. Seu equivalente, no entanto, em notação matricial é dado por

$$\vec{v} \times \vec{u} = \Omega(v)u \tag{2.1}$$

onde $\Omega(\cdot)$ representa o equivalente matricial do produto vetorial conforme segue

$$\Omega(v) = \begin{bmatrix} 0 & -v_z & -v_y \\ v_z & 0 & -v_x \\ -v_y & -v_x & 0 \end{bmatrix} \tag{2.2}$$

e $v_x$, $v_y$, $v_z$ são as componentes do vetor $v$. A matriz de produto vetorial é uma função ímpar (como o próprio produto vetorial), já que $\Omega(-\omega) = -\Omega(\omega)$. As vantagens de utilizá-la decorrem principalmente da facilidade de operações computacionais com matrizes, facilidade de notação e possibilidade de utilização das propriedades de matrizes. Tanto quanto o produto vetorial, a mudança da ordem das propriedades da matriz $\Omega(\cdot)$ inverte o sinal do resultado (Carrara, 1997)

$$\Omega(\omega)v = -\Omega(v)\omega \tag{2.3}$$

Por sua vez o produto escalar entre os vetores $\vec{v}$ e $\vec{u}$, denotado por $\vec{v}\cdot\vec{u}$ tem o seu equivalente matricial dado por

$$\vec{v} \cdot \vec{u} = v^T u \tag{2.4}$$

onde o sobrescrito $T$ indica a transposição do vetor ou matriz.

## 2.2 Quatérnios

Quatérnios são estruturas matemáticas da Geometria Analítica. São muito usados em representações das orientações e rotações de objetos no espaço. Essas estruturas usam uma notação em quatro dimensões para representar matrizes de rotação $3 \times 3$, pois há mais robustez numérica com relação à propagação de erros o que não ocorre quando da combinação de matrizes de rotação.

O espaço de quatro dimensões (4 D) dos quatérnios é composto por um eixo real e três eixos ortogonais $i$, $j$, $k$, chamados de "imaginários principais". Pode-se manipular o eixo dos imaginários principais como números complexos, logo:

$$i^2 = j^2 = k^2 = -1 \tag{2.5}$$

A representação ainda pode ser dada por

$$q \doteq s + xi + yj + zk = s + v \tag{2.6}$$

ou seja, pela soma de um escalar "$s$"com um vetor "$v$". Esta notação não faz parte da álgebra de vetores convencionalconvencional (pois não há propriedade que permite somar-se um escalar a um vetor). A álgebra de quatérnios é uma extensão da álgebra convencional

introduzida por Hamilton (Wertz 2002).

Na equação anterior $s$ é a parte real e $v$ é um vetor cujas componentes $v_x$, $v_y$ e $v_z$, são números reais e pertencem a $\mathbb{R}^3$, e $v$ é considerado parte puramente imaginária dada por

$$q = \begin{pmatrix} s \\ v_x \\ v_y \\ v_z \end{pmatrix} \tag{2.7}$$

Sendo os quatérnios semelhantes a números complexos lhes cabem, também, as mesmas operações, manipulações e combinações algébricas destes (ver Apêndice A).

Dada a equivalência entre a notação vetorial e matricial, passar-se-á a utilizar, deste ponto em diante, exclusivamente a forma matricial.

### 2.2.1 Quatérnios Unitários

Uma outra forma de representação dos quatérnios é a forma $q = s + au$, onde a parte imaginária $u$ é um vetor unitário; logo pode-se calcular $a$ e $u$ de $v = au$ como segue:

$$a = |v| \tag{2.8}$$

$$u = \frac{v}{a} = \frac{1}{a} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{2.9}$$

Todo quatérnio com parte real nula é chamado de quatérnio puro. Qualquer vetor em $\mathbb{R}^3$ pode ser representado como um quatérnio puro, bastando omitir-se a parte real. Um quatérnio que tenha módulo (ver Apêndice A) igual a unidade é chamado de "quatérnio unitário". Um quatérnio unitário puro é aquele que tem a parte real nula e o valor de seu módulo igual a 1. Qualquer vetor unitário em $\mathbb{R}^3$ (vetores normalizados por ex.) pode ser representado por um quatérnio unitário puro. O conjunto de todos os quatérnios unitários forma uma hiper esfera de raio unitário dentro de um espaço de quatro dimensões (Coutinho, 2001).

### 2.2.2 Representação de Matrizes de Rotação utilizando Quatérnios Unitários

Quatérnios unitários podem ser utilizados como uma representação equivalente às matrizes de rotação no espaço. Ver-se-á agora as transformações necessárias para se passar de quatérnios unitários a matrizes de rotação e de matrizes de rotação para quatérnios. A

matriz de rotação, representando a rotação de um ângulo $\theta$ em torno do vetor unitário $u$ pode ser dada pela matriz de Euler (Wertz, 2002).

$$R = \begin{bmatrix} tu_x^2 + cos\theta & tu_xu_y + u_zsen\theta & tu_xu_z - u_ysen\theta \\ tu_xu_y - u_zsen\theta & tu_y^2 + cos\theta & tu_yu_z + u_xsen\theta \\ tu_xu_z + u_ysen\theta & tu_yu_z - u_xsen\theta & tu_z^2 + cos\theta \end{bmatrix} \tag{2.10}$$

sendo

$$u = (u_x, u_y, u_z)^T \quad \text{e} \quad t = (1 - cos\theta) \tag{2.11}$$

Essa rotação pode ser representada, também, por um quatérnio unitário da seguinte forma:

$$q = cos\left(\frac{\theta}{2}\right) + sen\left(\frac{\theta}{2}\right) u \tag{2.12}$$

Da mesma forma, tendo um quatérnio unitário $q = s+v$, o eixo de rotação $u$ e o ângulo de rotação $\theta$ representados pelo quatérnio podem ser determinados pelas seguintes equações

$$\begin{aligned} cos(\theta) &= 2s^2 - 1 \\ sen(\theta) &= 2s\sqrt{1 - s^2} \\ u &= \frac{v}{\sqrt{1 - s^2}} \end{aligned} \tag{2.13}$$

Se for utilizada uma representação por meio de matriz de rotação, então $u$ e $\theta$ podem ser determinados com as equações que seguem, mas sempre com a restrição de $sen\theta \neq 0$.

$$\begin{aligned} cos(\theta) &= \frac{R_{xx} + R_{yy} + R_{zz} - 1}{2} \\ u_x &= \frac{R_{yz} - R_{zy}}{2sen(\theta)} \\ u_y &= \frac{R_{zx} - R_{xz}}{2sen(\theta)} \\ u_z &= \frac{R_{xy} - R_{yx}}{2sen(\theta)} \end{aligned} \tag{2.14}$$

onde

$$R = \begin{bmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{bmatrix} \tag{2.15}$$

Finalizando pode-se, ainda, calcular a matriz de rotação $3 \times 3$ equivalente a um dado quatérnio $q = s + \vec{v}$, pela substituição das equações anteriores na matriz $R$, logo:

$$R = 2 \begin{bmatrix} s^2 + v_x^2 - \frac{1}{2} & v_x v_y - s v_z & v_x v_z + s v_y \\ v_x v_y + s v_z & s^2 + v_y^2 - \frac{1}{2} & v_y v_z - s v_x \\ v_x v_z - s v_y & v_y v_z - s v_x & s^2 + v_z^2 - \frac{1}{2} \end{bmatrix} \tag{2.16}$$

Com as equações anteriores pode-se passar de quatérnios a matrizes de rotação ou vice-versa, levando-se em consideração um eixo de rotação unitário e um ângulo de rotação específico.

Se for usada a representação matricial, a rotação de um vetor qualquer $p \in \mathbb{R}^3$ é obtida pela simples aplicação da equação $p_r = Rp$. Neste caso $R$ é a matriz de rotação e $p_r$ o vetor rotacionado.

Se, no entanto, for usada a representação por quatérnios, o vetor $p \in \mathbb{R}^3$ pode ser representado pelo quatérnio puro $q_p = 0 + p$ e a rotação pode ser calculada via uma multiplicação de quatérnios (ver Apêndice A para multiplicação, produto vetorial e escalar) (Coutinho, 2001) dada por

$$q_{pr} = q q_p q = 0 + \left[ \left( s^2 - u^T u \right) p + 2(p^T u)u + 2s(\Omega(u)p) \right] = 0 + p_r \tag{2.17}$$

Neste caso $q_{pr}$ é um quatérnio puro, representando o vetor rotacionado $p_r$, e $q$ é o quatérnio associado à matriz de rotação $R$.

Pode-se ter os mesmos resultados, porém em um formato um pouco diferente, mas computacionalmente mais fácil de manipular, usando os Parâmetros Simétricos de Euler (Wertz, 2002) dados por $q_1$, $q_2$, $q_3$ e $q_4$, obtidos de

$$\begin{aligned} q_1 &= u_x sen\left(\frac{\theta}{2}\right) = v_x \\ q_2 &= u_y sen\left(\frac{\theta}{2}\right) = v_y \\ q_3 &= u_z sen\left(\frac{\theta}{2}\right) = v_z \\ q_4 &= cos\left(\frac{\theta}{2}\right) = s \end{aligned} \tag{2.18}$$

onde $u = u_x i + u_y j + u_z k$, tal que

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1 \tag{2.19}$$

A matriz de atitude $R$, conseqüentemente, fica da seguinte forma:

$$R = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{2.20}$$

A inversa desta matriz é obtida pela sua transposta, uma vez que em matrizes ortogonais próprias é válida a igualdade (ver Apêndice A para multiplicação, produto vetorial e escalar) (Coutinho, 2001)

$$R^{-1} = R^T \tag{2.21}$$

### 2.2.3 Vantagens da Utilização de Quatérnios

Existem muitas vantagens na utilização de quatérnios ao invés de matrizes de rotação $3 \times 3$. Uma vantagem imediata é que os quatérnios codificam rotações em apenas quatro números reais, enquanto que uma matriz $3 \times 3$ necessita de nove números para o mesmo fim.

Além da menor quantidade de dados que devem ser manipulados a principal vantagem está relacionada à propagação de erros. Ao multiplicarem-se matrizes de rotação umas com as outras ocorre uma propagação de erros, causada pelo produto de senos e co-senos, gerando uma matriz com o determinante diferente da unidade. Isso é incoerente, pois toda a matriz de rotação é ortogonal, logo tem o determinante igual à unidade. Deve-se então normalizar a matriz, mas isso exige um certo esforço computacional. A mesma coisa pode acontecer com quatérnios, quando esses começam a ser multiplicados entre si, apresentando um módulo diferente de um. A normalização também se faz necessária, mas neste caso trabalha-se com somente quatro números reais, facilitando o trabalho algorítmico e computacional.

Normalizar um quatérnio, porém, não significa, absolutamente, corrigir os erros acumulados. Seja, por exemplo, o quatérnio $'r'$ tal que $|r| \neq 1$. A normalização é feita por

$$q = \frac{r}{|r|} = \frac{1}{\sqrt{r_1^2 + r_2^2 + r_3^2 + r_4^2}}(r_1, r_2, r_3, r_4) \tag{2.22}$$

O erro, no entanto, poder estar acumulado no ângulo de rotação ($q_4$), ou na direção do eixo $(q_1, q_2, q_3)$. Sendo assim nem sempre se pode afirmar que a normalização elimina os erros.

Existem, porém, ainda outras vantagens. Com o uso de quatérnios, por exemplo, são eliminadas as funções trigonométricas utilizadas, normalmente, nas matrizes de rotação.

São eliminadas, principalmente, as singularidades na propagação das equações cinemáticas, enquanto que estas podem ocorrer na representação em ângulos de Euler. O uso de quatérnios permite, ainda, que sejam manipulados tanto pequenos quanto grandes ângulos envolvidos na determinação de atitude de um corpo rígido (Coutinho, 2001).

Com relação à computação gráfica os quatérnios proporcionam realismo a rotações de objetos. Utilizando-se incrementos no quatérnio e, transformando este em matriz de rotação, é possível simular rotações tri dimensionais de objetos (Kuipers, 1999). No caso específico deste trabalho o incremento do quatérnio é dada pelas velocidades angulares.

## 2.3 Equações dinâmicas do movimento

O momento angular $L$ é fundamental na mecânica rotacional (Wertz, 2002). Considerar-se-á inicialmente um sistema de partículas ($n$ pontos de massa concentrados), tendo o momento angular, portanto, dado pela expressão

$$L_{total} = \sum_{i=1}^{n} \Omega(r_i) m_i v_i \tag{2.23}$$

onde $m_i$, $r_i$ e $v_i$ são respectivamente a massa, posição e o vetor velocidade do "$i$ésimo"ponto de massa (Wertz, 2002).

As leis de Newton para o movimento, válidas somente para sistemas de coordenadas inerciais, serão utilizadas para derivar uma equação do movimento para $L$. É importante, portanto, assumir-se que $r_i$ e $v_i$ são relacionados a um sistema de referências inercial (Wertz, 2002). Por conveniência escrever-se-á $r_i$ como uma soma de termos que pode ser visualizada na expressão (2.24) e na Figura 2.1

$$r_i = R + \rho_i \tag{2.24}$$

onde $R$ é a posição do ponto de referência **O'** (no corpo rígido) e $\rho_i$ é a posição do ponto de massa relativo a **O'**.

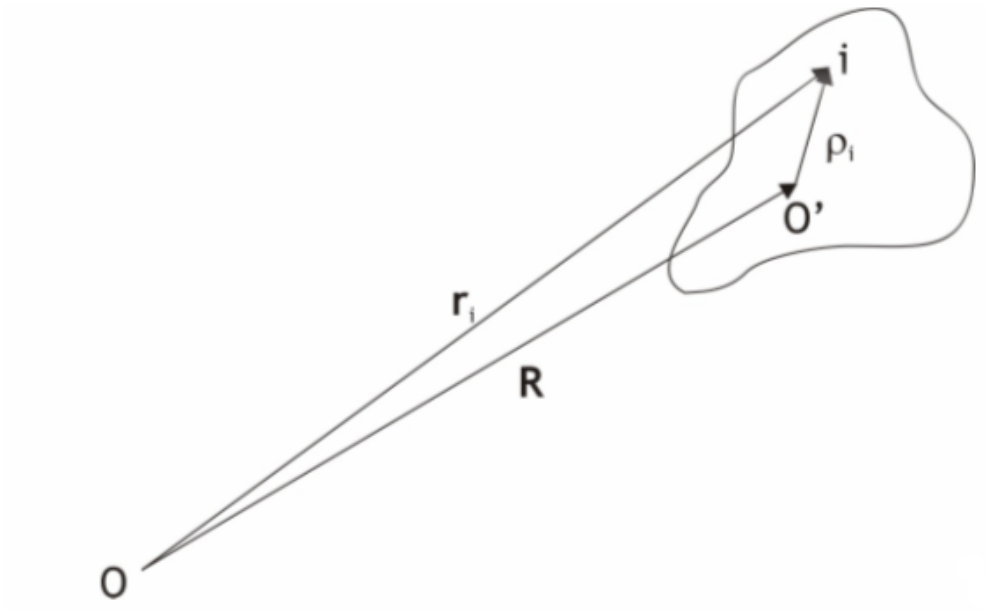


Figura 2.1 - Corpo rígido com referencial inercial **O**.

Derivando a equação (2.24) em relação ao tem-se a expressão

$$v_i = v + \frac{d\rho_i}{dt} \tag{2.25}$$

onde $v$ é a velocidade de **O'** em relação ao referencial inercial. Levando-se em consideração que o corpo rígido possui uma massa M e que se deve efetuar uma integral sobre toda essa massa para obter-se o momento angular total e, substituindo as equações (2.24) e (2.25) em (2.23) ter-se-á:

$$L_{total} = M\Omega(r)v + \Omega(r)\frac{d}{dt}\left[\int_M m\rho\ dm\right] + \Omega\left(\int_M m\rho\ dm\right)v + \Omega\left(\int_M m\rho\ dm\right)\frac{d\rho}{dt} \tag{2.26}$$

Se **O'** for identificado como o centro de massa (CM) ter-se-á

$$\int_M m\rho\ dm = 0 \tag{2.27}$$

conseqüentemente,

$$L_{total} = M\Omega(r)v + L \tag{2.28}$$

onde $m\Omega(r)v$ representa o momento angular de toda a massa, considerando como um ponto localizado no CM e o termo $L$ é a contribuição do movimento de todos os pontos

de massa relativos ao CM expresso como

$$L = \Omega \left( \int_M m\rho \; dm \right) \frac{d\rho}{dt} \tag{2.29}$$

Hughes (Hughes, 1986) se utiliza de uma notação semelhante. Nesta $L$ é o momento angular relativoe $L_{total}$ é o momento angular absoluto.

Assumindo-se o corpo como rígido é possível afirmar que os vetores $\rho_i$ são todos constantes em relação ao sistema de referência fixado no satélite. Essa consideração permite uma manipulação mais conveniente dos vetores. O problema da dinâmica de atitude, porém, preocupa-se somente como o movimento relativo ao centro de massa ($v = 0$), conseqüentemente, com $L$ conforme a equação (2.29) (Wertz, 2002).

Mesmo que as componentes de $\rho_i$ em relação os sistema de coordenadas do satélite sejam constantes, as suas derivadas em relação ao tempo são diferentes de zero se o satélite estiver girando com uma velocidade angular instantânea $\omega$, pois o vetor $\frac{d\rho_i}{dt}$ é a taxa de variação de $\rho_i$ relativa às coordenadas inerciais ao longo dos eixos coordenados fixos no corpo do satélite. Todas as derivadas temporais devem ser referentes a um sistema de referências inercial para que as leis de Newton possam ser aplicadas diretamente. Sabe-se, da cinemática que um vetor fixado a um sistema girante com velocidade $\omega$ possui velocidade dada por

$$\frac{d\rho_i}{dt} = \Omega(\omega)\rho_i \tag{2.30}$$

e, substituindo-se esta equação em (2.29) ter-se-á

$$L = \Omega \left( \int_M m\rho \; dm \right) (\Omega(\omega)\rho_i) = \int_M m \left[ \rho^2 \omega - \left( \rho^T \omega \right) \rho \right] \; dm \tag{2.31}$$

Definindo a matriz de inércia $I$ como

$$\begin{aligned}
I_{11} &= \int_M m \left( \rho_2^2 + \rho_3^2 \right) dm \\
I_{22} &= \int_M m \left( \rho_3^2 + \rho_1^2 \right) \\
I_{33} &= \int_M m \left( \rho_1^2 + \rho_2^2 \right) \\
I_{12} = I_{21} &= - \int_M m\rho_1\rho_2 \\
I_{23} = I_{32} &= - \int_M m\rho_2\rho_3 \\
I_{31} = I_{13} &= - \int_M m\rho_3\rho_1
\end{aligned} \tag{2.32}$$

então a equação (2.31) escrita na forma matricial resume-se a

$$L = I\omega \tag{2.33}$$

onde

$$I = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{12} & I_{22} & I_{23} \\ I_{13} & I_{23} & I_{33} \end{bmatrix} \tag{2.34}$$

Pode-se demonstrar que se o sistema de eixos fixado ao corpo coincidir com o sistema de eixos principal de inércia, então a matriz de inércia fica diagonal e na forma (Wertz, 2002)

$$I = \begin{bmatrix} I_{11} & 0 & 0 \\ 0 & I_{22} & 0 \\ 0 & 0 & I_{33} \end{bmatrix} \tag{2.35}$$

A equação básica da dinâmica de atitude relaciona a derivada temporal do momento angular $\frac{dL}{dt}$ com o torque aplicado $N$. Essa relação, porém, leva em consideração um sistema inercial de coordenadas. Contudo, nas derivadas temporais das componentes do momento angular em relação ao sistema de eixos do satélite posicionado no centro de massa, considerado mais conveniente, surge um termo adicional oriundo da mudança na direção do momento angular no sistema girante, dado por $\Omega(\omega)L$ (Wertz, 2002). Este termo tem origem na derivada temporal de um vetor qualquer. Ao derivar-se um vetor ao longo do eixo de coordenadas de um sistema que está posicionado no centro de massa do corpo rígido e relacionando-se as equações (2.30), (2.33) e (2.31) ter-se-á

$$\dot{L} = N - \Omega(\omega)L = I\dot{\omega} \tag{2.36}$$

Tem-se, portanto, dois sistemas de referência que são o inercial e o sistema fixo no corpo do satélite. Ambos sistemas se relacionam através da matriz de rotação que será vista mais adiante (Fonseca, Lourenção e Freitas, 1985).

Se o sistema de referência fixado ao corpo coincidir com o sistema de eixos principais de inércia, a matriz de inércia torna-se diagonal e as equações apresentadas, expressas em termos de suas componentes, ficam

$$\begin{aligned} I_x\dot{\omega}_x &= N_x + (I_y - I_z)\,\omega_y\omega_z \\ I_y\dot{\omega}_y &= N_y + (I_z - I_x)\,\omega_z\omega_x \\ I_z\dot{\omega}_z &= N_z + (I_x - I_y)\,\omega_x\omega_y \end{aligned} \tag{2.37}$$

ou

$$\begin{aligned}
\dot{L}_x &= N_x + \left(\frac{1}{I_y} - \frac{1}{I_z}\right) L_y L_z \\
\dot{L}_y &= N_y + \left(\frac{1}{I_z} - \frac{1}{I_x}\right) L_z L_x \\
\dot{L}_z &= N_z + \left(\frac{1}{I_x} - \frac{1}{I_y}\right) L_x L_y
\end{aligned} \tag{2.38}$$

onde $I_x$, $I_y$, $I_z$ são os momentos de inércia principais do corpo e $N_x$, $N_y$, $N_z$ os componentes do torque resultante.

Se o torque for nulo, tem-se a dinâmica regida pelo seguinte conjunto de equações (Wertz, 2002):

$$\dot{\omega} = I^{-1} \left[N - \Omega(\omega)(I\omega)\right] \tag{2.39}$$

Escrevendo em termos das componentes de $\dot{\omega}$

$$\begin{aligned}
\dot{\omega}_x &= -\frac{I_z - I_y}{I_x} \omega_y \omega_z \\
\dot{\omega}_y &= -\frac{I_x - I_z}{I_y} \omega_z \omega_x \\
\dot{\omega}_z &= -\frac{I_y - I_x}{I_z} \omega_x \omega_y
\end{aligned} \tag{2.40}$$

As equações apresentadas até aqui representam o movimento de rotação de um corpo rígido. Integrando-se as equações das velocidades angulares, pode-se aplicar os valores obtidos diretamente nas equações cinemáticas expressas em quatérnios (Wertz, 2002)

$$\begin{aligned}
\dot{q}_1 &= \frac{1}{2} \left(\omega_z q_2 - \omega_y q_3 + \omega_x q_4\right) \\
\dot{q}_2 &= \frac{1}{2} \left(-\omega_z q_1 - \omega_x q_3 + \omega_y q_4\right) \\
\dot{q}_3 &= \frac{1}{2} \left(\omega_y q_1 - \omega_x q_2 + \omega_y q_4\right) \\
\dot{q}_4 &= \frac{1}{2} \left(-\omega_x q_1 - \omega_y q_2 + \omega_z q_3\right)
\end{aligned} \tag{2.41}$$

Essas equações, por sua vez, também são integráveis numericamente e os valores resultantes aplicados na matriz de atitude que está na forma de parâmetros simétricos de Euler. Essas equações não chegam a ter um sentido físico, contudo, a facilidade está na brevidade com que se obtém a matriz de atitude.

## 2.4 Modelo Dinâmico e Cinemático de um Satélite com Apêndices Articulados

O equacionamento deste item difere-se do equacinamento do item anterior, pois aqui é levado em consideração os torques e forças atuando sobre o sistema (satélite) . Leva-se em consideração principalmente os torques e forças gerados pela movimentação de apêndices articulados ligados ao corpo principal do satélite que era (item anterior) considerado unicamente um corpo rígido. A estratégia neste momento é abordar cada apêndice (painel solar por ex.) como um corpo rígido.

### 2.4.1 Equações de um corpo articulado

Um satélite composto por painéis rotativos não é absolutamente rígido, pois possui apêndices articulados, logo as equações anteriores não são mais completamente válidas. Será feita, portanto, a suposição mencionada no início do item, ou seja, que o satélite é formado por diversas estruturas rígidas denominadas de elos e unidas por juntas articuladas (em nosso caso juntas rotativas) com um corpo principal (um corpo com geometria variável) conforme a Figura 2.2. Normalmente, a velocidade das articulações é conhecida pois existe um modelo de atuador que fornece a posição $\theta(t)$, sua velocidade $\dot{\theta}(t)$ ou mesmo o torque na junta. É comum a utilização da notação de Denavit-Hartenberg para a solução de casos como este. Esta notação estabelece regras para se compor os diversos sistemas de coordenadas das juntas. Por esta notação, cada junta recebe um sistema de coordenadas retangulares, cuja orientação dos eixos é realizada de forma a facilitar a transposição das diversas matrizes de rotação que compõem o movimento dos braços. Infelizmente, embora esta sistemática seja eficiente para compor movimentos, ela se revela ineficaz para a composição da dinâmica, por não levar em conta a posição do centro de massa de cada elo. Outra desvantagem desta formulação se deve ao fato de que a base de um braço mecânico absorve a reação aos torques aplicados nas juntas, o que não ocorre num corpo livre no espaço. No satélite torna-se imperativo, portanto, que os sistemas de coordenadas intermediários sejam fixados no centro de massa de cada um dos corpos articulados, assegurando desta forma que possam ser computados o centro de massa e matriz de inércia do conjunto. Será suposto também que em cada junta unem-se exclusivamente dois corpos (ou elos), sendo um deles chamado elo antecessor da junta e o outro o elo sucessor da junta. No caso estudado aqui, o elo antecessor será sempre o corpo principal do satélite (Carrara, 1997).

O modelo matemático desenvolvido considera o satelite como um sistema composto por vários corpos rígidos unidos por articulações. O movimento desses apêndices pode ser equacionado em função do tempo, pois seu comportamento ao longo do tempo é conhecido

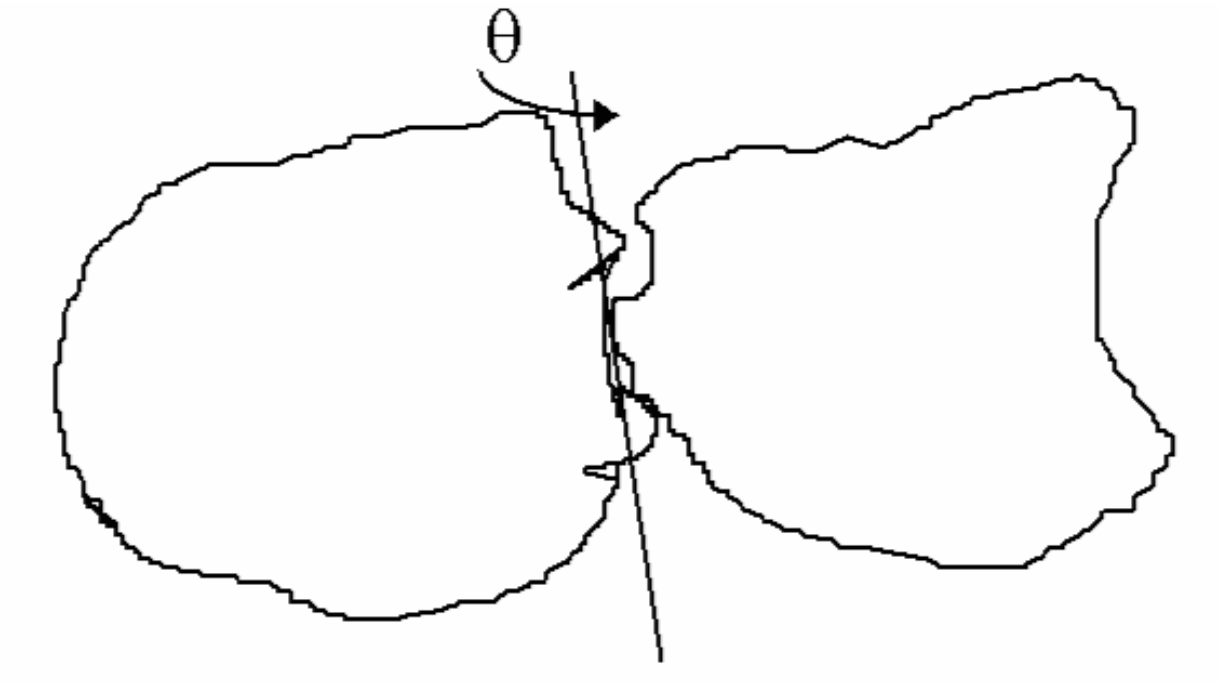


Figura 2.2 - Satélite com articulações.

e isso permite que a integração numérca da atitude seja feita em termos de um único corpo ao invés de vários. A equação diferencial vetorial do movimento de atitude é dada por (Carrara 1997).

$$\dot{\omega}_o^o = (I_o + J_n)^{-1} \left[ \sum (N_{cont} + N_{per}) - \Omega\left(\omega_o^o\right)\left(I_o\omega_o^o + h\right) - \dot{h} - \dot{H}_n \right] \tag{2.42}$$

$$\dot{h} = T \tag{2.43}$$

Onde $I_o$ é a matriz de inércia do corpo do satélite, $N_{cont}$ são os torques de controle, $N_{pert}$ são os torques devidos as perturbações, $h$ é o momento angular armazenado nas rodas de reação, $T$ são os torques internos aplicados nas rodas, $\omega_o^o$ é a velocidade angular do satélite e $\Omega(\cdot)$ indica o operador qua faz o produto vetorial do argumento (Wertz, 2002; Carrara 1997 e Carrara e Hassmann, 2007). $J_n$ e $H_n$ são dados por

$$J_n = \sum_{k=1}^{n} \left( A_{k,o} I_k A_{k,o}^T - m_k \Omega\left(a_{ok}^o - a_{ko}^o\right) \Omega\left(a_{ok}^o - a_{ko}^o\right) \right) + \frac{1}{m_t} \left( \sum_{k=1}^{n} \left( m_k \Omega\left(a_{ok}^o - a_{ko}^o\right) \right) \right)^2 \tag{2.44}$$

$$\dot{H}_n = \sum_{k=1}^{n} \left[ \Omega\left(\omega_o^o + \omega_k^o\right) A_{k,o} I_k A_{k,o}^T \left(\omega_o^o + \omega_k^o\right) + \right. \tag{2.45}$$

$$\left. + A_{k,o} I_k A_{k,o}^T \left(\Omega\left(\omega_o^o\right)\omega_k^o + \dot{\omega}_k^o\right) \right] + \sum_{k=1}^{n} m_k \Omega\left(a_{ok}^o - a_{ko}^o\right)\beta_k - m_t \Omega\left(r_{cm}^o\right) \sum_{k=1}^{n} \mu_k \beta_k$$

e $\beta_k$, $\mu_k$ vêm de

$$\beta_k = \Omega\left(\omega_o^o\right)\Omega\left(\omega_o^o\right) a_{ok}^o - \Omega\left(\omega_o^o + \omega_k^o\right)\Omega\left(\omega_o^o + \omega_k^o\right) a_{ok}^o + \Omega\left(a_{ok}^o\right)\left(\Omega\left(\omega_o^o\right)\omega_k^o + \dot{\omega}_k^o\right) \tag{2.46}$$

$$\mu_k = \frac{m_k}{m_o + \sum_{j=1}^{n} m_j} = \frac{m_k}{m_t} \tag{2.47}$$

e fazendo-se uma analogia às equações da dinâmica para corpos rígidos, devem-se expressar todos os vetores no sistema do corpo do satélite, sistema este no qual as medidas dos diversos sensores de atitude são referenciadas, e no qual é conhecida a matriz de inércia. Como a origem do sistema de direções inerciais é fixada ao centro de massa $r_{cm}$ do conjunto, necessita-se da posição deste centro com relação ao sistema o a cada instante (Carrara, 1997), logo

$$r_{cm}^o = \sum_{k=1}^{n} \mu_k \left(a_{ok}^o - a_{ko}^o\right) \tag{2.48}$$

Nessas equações $I_k$ é a matriz de inércia do apêndice $k$, $m_k$ sua massa, $A_{k,o}$ é a matriz de rotação entre esse apêndice e o corpo do satélite, e $\omega_k(t)$ é a velocidade angular da junta articulada. O sobre escrito nas variáveis indica em que sistema de coordenadas a variável é espressa, ou seja, $o$ para o corpo do satellite e $k$ para o apêndice. Cada apêndice $k$ apresenta dez parâmetros escalares ( $\theta_0$, $d_0$, $a_0$, $t_0$, $\theta_1$, $d_1$, $a_1$, $t_1$, $\theta_2$ e $d_2$) oriundos da notação de Denavit-Hartenberg (Asada e Slotine, 1986; Craig, 1989). Esses parâmetros são usados na cinemática de braços robóticos conforme a Figura 2.3. A vantagem dessa representação é ter-se um grupo específico de regras estabelecidas para a utilização dos parâmetros de Denavit-Hartenberg, que permitem definir cada parâmetro baseado somente na geometria. Se $R_i(\theta)$ representa a matriz de rotação em torno do eixo $i$ com um ângulo $\theta$, então a matriz resultante $A_{k,o}$ é (Carrara e Hassmann, 2007)

$$A_{k,o} = R_z(\theta_0) R_x(t_0) R_z(\theta_k(t)) R_z(t_1) R_x(\theta_1) R_z(\theta_2) \tag{2.49}$$

onde $\theta_k(t)$ é o angulo de rotação do apêndice $k$. Os vetores $a_{ok}$ e $a_{ko}$ representam a posição original do sistema de coordenadas $j$ fixo no eixo da junta no corpo do satélite e apêndice

$k$ respectivamente como pode ser visto na Figura 2.3. Quando dados nas coordenadas do corpo do satellite esses vetores ficam na seguinte forma

$$a^o_{ok} = R_z(\theta_0)v_0 \tag{2.50}$$

$$a^o_{ko} = -R_z(\theta_0)R_x(t_0)R_z(\theta_k(t))R_z(\theta_1)\left[v_1 + R_x(t_1)R_z(\theta_2)v_2\right] \tag{2.51}$$

onde $v_0 = (a_0, 0, d_0)$, $v_1 = (a_1, 0, d_1)$ e $v_2 = (0, 0, d_2)$. Finalmente, a velocidade angular da junta $\omega_k$ e sua aceleração angular $\dot{\omega}_k$ podem ser obtidas como

$$\omega^o_o = \dot{\theta}_k(t)R_z(\theta_0)R_x(t_0)z \tag{2.52}$$

$$\dot{\omega}^o_o = \ddot{\theta}_k(t)R_z(\theta_0)R_x(t_0)z \tag{2.53}$$

onde $z$ é o vetor unitário $(0,\ 0,\ 1)$.

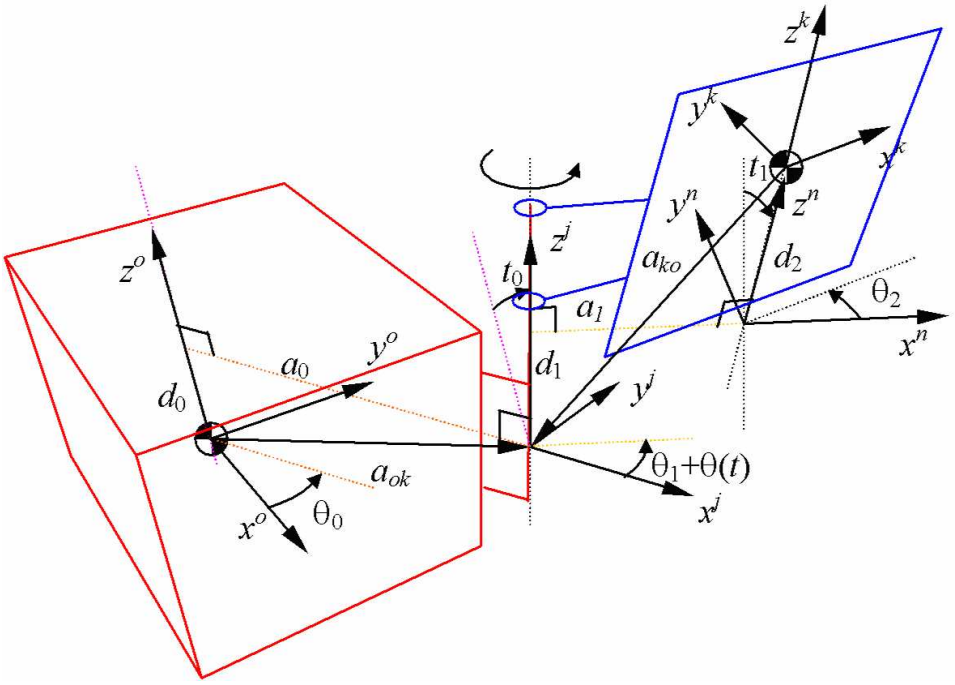


Figura 2.3 - Sistema de coordenadas do corpo principal, junta articulada e apêndice $k$.

### 2.4.2 Equações Cinemáticas do Movimento

Neste ponto a abordagem é a mesma usada para corpos rígidos sem apêndices. Serão usados quatérnios para a obtenção da matriz de atitude. A diferença está na forma como os quatérnios serão calculados, pois a estrutura da matriz de atitude é a mesma já apresentada anteriormente. O motivo para usarem-se quatérnios também é o mesmo, ou seja, a integração do movimento nas variáveis angulares de Euler leva usualmente a singularidades durante o processamento numérico, logo é conveniente efetuar-se esta integração por meio de quatérnions que eliminam o inconveniente das singularidades (Wertz, 2002):

$$\frac{dq}{dt} = \frac{1}{2}\tilde{\Omega}\left(\omega_o^o\right) q \tag{2.54}$$

com

$$q^T = [q_1 \quad q_2 \quad q_3 \quad q_4 \quad ] \tag{2.55}$$

e $\tilde{\Omega}(\cdot)$ representa, agora, a matriz de produto vetorial estendida (e com sinal trocado):

$$\tilde{\Omega}(\omega) = \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix} \tag{2.56}$$

As equações diferenciais dinâmicas e cinemáticas são integradas a partir das condições iniciais conhecidas $\omega_o(t_0)$ e $q(t_0)$, onde $t_0$ é o instante inicial. Uma vez obtido o vetor de quatérnions, determina-se a matriz de rotação $A$, que relaciona a posição inicial do satélite com a posição atual (Carrara, 1997)

$$A = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{2.57}$$

Fazendo a correspondência entre as componentes da matriz $A$ com

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{2.58}$$

pode-se obter os ângulos de Euler referentes a uma rotação 3-1-3, por exemplo:

$$\phi = arctan\left(\frac{a_{31}}{-a_{32}}\right), \quad 0 \leq \phi < 2\pi \tag{2.59}$$

$$\eta = arccos\left(a_{33}\right), \quad 0 \leq \eta < \pi \tag{2.60}$$

$$\psi = arctan\left(\frac{a_{13}}{a_{23}}\right), \quad 0 \leq \psi < 2\pi. \tag{2.61}$$

Resta agora obter-se $q(t_0)$ a partir da atitude inicial do satélite. Seja então a matriz de rotação $B$ que relaciona um sistema de referência inercial com a atitude inicial. Então, as componentes do quatérnion inicial podem ser obtidas, entre outras formas, por

$$q_4(t_0) = \frac{1}{2}\sqrt{1 + b_{11} + b_{22} + b_{33}} \tag{2.62}$$

$$q_1(t_0) = \frac{1}{4q_4}b_{23} - b_{32} \tag{2.63}$$

$$q_2(t_0) = \frac{1}{4q_4}b_{31} - b_{13} \tag{2.64}$$

$$q_3(t_0) = \frac{1}{4q_4}b_{12} - b_{21} \tag{2.65}$$

## 2.5 Integração das equações

As equações da dinâmica serão integradas numericamente levando em consideração as condições iniciais. Os resultados da integração permitem que seja feita a visualização gráfica do movimento de atitude.

É interessante notar que ocorre uma interseção entre o algoritmo de visualização e o de integração. Em outras palavras quer-se dizer que há laços computacionais comuns, ou seja, o laço computacional onde é feita a integração numérica é também responsável pela animação gráfica.

Nesta seção explanar-se-á como será viabilizada a integração das equações e, posteriormente, será apresentado o algoritmo de visualização.

Para a integração das equações será utilizado o método Runge-Kuttta de oitava ordem (Rao e Kuga, 1986). Este algoritmo é codificado em linguagem C, sendo utilizada também uma biblioteca de funções desenvolvidas para o simulador de atitude de apoio a missões espaciais para suporte à Plataforma Multi-Missão (PMM) da MECB (Carrara e Hassmann, 2007). Estas funções fazem parte do conjunto de programas destinados à qualificação do sistema de controle de atitude e órbita (AOC) do satélite.

A integração numérica visa a obtenção dos valores das velocidades angulares e quatérnios, sendo estes últimos centralizados no cálculo da matriz de atitude que proporcionará a visualização da atitude do satélite em questão. Deve-se lembrar que a animação e a integração ocorrem dentro de laços computacionais e que a animação requer a aplicação de várias matrizes de atitude obtidas em seqüência. Serão integradas, portanto, a equação matricial (2.39) e o conjunto de equações (2.41) (equações que regem o movimento rotacional).

### 2.5.1 Seqüência de integração e matriz de atitude

Para serem obtidos os resultados desejados para a animação computacional necessita-se de uma seqüência adequada de integração numérica. Os resultados da integração da equação (2.39) são armazenados em um vetor (aproveita-se o fato da equação ter caráter matricial). Esses valores são aplicados no conjunto de equações (2.41) e estas também são integradas, obtendo-se, então, os valores dos quatérnios.

Essa seqüência de integração é desenvolvida dentro de um laço de integração, permitindo obter-se um vasto conjunto de matrizes de atitude para um mesmo satélite. Esse conjunto possibilita a visualização da atitude, pois cada matriz de atitude corresponde a uma cena (*frame*); logo aplicando-se as matrizes, em seqüência, à animação tem-se o movimento do corpo (Carrara e Hassmann 2007). No final do laço, a cada instante (passo), obtém-se uma nova matriz de atitude e, conseqüentemente, a animação da propagação da atitude (ver Figura 2.4).

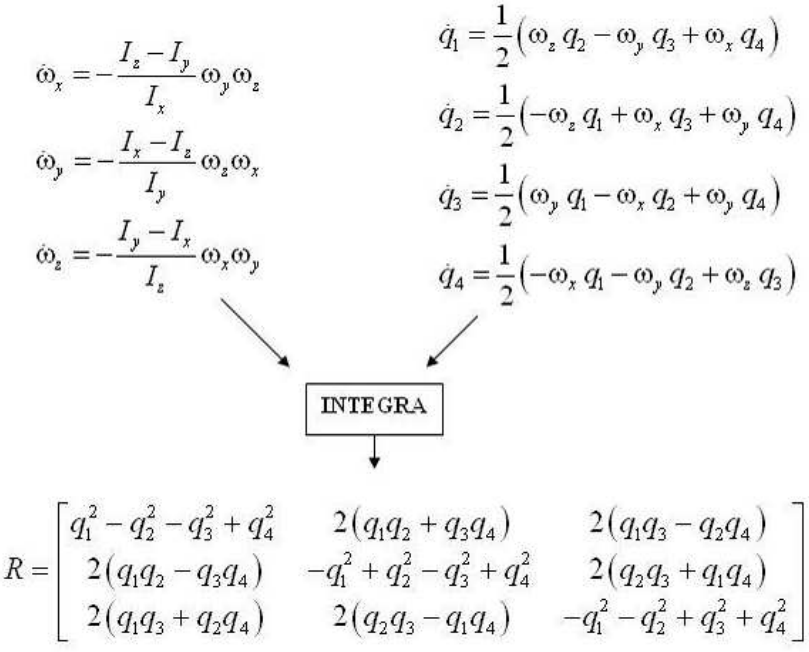


Figura 2.4 - Seqüência de integração para obtenção da matriz de atitude.

O laço de visualização é comum ao laço de integração. O OpenGL, particularmente, (ver Apêndice D) opera sempre com um laço computacional, pois a cada instante é criada uma nova imagem para apresentação na tela. Ao se fazer o laço de visualização comum ao de integração é possível animar-se a atitude em tempo real.

### 2.5.2 Carregando a Matriz de Atitude

Carregar uma matriz em OpenGl significa que todas manipulações de objetos gráficos, a partir do comando de carga (`glLoadMatrix`(), `glMultMatrixd`(), entre outras) seguirão as especificações desta matriz em particular. Isso significa que ao se necessitar de uma rotação qualquer em um objeto, por exemplo, via matriz de rotação, basta carregar-se a referida matriz na pilha de matrizes do OpenGl.

Para a simulação de atitude isso é bastante cômodo. Basta fazer-se a carga da matriz de atitude quando necessário e desenhar-se o satélite a partir desse ponto. No atual programa usou-se, particularmente, a função `glMultMatrixd`() (ver Apêndice D).

É necessário, porém, levar-se em consideração que matrizes de atitude têm três linhas e três colunas, enquanto que as funções citadas operam com matrizes $4\times4$. Isso é devido ao fato de que o OpenGl trabalha com matrizes homogêneas e qualquer matriz é vista como um vetor de dezesseis posições: as nove posições da matriz $3\times3$ e mais sete posições que dependerão do tipo de função utilizada (ver Anexo D). A função `glMultMatrixd`() multiplica a matriz que é seu parâmetro com a matriz do topo da pilha de memória. A matriz resultante é armazenada, então no topo da referida pilha (Wright e Lipchak, 2005).

## 2.6 Visualização sincronizada

A animação gráfica do movimento de atitude requer o conhecimento da atitude em intervalos relativamente pequenos de tempo, em geral menores do que $0.06s$, para garantir suavidade na transição entre as diversas imagens. Sabe-se que $0.06s$ é tempo mínimo necessário para haver percepção de imagens pelo olho humano (persistência da visão). Assim o integrador numérico de atitude deve ter seu passo ajustado de tal forma a garantir o atendimento deste requisito, ou seja, além de gerar a atitude sincronizada com o passo de integração, isto é, tal que o tempo de processamento seja inferior aos intervalos de tempo do passo, este passo deve ser também inferior a $0.06s$.

Deve-se lembrar que a formação de imagem é um resultado da integração. Como mencionado anteriormente, tanto a integração quanto a visualização pertencem ao mesmo laço computacional. Isso não significa que a atitude não possa ser obtida separadamente. Neste caso, porém, optou-se por conciliar atitude e visualização em um mesmo laço para

criar-se um realismo maior e economizar-se tempo computacional.

A matriz de atitude é calculada e aplicada à sub rotina de animação, fazendo com que haja o movimento (rotação) da geometria inicial do corpo conforme a regência dessa matriz de rotação. A movimentação ocorre em sincronismo com o relógio do computador ("clock"). Cada "time step"(intervalo de tempo) tem uma matriz de atitude característica resultante da propagação da atitude nesse determinado intervalo e com suas características e condições próprias (velocidades angulares por exemplo). Ao serem aplicadas sobre o corpo elas geram a animação.

O cálculo, porém, da atitude propagada é muito rápido e, conseqüentemente, a simples exibição do resultado provocaria o problema de uma movimentação em velocidade maior do que a natural. Para resolver esse problema e se ter uma correta simulação tanto da dinâmica, quanto da visualização, faz-se necessária a sincronização do processo com o "clock"do computador. Duas formas de sincronização foram testadas. Uma foi a de passo fixo com intervalo de espera; e a outra e definitiva foi a técnica de passo de integração ajustável, descrita a seguir.

Deve-se considerar ainda o que pode ocorrer à animação quando de uma interrupção do sistema operacional. Interrupções são condições naturais em sistemas preemptivos como o "Windows", mas podem ter um intervalo de duração um pouco maior, prejudicando a visualização. Uma parada de CPU, no entanto, não pára o tempo; logo haverá uma perda de sincronismo maior ou menor conforme for o tempo de parada.

### 2.6.1 Passo Fixo com Espera

O algoritmo de passo fixo com intervalo de espera é um laço de contagem de tempo que sincroniza o ciclo ou passo de integração (fixado previamente) com o "clock"da CPU. Neste caso, se houver uma interrupção, o laço de contagem de tempo será mais curto, pois o tempo inicial será comparado com o tempo em questão e nisso estará embutido a duração da interrupção. Caso a duração da interrupção seja maior que o tempo do passo haverá uma "aceleração"na atitude, pois não será necessário o laço de espera, e a integração segue adiante até haver a sincronização novamente.

O tempo do relógio do computador é armazenado numa variável antes do início da malha de integração. Ao fim de um passo da integração e apresentação do resultado na forma de animação o tempo do relógio é capturado novamente e é calculada a diferença entre o tempo final e inicial. Se esta diferença for inferior ao passo então o processamento da atitude foi mais rápido do que o tempo real e o processamento ficará retido neste ponto por intermédio de um laço de espera até que a condição de igualdade entre os tempos

seja satisfeita. O instante atual é então armazenado novamente para haver comparação no ciclo seguinte. Caso, porém, o tempo de interrupção seja maior que o passo de integração haverá um "salto" na integração, pois houve um avanço grande de tempo. Seria como um descarte de determinados ciclos de integração para iniciar-se mais à frente em um novo ciclo ao invés de se manter a seqüência correta determinada pelo passo de integração. O resultado numérico, de toda, forma independe do computador que está sendo utilizado, pois este influencia somente na contagem de tempo.

O problema deste método é que ele introduz um atraso no cálculo, logo há desperdício de tempo de processamento. É um método correto dinamicamente, mas de baixo rendimento. O algoritmo em pseudo linguagem é apresentado na Figura 2.5.

```
t0 = "clock"
Início da malha de integração
      Integrar de "t0" até "t0+dt"
      Apresentar o resultado da propagação de atitude
      Malha de espera até que "clock" ≥ "t0+dt"
      t0 = "clock"
Fim da malha de integração
```

Figura 2.5 - Algoritmo em pseudolinguagem para passo fixo com espera.

Caso, porém, o passo seja pequeno o suficiente para sincronizar-se com o "clock"do computador não haverá mais atraso. É necessário, de toda forma, descobrir-se qual passo é adequado para cada tipo de computador.

### 2.6.2 Passo de Integração Ajustável

Esta forma de sincronização é tão precisa quanto a anterior, mas proporciona um resultado visual melhor. Com o ajuste do passo é possível adaptar-se a quantidade de quadros para a visualização conforme a disponibilidade ou capacidade da "CPU". Tem-se então uma visualização mais suave e com a propagação de atitude sincronizada ao "clock"da "CPU". Se, neste caso, houver uma interrupção ter-se-á um atraso na propagação. Se a propagação estiver programada para ocorrer em um intervalo "t"ela acabará ocorrendo em "t"mais o tempo da interrupção.

Inicialmente fixa-se o passo de integração em um dado valor, de preferência bastante pequeno. Em seguida executa-se a malha de integração e visualização. A cada ciclo com-

pleto, ajusta-se o passo a ser empregado no próximo ciclo como sendo igual ao intervalo de tempo medido do último ciclo. Isto provoca a defasagem de um ciclo entre o instante de visualização e o instante de cálculo da atitude. Como o intervalo é bastante reduzido a defasagem é praticamente imperceptível e a visualização não apresenta as descontinuidades indesejáveis. O algoritmo em pseudo linguagem é apresentado na Figura 2.6.

```
dt =  1
t0 = "clock"
Malha de integração
      Integrar de "t" até "t+dt"
      Apresentar o resultado da propagação de atitude
      dt = "clock" - t0
Fim da malha de integração
```

Figura 2.6 - Algoritmo em pseudolinguagem para passo ajustável.

### 2.6.3 Comparação entre os dois métodos de visualização

Foram feitos testes exclusivos para a visualização com o objetivo de se comparar os métodos citados até agora. Para esses testes foram alteradas as cargas sobre a "CPU"em paralelo com a execução de uma simulação. Pelos resultados obtidos nestes testes, pode-se concluir que o número de quadros gerados por segundo é variável, dependendo do computador. Esta taxa varia conforme a utilização que é dada à "CPU". Quanto mais carga houver sobre a "CPU"menor o número de quadros por segundo.

O método de passo fixo é sem dúvida alguma a forma mais fácil de implementação deste programa. Não requer cálculo algum e se caracteriza unicamente pela introdução de um atraso na computação até haver sincronização entre cálculo e processador.

O método de passo fixo pode, porém, ter ou não um bom realismo visual. Se o passo de integração for igual ao tempo de processamento de um ciclo tem-se uma exibição ótima; se, contudo, o passo for maior que este intervalo tem-se uma exibição lenta (travamento entre quadros) e, do contrário, uma exibição em velocidade superior a real e conseqüentemente uma perda de sincronismo.

O método de passo ajustável consegue, por sua vez, resolver o problema de sincronismo sem causar dano ao realismo visual. Este método é, evidentemente, mais complexo do ponto de vista computacional, mas seu ganho é compensador. Por intermédio do método

de passo ajustável há uma simulação próxima da realidade tanto visualmente, quanto dinamicamente.

### 2.6.4 Visão em perspectiva

A projeção de um objeto distorce a sua aparência. A projeção ortográfica é paralela e mantém as mesmas dimensões de todos os lados da peça mesmo alterando-se as visadas. Não é, portanto, muito realista quando o observador começa a afastar-se da cena (Wright e Lipchak, 2005). Para obter-se pleno realismo dimensional optou-se pela projeção em perspectiva.

A projeção em perspectiva faz com que objetos mais afastados do observador sejam menores e mais finos do que os mais próximos. Um objeto de dimensões iguais, portanto, aparece, na cena, com um volume maior próximo ao observador e menor na região mais afastada (Wright e Lipchak, 2005).

O que é um conceito fácil de entender-se pode ser de difícil execução matemática, quando desenvolvido em um programa. O OpenGl, contudo, proporciona a função `gluPerspective` específica para este fim (ver Anexo F) (Wright e Lipchak, 2005). O uso dessa função proporciona realismo à cena unicamente com sua aplicação, podendo o resto da codificação manter-se inalterada e com transparência a uma eventual manipulação matemática sem haver, também, ônus no desempenho computacional, pois a função é executada em placa de vídeo (aceleradora), logo não há interferência no processamento realizado na CPU.

# 3 RESULTADOS

Inicialmente foram feitos alguns testes para a avaliação dos algoritmos de integração numérica. Para estes testes levou-se em consideração um corpo rígido, com simetria axial e sem torques externos e sem apêndices. Objetivou-se visualizar a dinâmica de corpo rígido, pois para este sabia-se quais os resultados esperados já que a integração analítica era possível.

Foi avaliada a velocidade angular em eixo principal com a matriz de inércia diagonal, com elementos iguais; matriz de inércia diagonal, com 2 elementos iguais; matriz de inércia diagonal, com elementos diferentes e matriz de inércia com elementos não nulos fora da diagonal. Foi avaliada também a velocidade angular em eixo não alinhado com matriz de inércia diagonal, com elementos iguais; matriz de inércia diagonal, com 2 elementos iguais; matriz de inércia diagonal, com elementos diferentes e matriz de inércia com elementos não nulos fora da diagonal. Todos os resultados foram considerados coerentes com a dinâmica do problema.

## 3.1 Dinâmica de corpo com apêndices

Um corpo rígido não era o único objetivo. As funções desenvolvidas neste trabalho devem simular um satélite dotado de apêndices articulados. Até este momento o integrador utilizado foi o desenvolvido por Rao e Kuga (Rao e Kuga, 1986) adaptado para esse trabalho. A dinâmica de apêndices era meramente pictórica, ou seja, um apêndice eram apresentados em tela, mas a cena não estava dinamicamente correta.

Para a dinâmica de corpos com apêndeces usou-se um pacote de funções desenvolvido por Carrara (Carrara e Hassmann, 2007) especialmente para a simulações deste tipo. A utilização deste pacote objetivou não somente a viabilização de uma simulação mais completa juntamente com o presente trabalho, mas também o teste da própria dinâmica.

Os resultados obtidos com este pacote de funções foi bastante positivo. As simulações se mostraram coerentes e a união do pacote de funções para simulação de satélites com apêndices e das funções de visulização mostrou-se adequada para o maior entendimento de uma simulação.

## 3.2 Testes da integração

A integração é viabilizada por uma sub rotina computacional que promove a integração numérica das equações (Rao e Kuga, 1986). Esta subrotina, inicialmente desenvolvida em linguagem FORTRAN, teve de ser adaptada à linguagem Visual C++. Este tipo de

adaptação pode, porém, introduzir erros, logo foram necessários testes para validar o integrador; e também a dinâmica do corpo rígido.

Para averiguar-se a correção da subrotina de integração usou-se como estratégia a visualização do movimento de nutação do satélite com um eixo de simetria cilíndrico. Sabe-se, do omovimento de nutação de um corpo simétrico, que este descreve um movimento composto que pode ser visualizado como o de um cone fixado ao corpo rolando (sem deslizar) sobre um outro cone fixado ao sistema inercial, cujo eixo de simetria coincide com o momento angular (Wertz, 2002; Hughes 1986). Esses cones não são interessantes para uma visualização do movimento de atitude, mas oferecem um auxílio adicional neste caso específico, (nutação) quando as equações de Euler podem ser integradas de forma analítica, da qual se conhece a solução. O cone fixado ao corpo é denominado cone do corpo, enquanto que o outro é o cone do espaço.

Para esses testes considerou-se o movimento rotativo de um corpo rígido livre de torques. Considerou-se que o momento angular esteja alinhado com o eixo inercial coordenado $Z$. Admitindo, além disso, que o eixo $z$ do sistema de coordenadas fixado ao corpo seja o eixo de simetria, então para haver movimento de nutação a velocidade angular deve ser decomposta numa parcela na direção do eixo de simetria ($\omega_z$) e numa outra parcela perpendicular a esta (adotado como $\omega_y$).

O movimento do nutação em corpos simétricos apresenta dois comportamentos distintos, correspondentes aos casos nos quais os momentos do inércia transversais do corpos são maiores ou menores do que o momento de inércia axial (Wertz, 2002). Estes dois casos correspondem a corpos com formato de lápis e disco (moeda) respectivamente. Para dar maior realismo às simulações, estabeleceu-se que as dimensões do corpo refletissem as características dos momentos de inércia. Em outras palavras, as dimensões $x$, $y$ e $z$ de um corpo simétrico (com $x = y$) e densidade homogênea são calculados por

$$\begin{aligned} x &= \sqrt{I_2 + I_3 - I_1} \\ y &= \sqrt{I_1 + I_3 - I_2} \\ z &= \sqrt{I_1 + I_2 - I_3} \end{aligned} \tag{3.1}$$

onde $x$, $y$ e $z$ são as dimensões gerais do satélite em cada eixo em relação ao seu centro de massa e $I_1$, $I_2$ e $I_3$ os seus momentos principais de inércia, com relação aos eixos $x$, $y$ e $z$, respectivamente.

A atitude inicial deve ser fixada de forma a fazer com que a direção do momento angular fique alinhada ao eixo $Z$ inercial. Sabendo-se que, no movimento de nutação, o momento angular, $L$, o vetor velocidade angular, $\omega$, e o eixo de simetria do corpo ($z$) são coplanares

(Wertz, 2002 e Hughes, 1986), pode-se com isso estabelecer o eixo $Z$ inercial no plano $xy$ do sistema fixado ao corpo como mostra a Figura 3.1.

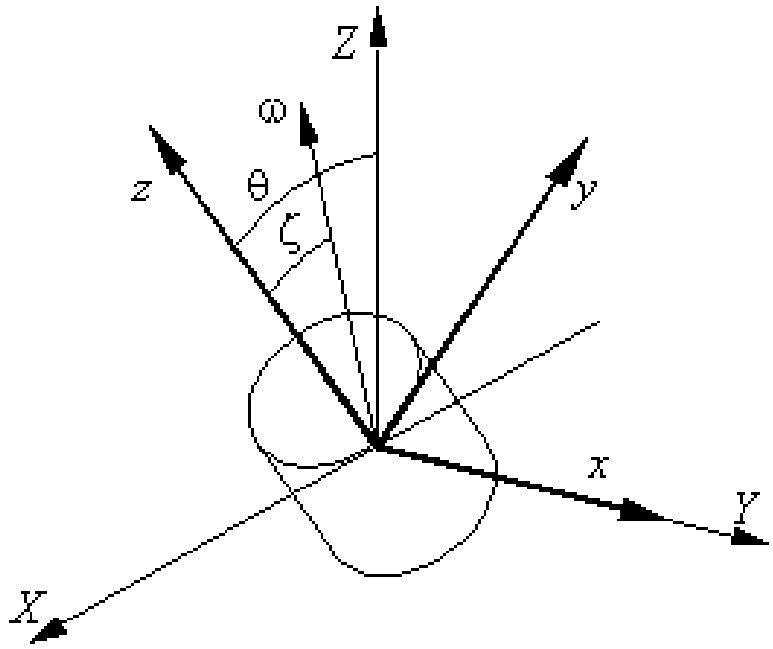


Figura 3.1 - Eixo $Z$ inercial no plano $xy$ do sistema fixado ao corpo.

A posição dos demais eixos é arbitrária e foi escolhida de forma a fazer com que o eixo $x$ coincida, na condição inicial, com o eixo $Y$ inercial. Com isso a matriz de atitude fica

$$q = rmxquat \begin{pmatrix} 0 & -cos\theta & sen\theta \\ 1 & 0 & 0 \\ 0 & sen\theta & cos\theta \end{pmatrix} \tag{3.2}$$

onde $\theta$ é o ângulo de nutação, dado por

$$\theta = arccos\left(\frac{L_z}{|L|}\right) \tag{3.3}$$

Com base na matriz de atitude inicial, calcula-se o quatérnio por meio da função `rmxquat`() (Carrara e Hassmann, 2007). Essa condição para testes obriga que o sistema de coordenadas esteja no centro de massa do satélite, sendo a rotação em torno do eixo $Z$.

Para o teste referenciado o momento angular $L$ está alinhado ao eixo $Z$ do referencial

inercial. $\omega$, em um espaço inercial, gira em torno de $L$ em um cone com ângulo de abertura em relação ao vértice dado por $\theta - \varsigma$ chamado cone do espaço (Wertz, 2002). De forma semelhante, $\omega$ mantém um ângulo fixo $\zeta$ com o eixo $z$ e gira em torno deste em um cone chamado de cone do corpo. O ângulo $\zeta$ pode ser calculado por

$$tg\zeta = \left(\frac{I_3}{I_1}\right) tg\theta \tag{3.4}$$

As Figuras 3.2 e 3.3 abaixo ilustram os resultados obtidos para corpos rígidos e sem apêndices. Pode-se ver os dois casos clássicos de nutação (Wertz, 2002). O primeiro caso é o de um corpo rígido na forma de "lápis"e o segundo caso é o de um corpo rígido na forma de "moeda".

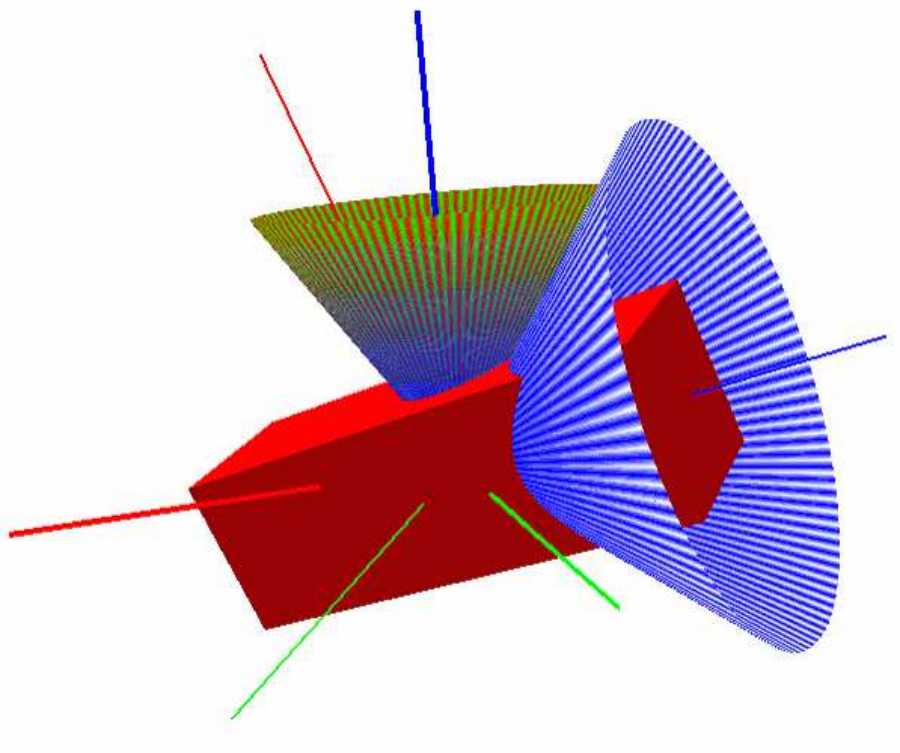

Figura 3.2 - Cones de nutação quando $I_x > I_z$.

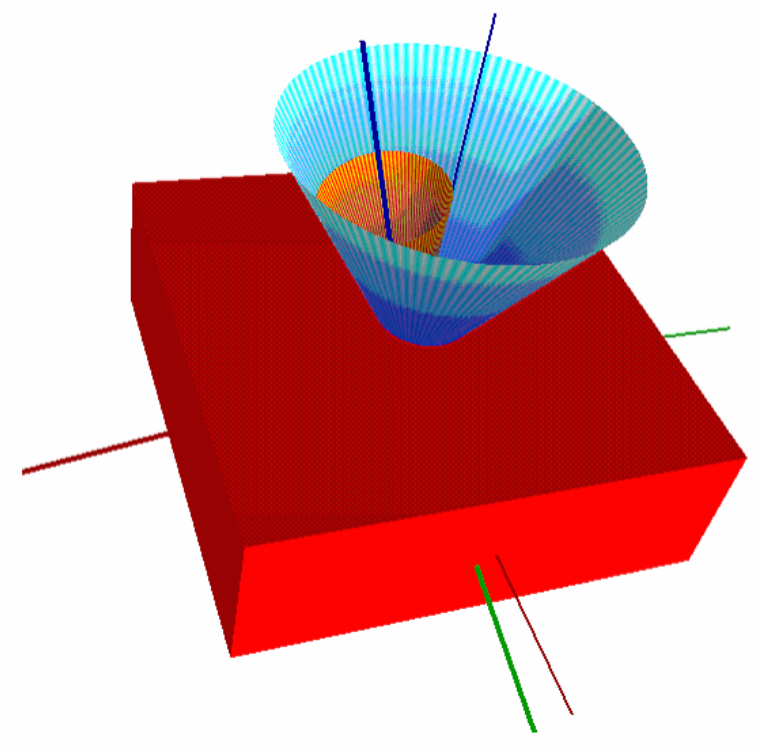

Figura 3.3 - do nutação quando $I_x < I_z$.

Existem, porém, casos em que a teoria de nutação não pode ser aplicada, e nesses casos os cones de nutação com apresentados até agora não fazem mais sentido. Para esses sistemas a integração das equações do movimento é obrigatoriamente numérica, pois elas não tem solução analítica. Esses casos também foram abordados neste trabalho e podem ser exemplificados por um satélite com apêndices articulados como mostrado na Figura 3.4. Os resultados dos testes para satélites com apêndices foram bastante coerentes. Nesta figura também é possível observar a aplicação de texturas tanto ao satélite quanto ao painel solar.

Outra forma utilizada para a testagem da integração foi a comparação com o integrador já existente no departamento de mecânica espacial e controle do INPE (Rao e Kuga, 1986). O critério para a comparação foram os quatérnios e o movimento de nutação, este último como mencionado acima. Ao serem comparados os dois conjuntos de quatérnios poderia se saber se o integrador usado para a criação deste conjunto de funções estaria correto ou não, pois o integrador original (Rao e Kuga, 1986) já havia sido largamente testado. Durante os testes foi ainda possível visualizar-se os cones de nutação e verificar-se a coerência do fenômeno conforme a bibliografia (Wertz, 2002). Os parâmetros comuns de teste foram o tempo de simulação de três segundos e os parâmetros dos integradores, sendo o erro relativo igual a 1.0e-6, o erro absoluto de 1.0e-6, o início do intervalo de inegração

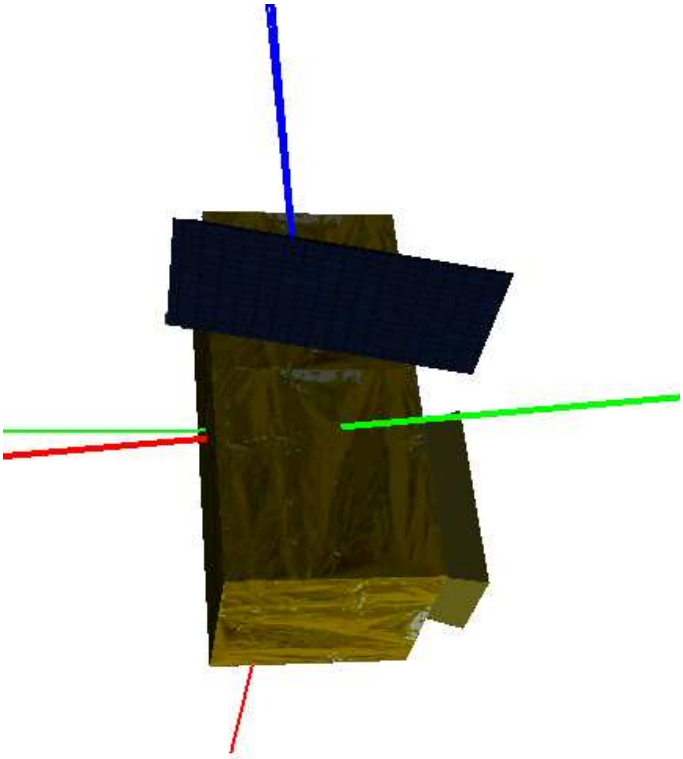

Figura 3.4 - Satélite com apêndice (painel solar).

em $t = 0$, o fim do intervalo de integracao em tout $= 0.01$ e o passo interno do integrador (dtint) de 0.01. São comuns, também, a todos os testes as velocidades angulares dos corpos distribuidas vetorialmente da seguinte forma $\omega = [0, 0.8, 1]$. Os valores iniciais dos quatérnios não são algo comum a todas as simulações, pois são obtidos por intermédio da matriz de inércia do satélite e do vetor de velocidades angulares conforme Wertz (Wertz, 2002), mas a forma de cálculo é a mesma.

### 3.2.1 Resultados para $I_x = I_y > I_z$

Como o corpo simulado é simétrico a matriz de inércia é diagonal, sendo os seguintes valores não nulos $I_{11} = I_{22} = 20$ e $I_{33} = 5$.

Seguem os gráficos de comparação entre os dois conjuntos de quatérnios obtidos nas simulações.

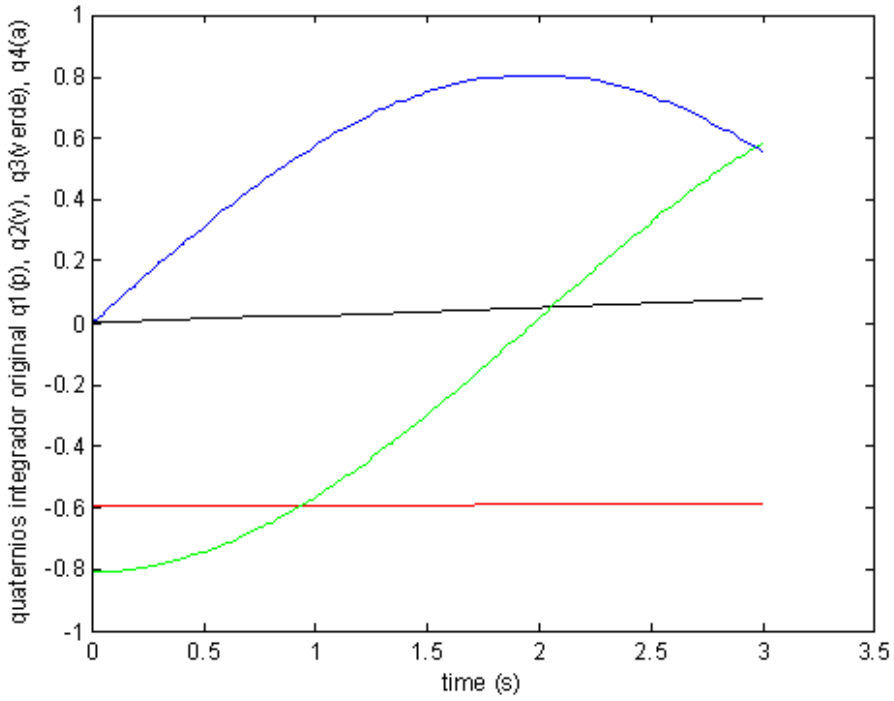


Figura 3.5 - Quatérnios obtidos com o integrador original para $I_x = I_u > I_z$ (Roo e Kuga, 1986).

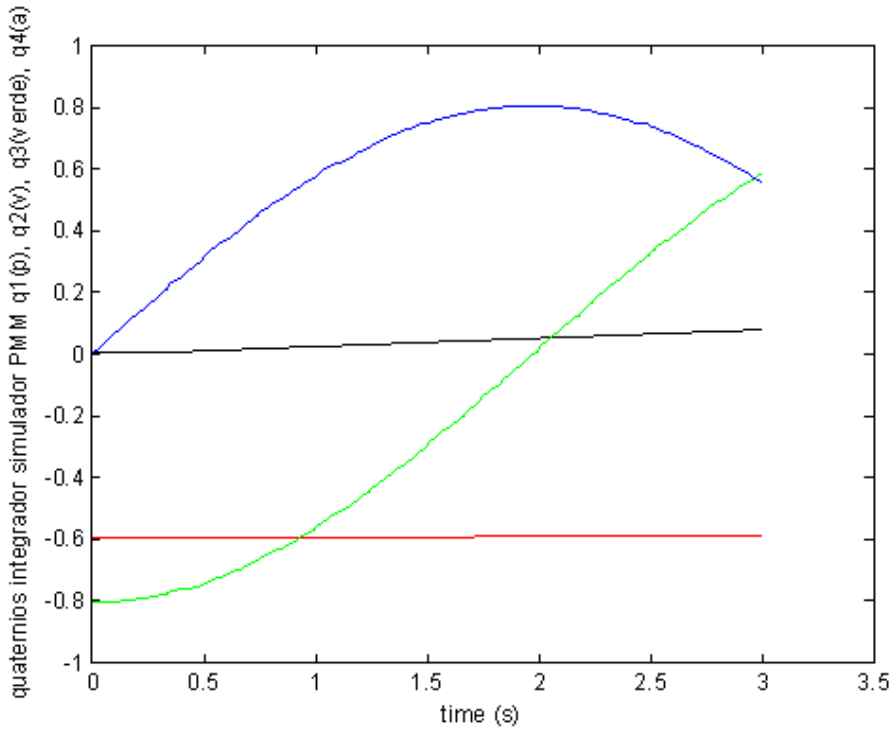


Figura 3.6 - Quatérnios obtidos com o novo integrador ($I_x = I_y > I_z$).

### 3.2.2 Resultados para $I_x = I_y < I_z$

Como o corpo simulado é simétrico a matriz de inércia é diagonal, sendo os seguintes valores não nulos $I_{11} = I_{22} = 5$ e $I_{33} = 20$, seguem os gráficos de comparação entre os dois conjuntos de quatérnios obtidos nas simulações.

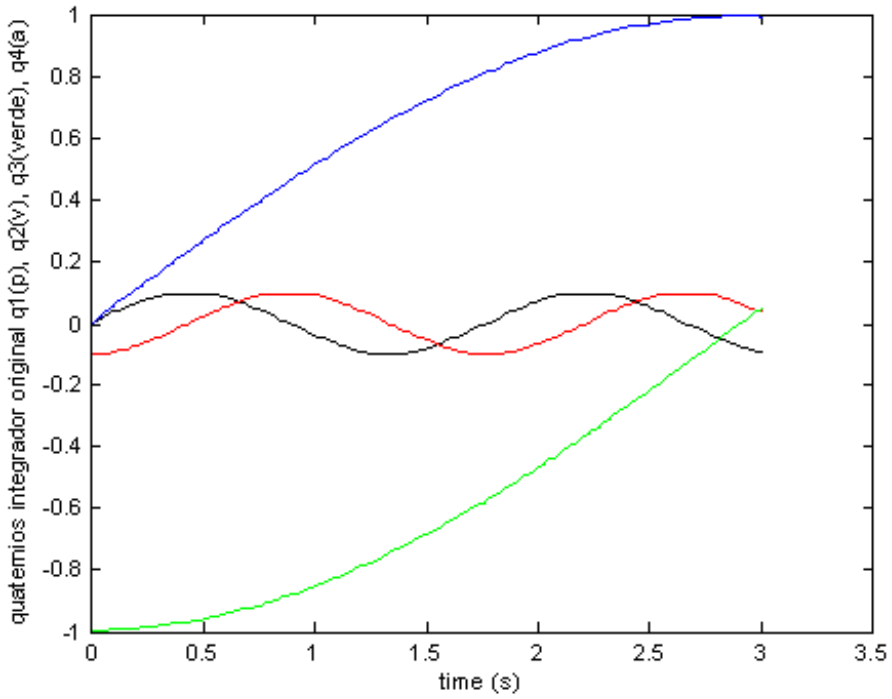


Figura 3.7 - Quatérnios obtidos com o integrador original para $I_x = I_y < I_z$ (Roo e Kuga, 1986).

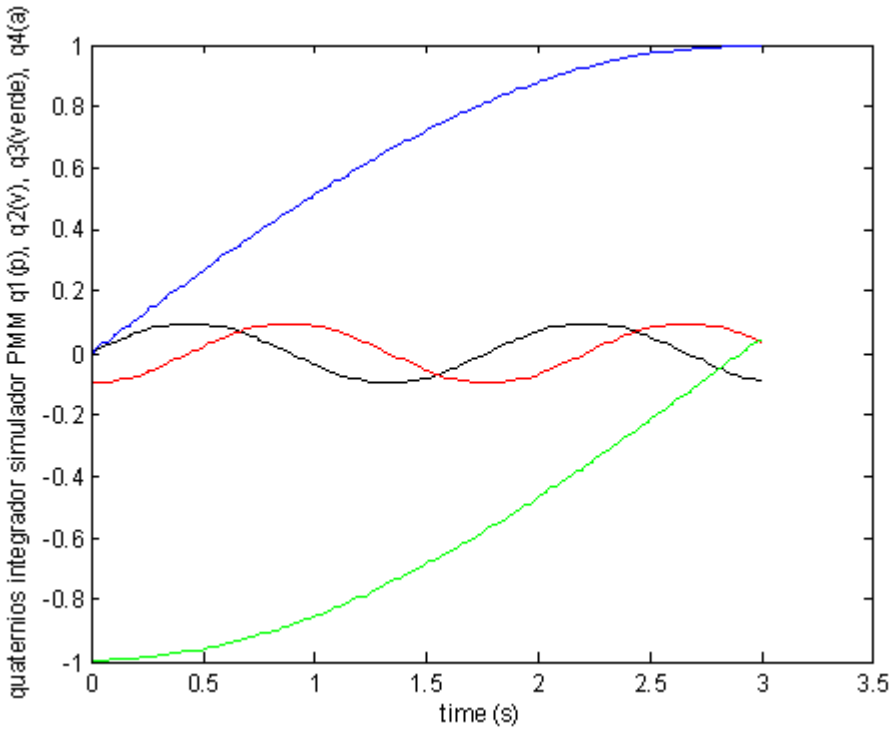


Figura 3.8 - Quatérnios obtidos com o novo integrador ($I_x = I_y < I_z$).

# 4 CONCLUSÃO

Desenvolveu-se neste trabalho uma série de funções que visam a animação gráfica da simulação de atitude de satélites. Este objetivo foi atingido inclusive com a geometria de um satélite sendo lida de um arquivo independente (tipo AC3D ou 3DS) e reproduzida na tela e animado conforme a atitude programada. Esse conjunto de funções permitiu a visualização da simulação de atitude de satélites artificiais com apêndices articulados. Permitiu também que uma biblioteca de funções de simulação de atitude (Carrara e Hassmann, 2007) fosse testada. Essa biblioteca é de extrema importância para um trabalho deste tipo, mas a falta de uma vilsualização impedia que houvesse uma avaliaçõa mais profunda dos resultados. Até o presente momento, no instituto, os resultados eram apresentados em gráficos bidimensionais. Esta biblioteca de simulação atualmente em desenvolvimento que visa o projeto PMM (Carrara e Hassmann, 2007) pode agora englobar as funções de visualização desenvolvidas neste trabalho.

O trabalho aqui desenvolvido é bastante versátil. O conjunto de funções desenvolvido pode ser utilizado em qualquer outro simulador que seja desenvolvido ou em futuras versões deste que está em desenvolvimento (Carrara e Hassmann, 2007). É possível, além disso, ler-se geometrias de satélites editadas em mais de um tipo de editor gráfico. No presente momento é possívelo somente a leitura de geometrias editadas por dois editores que são o "Blender"e o "3D Max". Tem-se, portanto, a sugetão para trabalhos futuros, ou seja, a construção de novas funções que leiam geometrias provenientes de outros editores e que sejam adicionadas a este trabalho, reforçando o cunho genérico desta ferramenta.

Este trabalho apresenta ainda todo o equacionamento para satélites com apêndices articulados. Este equacionamento pode ser utilizado para qualquer outro trabalho com satélites que apresentem apêndices articulados ou simplesmente para corpos rígidos. No segundo caso faz-se necessária um simplificação das equações.

Os resultados obtidos permitem que tenha uma noção mais realística de uma simulação de atitude de satélites com apêndices articulados. Espera-se, porém, em trabalhos futuros possibilitar ao usuário uma maior versatilidade com relação a aplicação de texturas e cores aos objetos e mais facilidade no que diz respeito à iluminação sobre esses objetos.

Deve-se ter em mente também que um conjunto de funções como o desenvolvido aqui não é algo dinâmico e novas funções com outras facilidades podem ser desenvolvidas. Funções para a leitura de geometrias oriundas de outros editores que não somente os dois utilizados até agora são um bom exemplo de contunuidade deste trabalho. Vale lembrar, finalmente, que nos apêndices deste trabalho há toda a orientação de como escrever um código mínimo

para apresentação de uma tela em OpenGl e de como, utilizando o conjunto de funções, preparar uma simulação.

# 5 REFERÊNCIAS BIBLIOGRÁFICAS


ASADA, H.; SLOTINE, J.-J. E.; Robot Analysis and Control. John Wiley and Sons, New York, 1986.

CARRARA, V.; Redes Neurais Aplicadas ao Controle de Atitude de Satélites com Geometria Variável, S.J.Campos, SP, INPE - 6384-TDI/ 603, 1997.

CARRARA, V.; HASSMANN, C. H. G.; An Attitude Simulator to Support Space Missions, 6th Brazilian Conference on Dynamics, Control and Their Applications, S.J.Rio Preto, 4-7 Maio, SP,2007.

COHEN, M.; MANSSOUR, I. H., OpenGL - Uma Abordagem Prática e Objetiva, 1a ed., 2006.

COUTINHO, M. G., Dynamic simulations of Multibody Systems, New York, Springer, 2001.

FONSECA, I. M.; LOURENÇÃO, P. T. M.; OLIVEIRA, J. R. F.; Procedimentos para a Estabilização Passiva de um Satélite sob Condições Iniciais Críticas. VIII COBEM, 1985.

GREENWOOD, D. T., Principles of Dynamics. 2nd ed, Prentice-Hall, New York, 1988.

HUGHES, P. C. Spacecraft Attitude Dynamics., McGraw Hill, New York, 1986.

KERNIGHAN, B. W.; RITCHIE, D. M., The C Programming Language. 2nd ed., Prentice-Hall, New Jersey, 1988.

KUIPERS, J. B., Quaternions and Rotation Sequences. Princeton University Press, Princeton, NJ, 1999.

MEIROVITCH, L., Methods of Analytical Dynamics. Dover Publications., 1998.

RAO, K. R.; KUGA, H. K., Manual de uso de um conjunto de integradores numéricos para problemas de condições iniciais, S.J.Campos, SP, INPE - 3830-RPI/ 154, 1986.

STANKOVIC, J. A.; Misconceptions About Real - Time Computing. IEEE 1988.

WERTZ, J. R.; Spacecraft Attitude Determination and Control. D. Reidel Pub. Co., Holanda, 1978.

WOO, M.; NEIDER, J.; DAVIS, T.; SHREINER, D. OpenGL Programming Guide. 3rd ed., Addison-Wesley, EUA, 1997.

WRIGHT, R. S. Jr.; LIPCHAK, B., OpenGL Super Bible. 3rd ed., SAMS Publishing, 2005.

Referências Bibliográficas Complementares

AMABILIS; http://www.amabilis.com/products.htm, 2008.

AUTODESK; http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112, 2008.

BLENDER; http://www.blender.org/, 2008.

GAMESTUDIO; http://www.3dgamestudio.com/, 2007.

KITWARE; http://public.kitware.com/VTK/index.php, 2008.

LAMINAR_RESEARCHE; http://www.x-plane.com/, 2007.

MICROSOFT_GAMES; http://www.microsoft.com/games/flightsimulatorx/, 2007.

NASA; http://www.nasa.gov/, 2007.

NASA_SIMLABS; http://www.simlabs.arc.nasa.gov/vms/vms.html, 2007.

OGRE; http://www.ogre3d.org/, 2007.

OPENDX; http://www.opendx.org/, 2008.

SCHWEIGER, MARTIN; http://orbit.medphys.ucl.ac.uk/, 2006.

SOURCEFORGE; http://irrlicht.sourceforge.net/tut008b.html, 2007.

TERATHON_SOFTWARE; http://www.terathon.com/c4engine/index.php, 2007.

TOUCHDOWENTERTAINMENT; http://www.touchdownentertainment.com/jupiter.htm, 2007.

WEB3D_CONSORTIUM; http://www.web3d.org/, 2008.

# A APÊNDICE A - OPERAÇÕES BÁSICAS COM QUATÉRNIOS

Quatérnios podem ser manipulados de forma semelhante a números complexos. Seguem, abaixo, algumas operações básicas com quatérnios, considerando-se $q_1 = s_1 + v_1$ e $q_2 = s_2 + v_2$ dois quatérnios onde $s_i$ é a parte real do quatérnio e $v_i$ é a parte vetorial (Coutinho, 2001).

## A.1 Adição

Operação semelhante a dos números complexos e dada por

$$q_1 + q_2 = (s_1 + s_2) + (v_1 + v_2) \tag{A.1}$$

## A.2 Produto Escalar

Equivale à soma entre o produto escalar da parte real e o produto escalar da parte vetorial conforme o exemplo abaixo.

$$q_1 \cdot q_2 = s_1 s_2 + v_1 \cdot v_2 \tag{A.2}$$

Com a notação adotada neste trabalho, porém, ter-se-á:

$$q_2^T q_2 = s_1 s_2 + v_1^T v_2 \tag{A.3}$$

## A.3 Multiplicação

Esta operação também é análoga à multiplicação de complexos. A parte vetorial, porém, segue as seguintes regras

$$\begin{aligned} ij &= -ji = k \\ jk &= -kj = i \\ ki &= -ik = j \\ ijk &= -1 \end{aligned} \tag{A.4}$$

Para efetuar-se a multiplicação, usando as regras acima tem-se

$$\begin{aligned} q_1 q_2 &= (s_1 + v_1)(s_2 + v_2) \\ &= (s_1 + v_{1x}i + v_{1y}j + v_{1z}k)(s_2 + v_{2x}i + v_{2y}j + v_{2z}k) \\ &= (s_1 s_2 - v_{1x}v_{2x} - v_{1y}v_{2y} - v_{1z}v_{2z}) + s_1(v_{2x}i + v_{2y}j + v_{2z}k) + s_2(v_{1x}i + v_{1y}j + v_{1z}k) + \\ &+ (v_{1y}v_{2z} - v_{1z}v_{2y})i + (v_{1z}v_{2x} - v_{1x}v_{2z})j + (v_{1x}v_{2y} - v_{1y}v_{2x})i \\ &= (s_1 s_2 - v_1^T v_2) + (s_1 v_2 + s_2 v_1 + \Omega(v_1)v2) \end{aligned} \tag{A.5}$$

Lembramos ainda que essa operação não é comutativa assim como não são as rotações com matrizes.

## A.4 Conjugado

A concepção é exatamente a mesma usada em números complexos, logo

$$\bar{q_1} = \overline{s_1 + v_1} = s_1 - v_1 \tag{A.6}$$

Tem-se ainda

$$\bar{q_1}\bar{q_2} = \overline{q_1 q_2} \tag{A.7}$$

## A.5 Módulo

O módulo é computado com o produto escalar entre o quatérnio e o seu conjugado.

$$|q|^2 = q^T \bar{q} = \bar{q}^T q = s^2 + v^T v = s^2 + v_x^2 + v_y^2 + v_z^2 \tag{A.8}$$

Sendo ainda

$$|q_1 q_2|^2 = |q_1|^2 |q_2|^2 \tag{A.9}$$

## A.6 Inversão

A inversão é diretamente derivada do módulo da seguinte forma

$$q^{-1} = \frac{\bar{q}}{|q_1|^2} \tag{A.10}$$

# B APÊNDICE B - MATRIZES E VETORES PARA A INTEGRAÇÃO

Considerando que a simulação e o controle de atitude são equacionados em termos de matrizes e vetores, será utilizado um conjunto de estruturas que facilitam a codificação dos programas. No contexto utilizado aqui as matrizes são todas de dimensão 3, ou seja, são matrizes quadradas $3 \times 3$. Os vetores são igualmente tridimensionais, e os quatérnions são quadridimensionais. As estruturas disponíveis são: `matrix3`, `vector3` e `quaternion`. É utilizado o conceito de vectrizes, que combina as propriedades de matrizes com a de vetores.

## a - **Motivação**

A utilização de matrizes se faz clara quando analisamos equações como a eq. B.1:

$$\dot{w} = I^{-1}(-w \times Iw) \tag{B.1}$$

## b - **Formalização**

Um vetor é uma estrutura composta por 3 valores reais (precisão dupla, ou *double*), na forma:

$$v_3 = (v_3.\_1 \quad v_3.\_2 \quad v_3.\_3) \tag{B.2}$$

Uma matriz $a$ é formada por 3 vetores linha, $a.\_1$, $a.\_2$ e $a.\_3$, e portanto tem a forma

$$a = \begin{pmatrix} a.\_1 \\ a.\_2 \\ a.\_3 \end{pmatrix} = \begin{pmatrix} a.\_1.\_1 & a.\_1.\_2 & a.\_1.\_3 \\ a.\_2.\_1 & a.\_2.\_2 & a.\_2.\_3 \\ a.\_3.\_1 & a.\_3.\_2 & a.\_3.\_3 \end{pmatrix} \tag{B.3}$$

Finalmente, o quatérnio tem a forma

$$q = \begin{pmatrix} q.\_1 \\ q.\_2 \\ q.\_3 \\ q.\_4 \end{pmatrix} \tag{B.4}$$

Nota-se que a matriz poderia ser definida como sendo um vetor de 9 componentes. A diferença serve para lembrar que existem produtos de matrizes por vetores, mas não existem produtos de vetores de 6 dimensões por outro de 3 dimensões, por exemplo. Em outras palavras, no produto de uma matriz $a$ por um vetor linha $b$, deve-se sempre

lembrar que, de fato, se está fazendo um produto na forma $ab^T$, onde o sobrescrito $T$ indica a transposição do vetor. O código para esta multiplicação é simplesmente $a*b$, sem a necessidade de transpor o vetor.

Para utilizar o pacote e criar variáveis com as estruturas de vetores, matrizes ou quatérnios, usa-se:

```
#include "matrices.h"
...
matrix3 mat_rot, mat_rot1, mat_rot2;
vector3 vec_pos;
quaternion q;
...
```

#### c - **Operações**

Estão implementadas neste pacote as principais operações matemáticas entre matrizes e vetores, para permitir a representação simbólica destas operações. Assim, um produto de duas matrizes é indicado por:

```
mat_rot = mat_rot1 * mat_rot2
```

Qualquer composição envolvendo operações de adição, subtração e multiplicação é igualmente válida:

```
mat_rot = (mat_rot1 + mat_rot2) * mat_rot3;
```

### B.1 Laço de visualização

Como comentado anteriormente se usará a biblioteca gráfica *OpenGl* para a implementação deste trabalho. Este é descrito como uma interface para o *hardware* gráfico. Em essência é uma biblioteca gráfica e de modelagem.

Ao se trabalhar com essa biblioteca necessita-se, invariavelmente, de um laço para a criação da imagem, conforme Figura B.1. Esse laço se faz necessário, pois a imagem é renovada constantemente na tela. Isso permite a animação visual, muito embora se possa criar imagens estáticas também.

Neste caso o laço será de grande importância, pois não só proporcionará a animação, mas dentro dele será executado, também, o laço de integração.

```
void main () // programa principal

{

      //declaração de variáveis

      //chamadas às funções de configuração do satélite e
      integrador

      //malha de visualização...

      //malha de integração

      //fim da malha de integração

      //fim da malha de visualização

      return;
}
```

Figura B.1 - Laço para a criação da imagem.

## B.2 Laço de integração

Internamente um integrador numérico de passo fixo possui igualmente ciclos de tempo: há o passo de integração, o ciclo de integração e o período de integração, mostrados na figura B.2 abaixo. O passo de integração é o menor intervalo de tempo no qual as equações da dinâmica serão avaliadas. O ciclo de integração é o intervalo no qual deseja-se conhecer o estado da dinâmica, uma vez que a integração nos passos é realizada sem controle externo. Cada ciclo de integração corresponde a vários passos de integração. Finalmente, o período de integração é o intervalo de tempo que se deseja simular.

Tipicamente, passos de integração de atitude situam-se entre milésimos e décimos de segundo. Ciclos de integração são compatíveis com ciclos de controle e situam-se entre décimos de segundo e poucos segundos. Períodos de integração em geral duram de alguns minutos a algumas horas. Estes intervalos não devem ser confundidos com o tempo de processamento, que é a duração de execução do programa. No caso de simuladores em tempo real, é necessário garantir que o tempo de processamento de passos e ciclos seja inferior ao período de integração.

Um programa típico escrito em C e que utilize as funções descritas aqui possui a seguinte forma geral.

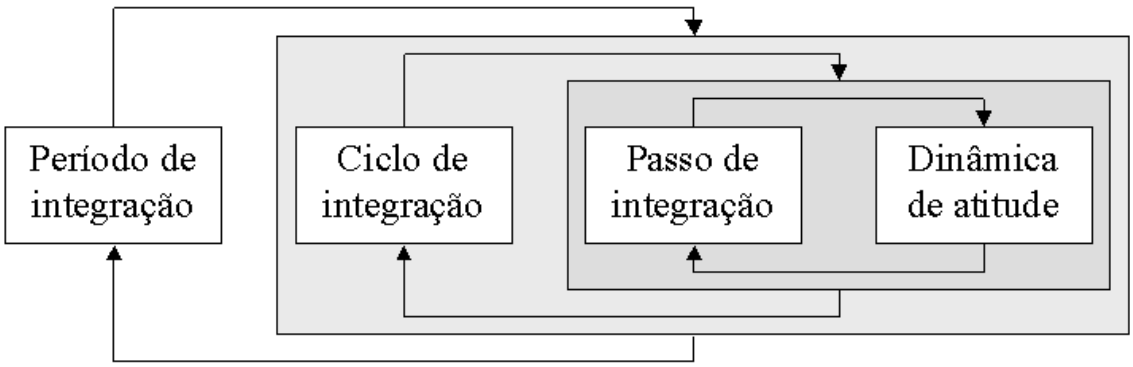


Figura B.2 - Intervalos de tempos presentes no integrador numérico de atitude.

```
// inclusão das constantes e dos protótipos das funções

#include "consta.h"

#include "attdyn.h"

#include "attaux.h"

#include "matrices.h"

void main () // programa principal

{

      // declaração de variáveis e chamadas das funções de configuração
do satélite e integrador

      // malha de visualização

      // início malha de integração

      // fim  malha de integração

      // fim da malha de visualização

      return;

}
```

Figura B.3 - Forma geral de um programa escrito em C e que utilize as funções de simulação.

# C APÊNDICE C - OpenGL

## C.1 O que é *OpenGL*

*OpenGL* é uma especificação aberta e multi plataforma de uma biblioteca de rotinas gráficas e de modelagem, ou API (*Application Programming Interface*), utilizada para o desenvolvimento de aplicações gráficas, tais como jogos e sistemas de visualização. O *OpenGL* é extremamente eficiente, uma vez que muitas das suas funções são executadas no processador das placas gráficas mais modernas. Programas são escritos em uma linguagem de programação, como C/C++, e fazem chamadas às rotinas dessa biblioteca (Cohen e Manssour, 2006).

Definida como "um programa de *interface* para *hardware* gráfico", essa biblioteca proporciona rotinas gráficas e de modelagem bi (2D) e tridimensional (3D), extremamente portáveis e rápidas. Usando *OpenGL* é possível criar gráficos 3D com uma qualidade visual muito boa. A maior vantagem, entretanto, na sua utilização é a rapidez, uma vez que usa algoritmos cuidadosamente desenvolvidos e otimizados.

*OpenGL* não é uma linguagem de programação, mas sim uma poderosa e sofisticada API. Normalmente se diz que um programa é baseado em *OpenGL* ou é uma aplicação *OpenGL*, o que significa que ele é escrito em alguma linguagem de programação que faz chamadas a uma ou mais funções do *OpenGL*.

As aplicações *OpenGL* variam de ferramentas CAD a programas de modelagem usados para criar personagens para o cinema, bem como simulações científicas. Além do desenho de primitivas gráficas, tais como linhas e polígonos, *OpenGL* dá suporte a iluminação, colorização, mapeamento de textura, transparência e animação, entre muitos outros efeitos. Atualmente, *OpenGL* é reconhecido e aceito como um padrão API para desenvolvimento de aplicações gráficas 3D em tempo real.

## C.2 Como trabalha o *OpenGL*

Quando se está utilizando *OpenGL* é preciso apenas determinar os passos necessários para alcançar a aparência ou efeito desejado de uma cena. Estes passos envolvem chamadas a esta API portável que inclui aproximadamente 250 comandos e funções (200 comandos do *core OpenGL* e 50 da GLU (*OpenGL Utility Library*).

Por ser portável, *OpenGL* não possui funções para gerenciamento de janelas, interação com o usuário ou arquivos de entrada/saída. Cada ambiente, como por exemplo o *Microsoft Windows*, possui suas próprias funções para estes propósitos. Não existe um formato de

arquivo *OpenGL* para modelos ou ambientes virtuais. *OpenGL* fornece um pequeno conjunto de primitivas gráficas para construção de modelos, tais como pontos, linhas e polígonos. A biblioteca GLU (que faz parte da implementação *OpenGL*) é que fornece várias funções para modelagem, tais como superfícies quádricas, curvas e superfícies NURBS (*Non Uniform Rational B-Splines*) (Woo et al, 1997; Wright e Lipchak, 2005).

O *OpenGL* é uma máquina de estados, conforme descrito por (Woo et al, 1997). É possível colocá-la em vários estados (ou modos) que não são alterados, a menos que uma função seja chamada para isto. Por exemplo, a cor corrente é uma variável de estado que pode ser definida como branco. Todos os objetos, então, são desenhados com a cor branca, até o momento em que outra cor corrente é especificada.

*OpenGL* mantém uma série de variáveis de estado, tais como estilo (ou padrão) de uma linha, posições e características das luzes, e propriedades do material dos objetos que estão sendo desenhados. Muitas delas referem-se a modos que podem ser habilitados ou desabilitados com os comandos `glEnable`() e `glDisable`().

Cada variável de estado possui um valor inicial (*default*) que pode ser alterado. As funções que são utilizadas para saber o seu valor são: `glGetBooleanv`(), `glGetDoublev`(), `glGetFloatv`, `glGetIntegerv`, `glGetPointerv`() ou `glIsEnabled`(). Dependendo do tipo de dado, é possível saber qual destas funções deve ser usada (Woo et al, 1997).

O *OpenGl* é uma biblioteca extensa, com muitos recursos e, naturalmente, nem sempre é fácil a sua uilização. Por este motivo foram desenvolvidos e disponibilizados diversos pacotes de visualização gráfica e científica baseados em *OpenGl*, destinados a facilitar o uso de funções gráficas por usuários, como por exemplo o *OpenDX* (Opendx, 2008), o VTK (Kitware, 2008) e o VRML (Web3D_Consortium, 2008). Estes pacotes aglutinam vários funções do OpenGl num conjunto reduzido de funções e dirigido a aplicações.

A facilidade de gerar cenas 3D com alta velocidade faz do *OpenGl* uma biblioteca exclusivamente gráfica. Assim, qualquer processamento que envolva dinâmica, detecção de colisões ou interferência entre objetos, simulação de objetos naturais como água, nuvens, ondas, ou ainda um maior realismo gráfico como fenômenos ópticos (refração, difração, reflexão e sombra) fica a cargo do programador. Diversos pacotes em *OpenGl*, adicionam recursos extras como estes já comentados. Estes pacotes recebem o nome genérico de "motores de jogos"(*game engines*) e são distribuídos gratuitamente *(Ogre 3D (Ogre, 2007), Jupiter (Touchdownentertainment, 2007), Irrlich e Kochol (Sourceforge, 2007))*, ou comercialmente *(3D Game Studio (GameStudio, 2007) , C4 Game Engine (Terathon_Software, 2007))*.

### C.2.1 "*Pipeline*"do *OpenGL*

A palavra *pipeline* é usada para descrever um processo que pode ter dois ou mais passos distintos. A Figura C.1 mostra uma versão simplificada do *pipeline OpenGL*. Como uma aplicação faz chamadas às funções API *OpenGL*, os comandos são colocados em um *buffer* de comandos. Este *buffer* é preenchido com comandos, vértices, dados de textura, etc. Quando este *buffer* é "esvaziado", os comandos e dados são passados para o próximo estágio (Wright e Lipchak, 2005).

Várias implementações de *OpenGL* são similares com relação ao *pipeline*. Todas, porém, seguem, pelo menos, a configuração geral da Figura C.1.

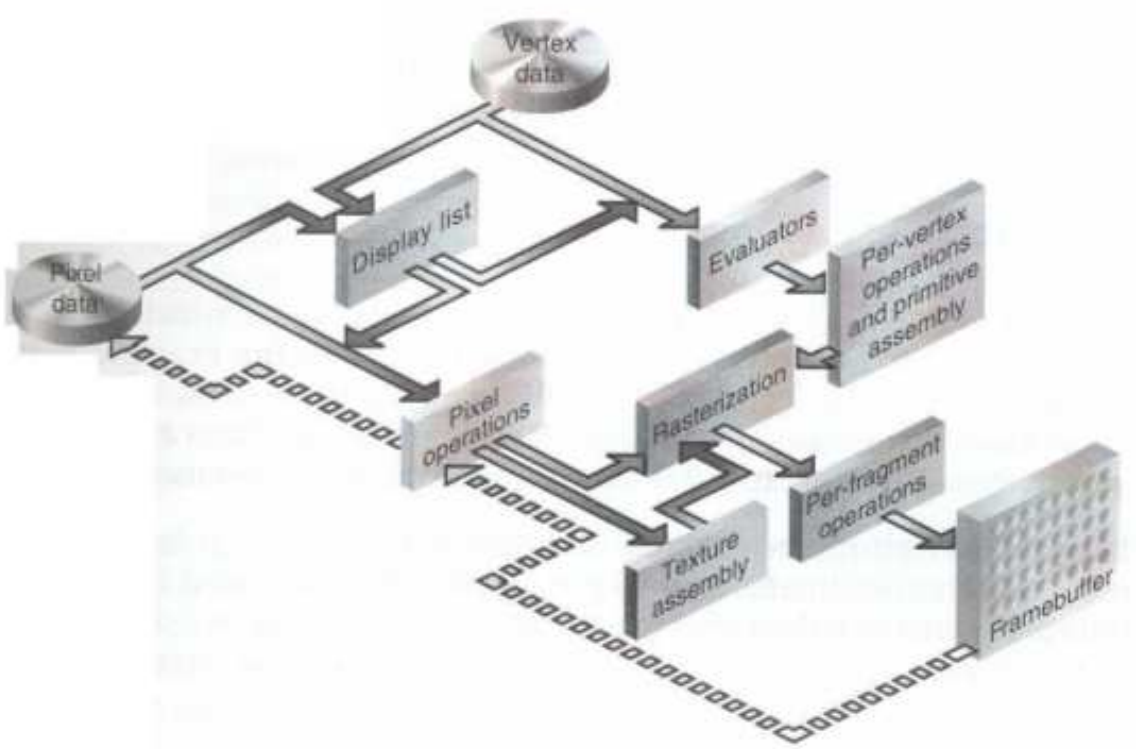


Figura C.1 - *PipeLine* do OpenGl.

Segue-se um comentário sobre cada item do *pipeline*. A organização deste *PipeLine* é como apresentada na Figura C.1.

a - **Lista de imagens (*Display List*)**

Todos os dados gráficos estão armazenados nesta lista. Estejam eles descritos na forma

de primitivas ou *bit maps*. Quando a lista de imagens é executada são apresentadas as imagens (Woo et all, 1997).

b - **Avaliadores (*Evaluators*)**

Todas as primitivas gráficas são, eventualmente, descritas por vértices. Curvas paramétricas e superfícies são descritas inicialmente por pontos de controle e funções polinomiais chamadas de funções base. Avaliadores promovem um meio obter, dos pontos de controle, os vértices usados para descrever a superfície. O método é um mapeamento polinomial que pode gerar superfícies, cores, coordenadas e outros elementos a partir dos pontos de controle (Woo et al, 1997).

c - **Operações a cada vértice (*Per-Vertex Operations*)**

Converte os vértices (dados de coordenadas de vértices) em primitivas. É um passo que pode ser bastante congestionado pela quantidade de operações que aqui podem ser executadas. Nesta fase é, por exemplo, processada a textura de uma superfície também. Textura e luz são processadas nesta fase (Woo et al, 1997).

d - **Montagem de primitivas (*Primitive Assembly*)**

Proporciona a montagem das primitivas aplicando a elas cor, textura e, principalmente, perspectiva. Nesse passo é criada a idéia de fundo e terceira dimensão.

e - **Operações com "Pixels"(*Pixel Operations*)**

Dados geométricos tomam um caminho dentro do *pipeline* e dados de *pixels*, ou texturas, tomam caminho diferente. Esses apresentam-se compactados e agrupados em um vetor na memória do sistema. Esses dados, ao passarem pelo *pipeline* são descompactados e processados, sendo escritos em um mapa de texturas ou enviados para a "rasterização".

f - **Montagem de textura (*Texture Assembly*)**

Textura é aplicada a objetos geométricos para torná-los mais realísticos. Quando várias texturas estão disponíveis deve-se escolher qual será usada ou não e em quais objetos serão aplicadas. A escolha é feita via comandos pelo usuário, mas a organização em memória e na apresentação da cena é feita na etapa de montagem de textura.

g - **Rasterização (*Rasterization*)**

É a transformação de dados geométricos em *pixels*. Cada *pixel* tem correspondência a um fragmento (normalmente um triângulo ou quadrilátero). A correspondência é calculada para cada parte da imagem (linhas, texturas, etc.) para a apresentação na tela.

h - **Operações com fragmentos (*Fragment Operations*)**

Nesse momento várias operações são executadas com o intuito de formalizar os fragmentos. Umas das operações, por exemplo, é a aplicação de textura onde cada *texel* (elemento de textura) é gerado, da memória de textura, para cada fragmento e a ele aplicado. Após o processamento é feita a apresentação no `Frame Buffer`.

## C.3 Utilização

Como uma API, *OpenGL* segue a convenção de chamada da linguagem C. Isto significa que programas escritos em C podem facilmente chamar funções desta API, tanto porque estas foram escritas em C, como porque é fornecido um conjunto de funções C intermediárias que chamam funções escritas em assembler ou outra linguagem (Wright e Lipchak, 2005).

Apesar do *OpenGL* ser uma biblioteca de programação "padrão", existem muitas implementações desta biblioteca, por exemplo, para *Windows* e para *Linux*. A implementação utilizada no ambiente *Linux* é a biblioteca `Mesa`. Também existem implementações para os compiladores *Visual C++*, *Borland C++*, *Dev C++*, *Delphi* e *Visual Basic*. Pode-se facilmente obter as bibliotecas e a documentação de cada implementação na internet.

No caso da implementação da *Microsoft*, o sistema operacional fornece os arquivos `opengl32.dll` e `glu32.dll`, necessários para execução de programas *OpenGL*. Além disso, são fornecidas com suas ferramentas de programação, como por exemplo com o *Microsoft Visual C++*, as bibliotecas `opengl32.lib` (*OpenGL*) e `glu32.lib` (GLU - biblioteca de utilitários *OpenGL*). Assim, para criar programas com ferramentas *Microsoft* que usem *OpenGL*, tal como o *MS Visual C++* 6.0, é necessário adicionar estas duas bibliotecas à lista de bibliotecas importadas. Protótipos para todas as funções, tipos e macros OpenGL estão (por convenção) no *header* `gl.h`. Os protótipos da biblioteca de funções utilitárias estão em um arquivo diferente que é o `glu.h`. Estes arquivos normalmente estão localizados em uma pasta especial ou diretamente na pasta de compilação e execução do programa. O código da figura C.2 mostra as inclusões típicas para um programa *Windows* que usa *OpenGL* (Wright e Lipchak, 2005):

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
```

Figura C.2 - Código de cabeçalho

Para utilizar *OpenGL* e GLUT com ferramentas *Microsoft*, no caso o *MS Visual C++* 6.0, é necessário:

1. Fazer o *download* e instalar a biblioteca GLUT;
2. Criar um projeto para adicionar as bibliotecas necessárias com o programa;
3. Incluir os *headers* adequados (<`gl/gl.h`> e <`gl/glu.h`>, ou <`gl/glut.h`>).

A implementação `Mesa` do *OpenGL* utilizada no ambiente *Linux* é um projeto de *software* livre. A distribuição desta biblioteca, cuja versão 5.0 possui suporte para o conjunto de instruções da *OpenGL* 1.4, inclui `makefiles` e instruções para a sua correta utilização.

## C.4 Matrizes no *OpenGl*

O *OpenGl* não representa matrizes 4×4 de forma bi dimensional, mas na forma de um vetor de dezesseis posições. Essa forma de representação é chamada de "ordenação majoritária de matrizes por coluna"(Wright e Lipchak, 2005). Essa forma de estruturação de dados é preferida à tradicional, pois para obter-se um vetor direcional ou a tradução de uma matriz, um vetor de 16 posições faz somente uma cópia da memória para obter todos os dados de um único lugar. Na estrutura tradicional é necessário acessarem-se 3 ou 4 posições de memória para obter-se um simples vetor da mesma matriz (Wright e Lipchak, 2005).

## C.5 Funções de destaque utilizadas

`gluPerspective`

Essa função cria a matriz que descreve uma perspectiva em coordenadas reais de tela. A razão de aspecto deve coincidir com a da janela de visualização. A divisão de perspectiva é baseada no campo de visada e nas distâncias próxima e longe dos planos de projeção (Wright e Lipchak, 2005).

`glLoadMatrix(GLdouble *m)`

Substitui a matriz corrente (carregada anteriormente) pela matriz que lhe foi atribuída na forma de um vetor de 16 posições. Tem como parâmetro um vetor de 16 posições que representa uma matriz 4×4.

`glMultMatrixd(GLdouble *m)`

Multiplica a matriz corrente pela especificada como parâmetro. Tem como parâmetro de entrada, também, um vetor de 16 posições que representa uma matriz 4×4.

# D APÊNDICE D - FUNÇÕES DESENVOLVIDAS NESTE TRABALHO

## D.1 Camera() e Teclado()

As funções `Camera`() e `Teclado`() funcionam em complemento uma da outra. Necessitam, também, de uma estrutura para a criação de uma variável global. Esta variável fará as movimentações, em última instância, da cena.

A função `Camera`() tem como parâmetro uma estrutura do tipo `MOVIMENTO`. Ela é responsável pelo ajuste da imagem quando a cena é apresentada pela primeira vez. Permite ao usuário ver todos os eixos coordenados (inerciais e do satélite) sem que haja superposição de eixos, quando inicializada com o parâmetro global carregado com os primeiros valores de posição de cena. É responsável também por todos os outros movimentos, quando recebe a função `Teclado`() como parâmetro.

A função `Teclado`() permite que sejam alterados os parâmetros de câmera, quando são acionadas determinadas teclas de comando. As teclas são as seguintes

- Seta para cima e para baixo (+ e - ): translação em $Z$ de tela (zoom).
- Seta para direita e para esquerda (`VK_RIGHT` e `VK_LEFT`): rotação em $X$ de tela.
- Teclas de seta para cima e para baixo (`VK_UP` e `VK_DOWN`): rotação em $Y$ de tela.
- Teclas "O"e ".": rotação em $Z$ de tela.
- Tecla "Home"(`VK_HOME`): volta à posição inicial da cena.

## D.2 void MatrizAtitude(matrix3 M, vector3 vec)

Função que carrega a matriz de atitude no contexto do programa por intermédio da função `glMultMatrixd`(...) (ver Apêndice C para carga de matrizes). Tem como parâmetros de entrada uma matriz 3×3 e um vetor de três posições.

A matriz $M$ deve ser calculada pelas funções `quatrmx(satélite)` ou `get_body_rmatrix(apêndices)` (Carrara e Hassmann, 2007). A matriz $M$ é a matriz de atitude e é obtida via os quatérnios iniciais ou resultantes da integração das equações de movimento do satélite (Wertz, 2002). A função transpõe a matriz de entrada e adiciona um vetor caso se esteja lidando com um apêndice e não com o satélite propriamente dito. Em seqüência a essas operações a matriz é carregada no contexto do programa (Wright e Lipchak, 2005) ou (ver Apêndice C).

**D.3** void EixosCoordenados(float comp, float width)

Função que desenha os eixos coordenados. Conforme a escolha do usuário poderão ser os eixos inerciais, os eixos coordenados do satélite ou os eixos coordenados de qualquer um dos oito possíveis apêndices.

A função tem dois parâmetros de entrada. Ambos do tipo *float* (Kernighan e Ritchie, 1988) que representam o comprimento e a espessura de cada linha que compõem os eixos. A posição e o eventual movimento dos eixos é responsabilidade do usuário. Essas características são proporcionadas via linhas de código ou por intermédio de funções do *OpenGl* (ver Apêndice C).

**D.4** ANGULO AnguloCone(matrix3 I, vector3 x_w)

Esta função tem caráter complementar. Ela é do tipo `ANGULO` (ver Apêndice E), e retorna uma estrutura que contém os ângulos $\theta$ e $\varsigma$ (equações (5.3) e (5.4), respectivamente) que serão utilizados para o cálculo dos raios dos cones do espaço e do satélite quando do desenho desses.

**D.5** void Cones(float altura, int tipo, ANGULO ang)

A função `cones` desenha os cones imaginários de deslocamento que ocorrem quando do movimento de nutação de um satélite. Vale salientar que esses cones só fazem sentido quando houver simetria na geometria do satélite e $I_1 = I_2$ ($I_1$, $I_2$ e $I_3$ são os elementos da diagonal principal da matriz de inércia do satélite) (Wertz, 2002). Para a presente simulação usou-se o eixo $Z$ inercial como o eixo principal do cone do espaço e o eixo $Z$ do sistema do satélite como o eixo principal do cone do satélite.

Esta função tem três parâmetros de entrada.

a) Altura: é a altura do cone desejada.

b) Tipo: significa o tipo de cone que vai ser desenhado. Terá duas opções, sendo 1 para cone do espaço e 2 para cone do corpo.

c) Ângulo: é o ângulo de abertura de cada cone com relação ao eixo da altura. Este parâmetro é uma estrutura que carrega concomitantemente os ângulos $\theta$ e $\varsigma$ para cada cone (Wertz, 2002). Esse parâmetro é o retorno da função `AnguloCone`(...).

**D.6** void DesenhaCorpo_AC3D(CORPOS c, int index, int normal)

Esta função é responsável por reproduzir a geometria do satélite criada pelo programa *Blender 3D*. Todas as informações são retiradas do respectivo arquivo gerado pelo programa, que é do tipo AC3D.

Esta função tem dois parâmetros de entrada.

a) O primeiro parâmetro é uma estrutura do tipo corpos que detém todas as características necessárias para reproduzir o satélite ou apêndices na tela (para a estrutura CORPOS ver Anexo E).

b) Index: esse parâmetro pode ter os valores de -1 até 7. O valor -1 significa que o corpo a ser desenhado é o satélite (corpo principal) os outros valores de 0 até 7 significam as 8 possibilidades de apêndices que este simulador pretende representar (Carrara e Hassmann, 2007).

c) A variável "normal"indica a direção do vetor normal a superfície que está sendo reproduzida em tela, pois essa informação é importante para obter-se a correta iluminação (Wright e Lipchak, 2005).

**D.7** quaternion Quat_Inicial(matrix3 I, vector3 x_w){

Função que tem como parâmetros de entrada a matriz de inércia do satélite e o vetor velocidade angular. Esta função retorna o quatérnio inicial para dar início à simulação e tem característica de inicialização. A utilização dessa função, porém, é fundamental somente se houver nutação com visualização dos cones e se o vetor momento angular for paralelo ao eixo $Z$ do sistema inercial.

**D.8** void LeVertices_AC3D(char file[256], int index)

Esta função não lê somente os vértices que caracterizam a figura que será reproduzida, mas permite que, absolutamente, todas as estruturas do satélite que será visualizado sejam armazenadas em uma única variável do tipo `CORPOS` (ver Apêndice E).

A função `LeVertices`(...) tem dois parâmetros de entrada. O primeiro é um vetor de caracteres que deverá armazenar o nome de um arquivo. O segundo é um índice do tipo inteiro. A variável `file` deve conter o nome de arquivo correspondente a uma das partes que compõem o satélite que será visualizado, já o índice indica se a parte em questão é o corpo principal (índice -1) ou de algum apêndice (índices 0 a 7).

Chamando-se essa função em seqüência e mudando-se o nome de arquivo de entrada e o índice, a variável do tipo `CORPOS` (variável global) é carregada. Todo o conjunto (satélite e apêndices) fica, portanto, de fácil acesso para a simulação.

**D.9** int LeNumero(char string[256])

Esta função é uma facilidade criada para a leitura de um número em uma seqüência de caracteres. A função tem como parâmetro de entrada a seqüência em que está o número e retorna o número desejado. É útil somente para número inteiros. Esta função é utilizada na leituras dos arquivos de geometria que sejam do tipo texto.

**D.10** void ClearReset()

Função bastante simples que não tem parâmetros de entrada nem retorna valor algum. O objetivo desta função é "limpar a cena"antes de cada nova imagem ser formado. Outro objetivo é a chamada da função `Camera`(...). Deve-se lembrar que esta função está dentro do laço de formação de imagens.

**D.11** STRING PulaLinha(FILE *fp, char palavra[256], int num)

Esta é uma função auxiliar. Seu objetivo é permitir que se percorra um arquivo do tipo AC3D tendo como referência uma seqüência de caracteres (parâmetro de entrada) que deverá ter par dentro do arquivo. A função retorna a linha em que é encontrada a seqüência desejada. Tem ainda um ponteiro para o arquivo que deve ser percorrido e o tamanho da palavra a ser procurada.

**D.12** char Load_3DS (char *p_filename, int index, float scale)

Função que carrega o conteúdo de um arquivo do tipo 3DS em uma variável global do tipo `obj_type` (ver Apêndice E). Dessa forma é possível reproduzir o conteúdo do arquivo em tela de forma animada.

Esta função tem como parâmetros de entrada o nome do arquivo que detem a geometria que deve ser carregada e uma variável do tipo `float` que é um multiplicador de escala. Esse multiplicador tem a função de aumentar ou diminuir o desenho na tela conforme a vontade do usuário.

**D.13** void DesenhaCorpo_3ds(obj_type object[8][10], int index, int normal)

Reproduz o conteúdo da variável global do tipo `obj_type` (ver Apêndice E) em tela. Utiliza as facilidades do *OpenGl* para tal tarefa (Wright e Lipchak, 2005).

Esta função tem como parâmetros de entrada uma variável do tipo `obj_type` e as variáveis `index` e `normal`. A variável `index` indica se a geometria que deve ser desenhada é o corpo principal ou é um apêndice e a variável `normal` indica a direção do vetor normal a superfície que está sendo reproduzida em tela para obter-se a correta iluminação.

### D.14 void Load_AC3D(char file[256], int index, float scale)

Função que carrega o conteúdo de um arquivo do tipo AC3D em uma variável global do tipo `CORPOS` (ver Apêndice E). Dessa forma é possível apresentar a geometria descrita no arquivo na janela de visualização.

Esta função tem como parâmetros de entrada o nome do arquivo que detem a geometria que deve ser carregada, o índice desse arquivo na hierarquia da simulação (corpo principal ou apêndice) e uma variável do tipo `float` que é um multiplicador de escala. Esse multiplicador tem a função de aumentar ou diminuir o desenho na tela para se ajustar a escala dos demais objetos.

# E APÊNDICE E - ESTRUTURAS COMPUTACIONAIS CRIADAS PARA ESTE TRABALHO

As estruturas abaixo citadas são desenvolvidas em linguagem *Visual* C++ 6.0.

## E.1 VERTICES

Estrutura que será posteriormente utilizada na estrutura `corpos`. Esta estrutura tem por objetivo armazenar as coordenadas $x$, $y$, $z$ de cada vértice da geometria do satélite ou apêndice contidos em um arquivo do tipo AC3D.

```
struct VERTICES{
     float x;
     float y;
     float z;
};
```

## E.2 FACE

Esta estrutura armazena os vértices necessários para compor cada face de um corpo, bem como as características de textura da face em questão. É também utilizada para compor a estrutura maior chamada `corpos`.

```
struct FACE{
     int numSurf;
     vector3 material;
     int mat;
     int refs;
     int v[4];
};
```

## E.3 VERT_FACE

Estrutura intermediária composta pelas estruturas `vertices` e `face`. Esta estrutura comporta 1000 faces e 3950 vértices por vez e será utilizada na estrutura principal chamada `corpos`.

```
struct VERT_FACE{
     VERTICES ponto[3950];
     FACE face[1000];
};
```

## E.4 CORPOS

Estrutura principal para o armazenamento das características geométricas do satélite e de cada apêndice que formam a geometria completa que será visualizada. Esta estrutura serve unicamente para armazenar arquivos do tipo AC3D e é composta por duas variáveis do tipo `VERT_FACE` sendo uma a variável `sat` que deverá conter todas as características do satélite e um vetor de oito posições de nome `ap` que comporta as características de cada um dos oito apêndices possíveis de serem criados na simulação (Carrara e Hassmann, 2007).

A grande vantagem de usar-se estruturas definidas dessa forma reside no fato de declarar-se uma única variável para descrever toda a geometria, tanto do satélite quanto dos apêndices. Esta estrutura é carregada em uma única chamada da função `LeVertices`(...) (ver Apêndice D). Sempre que o usuário a acessar poderá escolher entre acessar as informações do satélite (variável `sat`) ou as informações de um dos oito possíveis apêndices (variável `ap`).

```
struct CORPOS{
     VERT_FACE sat;
     VERT_FACE ap[8];
};
```

## E.5 MOVIMENTO

Esta estrutura permite que haja comunicação entre as funções `teclado` e `câmera` por intermédio de uma variável deste tipo.

```
struct MOVIMENTO{
     //Camera - Rotacoes iniciais
     GLfloat xRot;
     GLfloat yRot;
     GLfloat zRot;
     //Camera - Translacoes iniciais
     GLfloat xTrans;
     GLfloat yTrans;
     GLfloat zTrans;
};
```

## E.6 ANGULO

Esta estrutura armazena os ângulos $\theta$ e $\varsigma$ que serão usados no cálculo do raio dos cones de nutação.

```
struct ANGULO{
     double teta;
     double zeta;
};
```

### E.7 vertex_type

Esta estrutura fornece o tipo para a posterior leitura das coordenadas dos vértices para arquivos 3DS. Cada vértice tem as três coordendas armazenadas em $x$, $y$ e $z$, respectivamente.

```
typedef struct{
     float x,y,z;
}vertex_type;
```

### E.8 polygon_type

Na estrutura de um arquivo 3DS há a informação dos vértices que formam uma face. As faces reunidas formarão toda a geometria do corpo. A estrutura abaixo permite que sejam manipuladas essas informções. Os vértices relativos a cada face ficam aí armazenados nesta estrutura.

```
typedef struct{
     int a,b,c;
}polygon_type;
```

### E.9 mapcoord_type

Armazena as coordenadas de textura de cada vértice de um arquivo do tipo 3DS.

```
typedef struct{
     float u,v;
}mapcoord_type;
```

### E.10 obj_type

Esta estrutura armazena todas as informações necessárias para remontar os polígonos de uma estrutura em tela. Serve, porém, somente para arquivos do tipo 3DS.

```
typedef struct{
     char name[20];
     int vertices_qty;
     int polygons_qty;
     vertex_type vertex[MAX_VERTICES];
```

```
    polygon_type polygon[MAX_POLYGONS];
    mapcoord_type mapcoord[MAX_VERTICES];
    int id_texture;
}obj_type;
```

### E.11 String

Este tipo é usado para criar uma variável que armazena uma cadeia de caracteres que é retornada pela função `PulaLinha`(...).

```
struct STRING{
    char string[256];
}str;
```

### E.12 DH

DH é uma estrutura auxiliar que será usada para gerar uma variável que é parâmetro para a função `Set_DenavidHartenberg(int n_bodies, DH dh[8])`. Nesta variável estarão armazenados os valores que serão passados para essa função configurar os parâmetros de "Denavid Hartenberg".

```
struct DH{
     double dh0[4];
     double dh1[4];
     double dh2[4];
     double massa;
     matrix3 I;
     double ang;
     double ang_vel;
}dh[8];
```

# F PREPARAÇÃO DE UMA SIMULAÇÃO

## F.1 Estrutura de uma simulação

A estrutura básica de uma simulação pode ser vista na Figura F.1. Os dados são inseridos ou inicializados no programa principal. No interior deste há um laço que engloba a integração e a animação. A integração gera um vetor de quatérnios. Esses por sua vez dão origem à matriz de inércia que é aplicada à geometria do satélite que está armazenada em uma estrutura previamente carregada quando da inicialização do programa principal.

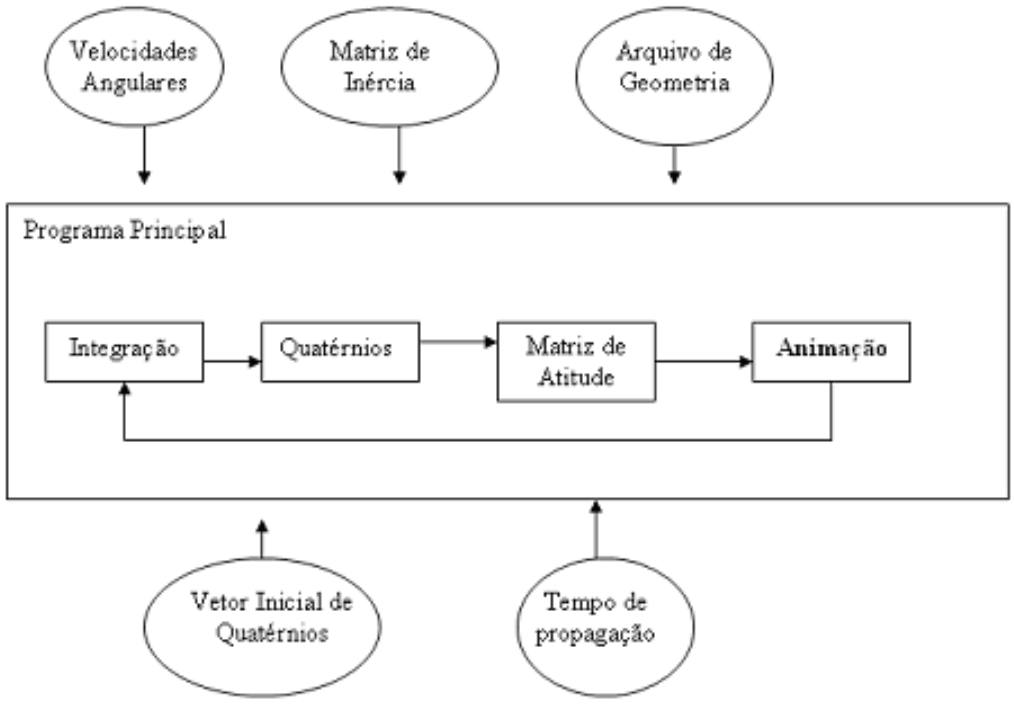


Figura F.1 - Diagrama da estrutura do programa.

## F.2 Código mínimo para apresentação de uma cena em tela.

Comenta-se a seguir o código mínimo necessário para se poder vizualizar o movimento de atitude de um corpo qualquer na tela de um computador. Este código, quando executado, gera uma janela de fundo negro. Para se fazer uma simulação deverão ser acrecentadas as funções de simulação que estão definidas em arquivos `".h"`, bastando adicionar linhas de código do tipo `#include arquivo.h` de acordo com o tipo de biblioteca que se queira. Os itens a seguir detalham as partes integrantes deste programa.

a) Cabeçalho de inclusão de arquivos. São arquivos para compatibilidade com o sistema operacional *Windows* e bibliotecas do OpenGL32, GLu32 e Glaux respectivamente.

```
# include  <windows.h>
# include  "gl.h"
# include  "glu.h"
# include  "glaux.h"
```

b) Manipuladores de contexto e de janelas de apresentação da cena. O primeiro manipulador (HDC) é particularmente importante. Antes de criar-se qualquer desenho em uma janela é necessário um dispositivo de contexto. Cada operação de desenho no sistema operacional *Windows* necessita de um dispositivo de contexto que identifique o objeto específico que está sendo desenhado. A variável `hDC` é o manipulador do dispositivo de contexto para a janela identificada pelo manipulador de janela `hWnd` (Wright e Lipchak, 2005). A variável `hRC` serve de manipulador permanente para a interpretação de contexto, já `hWnd` é um manipulador para a janela (espécie de ponteiro da janela) e a última variável `hInstance` é um ponteiro para a instância da aplicação.

```
HDC         hDC=NULL;
HGLRC       hRC=NULL;
HWND        hWnd=NULL;
HINSTANCE   hInstance;
```

c) Variáveis globais. A variável `keys` é um vetor que serve para o armazenamento de informações sobre as teclas acionadas no teclado, permitindo interação com o usuário. A variável `active` permite que o sistema operacional saiba se a janela de cena ainda é ativa ou não.

```
bool  keys[256];
bool  active = TRUE;
```

d) Declaração da função `WndProc(HWND,UINT,WPARAM,LPARAM)`. Essa função será comentada posteriormente na seqüência da codificação.

```
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
```

e) A função `ReSizeGLScene(GLsizei width, GLsizei height)` é responsável por manter a cena dentro da tela de apresentação caso o usuário resolva mudar seu tamanho, além de manter constante a razão de aspecto. Recebe como parâmetros

de entrada a largura e altura da janela de apresentação. Com essas informações a função recalcula a razão de aspecto.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
{
     if(h == 0)//previne divisão por zero
          h = 1;

     glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glViewport(0, 0, w, h);//mantém tudo dentro das dimensões da
     janela
     glMatrixMode(GL_PROJECTION);//Reinicia o sistema de coordenadas
     glLoadIdentity();
     gluPerspective(50.0f,(GLfloat)w/(GLfloat)h, 1.0, 800.0);
     glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}
```

f) A função `InitGL(...)` é uma função de inicialização. Cabe a esta função toda a inicialização de um programa baseado em *OpenGL*. É nela, por exemplo, que é especificada a cor de fundo da janela de apresentação. A iluminação artificial e a forma de cálculo da matriz também são aqui inicializados.

```
int InitGL(GLvoid)
{
     GLfloat  ambientLight[ ] = {0.3f,0.3f,0.3f,1.0f};
     GLfloat  diffuseLight[ ] = { 0.7f, 0.7f, 0.7f, 1.0f };
     GLfloat lightPos[ ] = {-5.0f, 5.0f, -5.0f, 0.0f};
     glEnable(GL_DEPTH_TEST);
     glEnable(GL_LIGHTING);
     glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
     glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
     glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
     glEnable(GL_LIGHT0);
     glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
     glClearColor(0.0f, 0.0f, 0.5f, 1.0f );
     return TRUE;
}
```

g) A função `DesenhaCena()` é responsável pela aglutinação de todas as outras rotinas e comandos relativos ao desenho da cena. Ao se colocar todas as rotinas de desenho dentro de uma única função, além de organizar-se a codificação de um programa se tem uma grande versatilidade no que diz respeito ao *OpenGL*. Este, como já mencionado, cria as imagens por intermédio de um laço. A função `DesenhaCena()` é então chamada dentro deste laço e, conseqüentemente, aciona as subrotinas de desenho. Nesta função é possível, também, aglutinar funções

secundárias de desenho como desenhar toda uma cena com comandos específicos (para comandos de desenho ver Wright e Lipchak, 2005). É por este motivo que ela aqui está definida em branco, pois cabe ao usuário organizá-la como melhor lhe convir.

```
int DesenhaCena ()
{
return TRUE;
}
```

h) A função `KillGLWindow(...)` é responsável pela "desmontagem"de todo o programa. Um programa que usa a biblioteca *OpenGL*, executada sob o Windows, cria vários processos. Esta função é responsável pelo término de cada processo e pela liberação de contexto e da desalocação de memória.

```
GLvoid KillGLWindow(GLvoid)
{
    if (hRC)
    {
        if (!wglMakeCurrent(NULL,NULL))
        {
            MessageBox(NULL, Release Of DC And RC Failed.", "SHUT-
DOWN ERROR", MB"_OK | MB_ICONINFORMATION);
        }

        if (!wglDeleteContext(hRC))
        {
            MessageBox(NULL,Release Rendering Context Failed.",
"SHUTDOWN ERROR",MB"_OK | MB_ICONINFORMATION);          }
        hRC=NULL;
        }
        if (hDC \&\& !ReleaseDC(hWnd,hDC))
        {
        MessageBox(NULL,Release Device Context"Failed., "SHUTDOWN
ERROR",MB"_OK | MB_ICONINFORMATION);
        hDC=NULL;
    }
    if (hWnd \&\& !DestroyWindow(hWnd))
    {
        MessageBox(NULL, Could Not Release hWnd.", "SHUTDOWN
ERROR",MB"_OK | MB_ICONINFORMATION);
        hWnd=NULL;
    }
    if (!UnregisterClass(OpenGL",hInstance))"
    {
        MessageBox(NULL, Could Not Unregister Class.", "ER-
```

```
ROR",MB"_OK | MB_ICONINFORMATION);
          hInstance=NULL;
     }
}
```

i) A função `CreateGLWindow(...)` cria a janela de apresentação da cena. Esta função tem como parâmetros um título (seqüência de caracteres) para a janela, a largura e altura desta janela e o número de bits que serão usados para a reprodução das cores que serão apresentadas nesta janela.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits)
{
      GLuint PixelFormat;
      WNDCLASS    wc;
      DWORD   dwExStyle;
      DWORD   dwStyle;
      RECT    WindowRect;
      WindowRect.left=(long)0;
      WindowRect.right=(long)width;
      WindowRect.top=(long)0;
      WindowRect.bottom=(long)height;

      hInstance = GetModuleHandle(NULL);
      wc.style   = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
      wc.lpfnWndProc = (WNDPROC) WndProc;
      wc.cbClsExtra = 0;
      wc.cbWndExtra = 0;
      wc.hInstance = hInstance;
      wc.hIcon = LoadIcon(NULL, IDI_WINLOGO);
      wc.hCursor = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground = NULL;
      wc.lpszMenuName = NULL;
      wc.lpszClassName  = OpenGL";"

      if (!RegisterClass(\&wc))
      {
           MessageBox(NULL, Error To Register Window Class. ", "ER-
ROR",MB"_OK|MB_ICONEXCLAMATION);
          return FALSE;
      }

      dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
      dwStyle=WS_OVERLAPPEDWINDOW;
      AdjustWindowRectEx(\&WindowRect, dwStyle, FALSE, dwExStyle);

//Cria a Janela
if (!(hWnd=CreateWindowEx(dwExStyle,
```

```
          OpenGL", //Nome da Calse"
          title,  //Título da Janela
          dwStyle |   //Define estilo da janela
          WS_CLIPSIBLINGS |   WS_CLIPCHILDREN,
          0, 0,       //Posição da Janela
           WindowRect.right-WindowRect.left,//Calc. Largura Janela
          WindowRect.bottom-WindowRect.top,//Calc. Altura Janela
          NULL,
          NULL,
          hInstance,
          NULL)))
{
     KillGLWindow();
     MessageBox(NULL,Error.", "ERROR",MB"_OK|MB_ICONEXCLAMATION);
     return FALSE;
}

Static PIXELFORMATDESCRIPTOR pfd=
{
     sizeof(PIXELFORMATDESCRIPTOR),
     1,
     PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
     PFD_DOUBLEBUFFER, PFD_TYPE_RGBA,
     bits,
     0, 0, 0, 0, 0, 0,
     0,
     0,
     0,
     0, 0, 0, 0,
     16,
     0,
     0,
     PFD_MAIN_PLANE,
     0,
     0, 0, 0
};

if (!(hDC=GetDC(hWnd))) {
     KillGLWindow();
MessageBox(NULL,Can'tCreateAGLDeviceContext.","ERROR",       MB"_-
OK|MB_ICONEXCLAMATION);
return FALSE;
}

if (!(PixelFormat=ChoosePixelFormat(hDC,\&pfd)))
{
     KillGLWindow();
```

```
MessageBox(NULL,Can'tFindASuitablePixelFormat.","ERROR", MB"_-
OK|MB_ICONEXCLAMATION);
     return FALSE;
}
if(!SetPixelFormat(hDC,PixelFormat,\&pfd))
{
     KillGLWindow();
MessageBox(NULL,Can'tSetThePixelFormat.","ERROR", MB"_OK|MB_-
ICONEXCLAMATION);
     return FALSE;
}

if (!(hRC=wglCreateContext(hDC)))
{
     KillGLWindow();
MessageBox(NULL,Can'tCreateAGLRenderingContext.","ERROR", MB"_-
OK|MB_ICONEXCLAMATION);
     return FALSE;
}

if(!wglMakeCurrent(hDC,hRC))
{
     KillGLWindow();
MessageBox(NULL,Can'tActivateTheGLRenderingContext.","ERROR",
MB"_OK|MB_ICONEXCLAMATION);
return FALSE;
}

ShowWindow(hWnd,SW_SHOW);
SetForegroundWindow(hWnd);
SetFocus(hWnd);
ReSizeGLScene(width, height);

if (!InitGL())
{
     KillGLWindow();
MessageBox(NULL,InitializationFailed.","ERROR", MB"_OK|MB_-
ICONEXCLAMATION);
     return FALSE;
}
return TRUE;
```

j) A função `WndProc`(...) é exclusiva para o sistema operacional *Windows*. Essa função gerencia todas as mensagens enviadas pelo sistema operacional para esse programa.

```
LRESULT CALLBACK WndProc(   HWND hWnd,
                            UINT    uMsg,
                            WPARAM  wParam,
                            LPARAM  lParam)
{
  switch (uMsg)
  {
    case WM\_ACTIVATE:{
         if((LOWORD(wParam) != WAvINACTIVE)\&\&
         !((BOOL) HIWORD(wParam)))
              active=TRUE;
         else
              active=FALSE;
         return 0;
  }
    case WM_SYSCOMMAND: {
         switch (wParam)
         {
              case SC_SCREENSAVE:
              case SC_MONITORPOWER:
              return 0;
         }
         break;
    }
    case WM_CLOSE:{
         PostQuitMessage(0);
         return 0;
    }
    case WM_KEYDOWN:{
         keys[wParam] = TRUE;
         return 0;
    }
    case WM_KEYUP:{
         keys[wParam] = FALSE;
         return 0;
    }
    case WM_SIZE:{
         ReSizeGLScene(LOWORD(lParam),HIWORD(lParam));
         return 0;
    }
  }
  return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

l) A codificação que segue pode ser chamada de programa principal. É a partir deste código que todos os outros códigos mencionados anteriormente serão executados. Esta codificação detém um laço (laço do tipo *while*) responsável pela apresentação em seqüência de cada *frame* para a formação da imagem no *OpenGL* (para maiores informações ver Wright e Lipchak, 2005). No presente trabalho ele comporta, também, o laço de integração para obter-se a propagação da atitude.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
     MSG msg;
     if(!CreateGLWindow(Visualização da Atitude",1000,800,16))"
          return 0;

     //Todas inicializações devem ser feitas aqui.
     double t0 = timeGetTime(); //inicio da simulação
     select_integ_method(2,0);//runge kuta 7-8

     // Parâmetros do integrador
     int     neqn = 7;
     double  relerr = 1.0e-6;
     double  abserr = 1.0e-6;
     double  t = 0;//inicio do intervalo de integracao
     double  tout = 0.01;//fim do intervalo de integracao
     double  dtint = 0.01;// Passo interno

     set_integ_precision(relerr,abserr);
     set_step_time(dtint);
     set_int_step_time(dtint);

     //*******************************
     while(t < T_FINAL){
     //Loop de integração e visualização da atitude
          if(PeekMessage(\&msg,NULL,0,0,PM_REMOVE)){
               if (msg.message == WM_QUIT)
                    break;
               else{
                    TranslateMessage(\&msg);
                    DispatchMessage(\&msg);
               }
          }
          else{
               sat_propagation();  //Propagação da atitude
```

```
                t = tout;
                if(keys[VK_ESCAPE]) //Se esc, sai do programa
                    break;   //Foi teclado ESC
                else{
                    DesenhaCena();  //Aqui ocorre a visualização
                    SwapBuffers(hDC);
                    //Swap Buffers (Double Buffering)
                }
                double t1 = timeGetTime();
                tout += (t1 - t0)*0.001;
                //imcrementa intervalo de tempo (tn para tn+1)
                dtint = (t1 - t0)*0.001;
                set_step_time(dtint);
                set_int_step_time(dtint);
                t0 = t1;
            }
        \}
        KillGLWindow();
        return(0);
}
```

# G EXEMPLO DA MONTAGEM DE UMA SIMULAÇÃO

## G.1 Inicialização das variáveis e estruturas de uma simulação

A inicialização de variáveis segue dois caminhos. O primeiro é uma simples inicialização de uma variável global já no cabeçalho do programa. Essa forma de inicialização não tem maiores dificuldades e é mencionada em Kernighan e Ritchie (1988).

A segunda forma de inicializção usada nesse trabalho é a que ocorre dentro da função `int WINAPI WinMain`(...). Essa é a principal função do programa, podendo também ser chamada de programa pricipal. Dentro dessa função, por exemplo, são executados os laços de animação e de integração numérica e é criada a janela de apresentação da cena animada.

## G.2 Seqüência de inicialização de funções

Para a simulação funcionar é necessária um determinada ordem na chamada de funções e suas inicializações. Segue um exemplo comentado de como deve ser essa ordem.

a) A chamada da função de criação da janela de visualização permite que seja criada a referida janela. Todas as cenas que forem criadas a partir desse momento aparecerão dentro dessa janela.

```
if(!CreateGLWindow("Visualização da Atitude",1000,800,16))
     return 0;
```

b) A inicialização da câmera (ver Apêndice D para a função `Camera(mov)`) e das respectivas variáveis associadas faz com que a cena seja apresentada em perspectiva. Dessa forma todos os três eixos coordenados são visíveis. Se os valores abaixo fossem todos nulos a câmera seria posicionada sobre o eixo $Z$.

```
mov.xRot = 105.0f;  mov.yRot = 5.0f;  mov.zRot = -15.0f;
mov.xTrans = 0.0f; mov.yTrans = 0.0f; mov.zTrans = -15.0f;
Camera(mov);
```

c) A seguir efetua-se a inicialização das velocidades angulares e da matriz de inércia do satélite sem os apêndices, pois esses terão as suas próprias matrizes de inércia inicializadas posteriormente. As duas funções "`set`..."permitem que a biblioteca de funções armazene os valores das velocidades e da matriz (Carrara e Hassmann, 2007).

```
vector3 w; //velocidade angular do satelite
em suas coordenadas-lapis
w._1 = 0;   w._2 = 0.4; w._3 = 0.5;
matrix3 sat_in = {20, 0, 0,  0, 20, 0, 0, 0  ,5};
```

```
set_ang_velocity(w);
set_sat_inertia(sat_in);
```

d) As próximas linhas são referentes a integração numérica. A função `select...` seleciona o método de integração desejado (Carrara e Hassmann, 2007). Subseqüentemente tem-se variáveis que serão os parâmetros de entrada das funções `set...` . Essas funções preparam o integrador que será executado dentro do laço de integração e visualização.

```
select_integ_method(2,0);   //runge kuta 7-8
double    relerr = 1.0e-6;
double    abserr = 1.0e-6;
double    t = 0;  //inicio do intervalo de integracao
double    tout = 0.01; //fim do intervalo de integracao
double    dtint = 0.01; // Passo interno
set_integ_precision(relerr,abserr);
set_step_time(dtint);
set_int_step_time(dtint);
```

e) A partir deste momento são feitas as inicializações de apêndices e do corpo principal. Inicialmente são organizados os parâmetros dos apêndices. Esses parâmetros são relacionados na estrutura `struct DH` (ver Apêndice E) conforme as regras citadas em (Carrara e Hassmann, 2007). No exemplo abaixo está sendo preparado um único apêndice, mas podem ser configurados até oito apêndices, sendo a geometria do satélite associada a esses parâmetros. A função `Set_DenavidHartenberg(0,dh)` permite que os parâmetros sejam reconhecidos automaticamente (ver Apêndice D).

```
//parametros dos apendices
dh[0].dh0[0] = 90;   dh[0].dh1[0] = 0;   dh[0].dh2[0] = -90;
dh[0].dh0[1] = 0;    dh[0].dh1[1] = 0;   dh[0].dh2[1] = 0;
dh[0].dh0[2] = 0;    dh[0].dh1[2] = 0;   dh[0].dh2[2] = 0;
dh[0].dh0[3] = -90;  dh[0].dh1[3] = 90;  dh[0].dh2[3] = 0;



dh[0].I._1._1 = 1; dh[0].I._1._2 = 0; dh[0].I._1._3 = 0;
dh[0].I._2._1 = 0; dh[0].I._2._2 = 1; dh[0].I._2._3 = 0;
dh[0].I._3._1 = 0; dh[0].I._3._2 = 0;  dh[0].I._3._3 = 1;
dh[0].massa = 100;
dh[0].ang = 0;
dh[0].ang_vel = 1;
Set_DenavidHartenberg(1,dh);
```

f) As duas linhas subseqüentes calculam um quatérnio inicial e a função `set_attitude`(...) inicializa o valor calculado na biblioteca de funções (Carrara e Hassmann, 2007). O presente exemplo é restrito ao caso de nutação.

```
quaternion x_q = Quat_Inicial(get_sat_inertia(),
get_ang_velocity());
set_attitude(x_q);
```

g) Neste momento já é possível fazer a leitura das geometrias. São carregadas em uma estrutura (ver Apêndice E) tanto as geometrias do corpo principal quanto as geometrias dos apêndices. Neste exemplo está sendo carregada a geometria de um satélite e de um apêndice. As geometrias são do tipo AC3D, mas poderiam ser também do tipo 3DS.

```
Load_AC3D("vert.txt",0,1); //satelite
Load_AC3D("apend.txt",1,1); //apendice 1
```

Deste ponto em diante começa o laço computacional de integração numérica e visualização do programa principal. É dentro deste laço que se encontra a chamada da função `int DesenhaCena`() que será explanada no próximo item.

### G.3 Chamadas das funções para a animação

Essas chamadas ocorrem dentro da função `DesenhaCena`(), que, por sua vez, é chamada dentro do laço comum de integração e animação que está inserido no progama principal. Todas as funções estão descritas no Apêndice D.

a) Chama-se inicialmente a função `ClearReset`(). Ela garante que não haverá sobreposição de imagens entre um quadro e outro (descrição no Apêndice D).

```
ClearReset(); //Inicialização de frame
```

b) As duas funções que seguem são necessárias para garantir a correta iluminação da imagem. Conforme a montagem dos polígonos que compõem os objetos, a normal às faces poderá ser considerada "para dentro" ou "para fora" do objeto. Dependendo dos parâmetros dessas funções pode-se selecionar a normal num sentido (`GL_CW`) ou no outro (`GL_CCW`), fazendo com que a cena seja corretamente visualizada pelo usuário (ver Wright e Lipchak, 2005 para os possíveis parâmetros dessas funções). Deve-se, porém, tomar o cuidado de se observar o comportamento da cena com cada chamada dessas funções. Dependendo da forma que o arquivo de geometrias foi criado é possível que a cada criação de um novo objeto deva-se alterar os parâmetros dessas funções.

```
glFrontFace(GL_CCW);
glDisable(GL_CULL_FACE);
```

c) De agora em diante começam os desenhos propriamente ditos. As duas funções abaixo desenham um grupo de eixos coordenados e o cone de nutação do espaço. Esses objetos não apresentam movimento, pois são referências fixas. Para objetos como esses serem animados é necessário antes fazer a carga da matriz de atitude respectiva. Deve-se lembrar também que são objetos opcionais e não têm influência na apresentação da atitude de um satélite. O cone que é criado pela função abaixo, por exemplo, só tem sentido em um caso de nutação.

```
EixosCoordenados(5.0,4.0); //eixos inerciais
Cones(3,1,AnguloCone(get_sat_inertia(),
get_ang_velocity()));// cone espaco
```

d) Efetua-se a seguir a chamada da função `MatrizAtitude`(...) e `DesenhaCorpo`(...) com índice zero que simboliza a reprodução do corpo principal. As funções `EixosCoordenados`(...) e `Cones`(...) são opcionais e vão depender da vontade do usuário. As funções `Cones`(...) só fazem sentido em casos de satélites com movimento de nutação. No caso de dispositivos com apêndices essas funções não devem ser chamadas, pois não representam o movimento.

```
vector3 vec = {0,0,0};
MatrizAtitude(quatrmx(get_attitude()),vec);
EixosCoordenados(5.0,2.0);
Cones((float)3.48,2,AnguloCone(get_sat_inertia(),
get_ang_velocity()));
```

e) O desenho dos corpos (tanto principal como apêndices que terão movimento) deve ser feito após uma chamada da função `MatrizAtitude`(...) que atribui a matriz de atitude do satélite. Do contrário não há movimento algum. Sabe-se que o corpo que será desenhado é o principal, pois o parâmetro da função `DesenhaCorpo`(...) é nulo.

```
DesenhaCorpo_AC3D(cps,0,1);
```

f) A animação gerada até este ponto é de um corpo sem seus apêndices. Para a colocação de apêndices na simulação deve-se, antes de mais nada, manipular a pilha de memória. Para haver uma animação correta é necessário multiplicar-se a matriz de atitude do satélite pela matriz que rege o apêndice. Chamando-se a função `glPushMatrix`() (Wright e Lipchak, 2005) a matriz de atitude do satélite é armazenada na pilha corrente de matrizes e pode ser chamada posteriormente. Esta matriz, porém, ainda é a corrente. Quando da chamada da

função `MatrizAtitude`(...) há a multiplicação desta matriz corrente pela matriz do apêndice e a animação ocorre corretamente, ou seja, o apêndice em questão terá sempre referência ao corpo principal. Posteriormente se faz o desenho do apêndice (`DesenhaCorpo_AC3D`(...)). Para garantir que qualquer outro apêndice tenha relação com o corpo principal faz-se a chamada de `glPopMatrix`() para recuperar a matriz previamente armazenada com `glPuschMatrix`(). Qualquer chamada de um apêndice a mais segue a mesma seqüência.

```
glPushMatrix();
MatrizAtitude(transpose(get_body_rmatrix(1)),
get_body_ovect(1));
DesenhaCorpo_AC3D(cps,1,-1);
glPopMatrix();
```

## G.4 Translações de cena

Nem sempre a geometria do satélite e de seus apêndices estão desenhadas de forma confortável à simulação em seus arquivos de origem (tipos AC3D ou 3DS). Muitas vezes o projetista define os objetos afastados da origem por algum motivo ou outro.

As funções de simulação presentes nesse trabalho levam em consideração, porém, a origem no centro de tela. Para que os desenhos sejam corretamente reproduzidos podem ser necessárias translações.

O *OpenGl* aferece essa facilidade ao usuário por intermédio da função `glTranslatef`(...) que faz a translação de qualquer imagem nos eixos $x$, $y$ e $z$ (Wright e Lipchak, 2005). Para que o corpo seja translacionado basta chamar-se a função antes de desenhar o corpo. No exemplo abaixo o corpo será transladado uma unidade em $x$ e duas em $y$.

```
glTranslatef(1,2,0);
DesenhaCorpo_AC3D(cps,1,-1);
```

## G.5 Parâmetros de Denavid Hartemberg

Esses parâmetros são os responsáveis pelo posicionamento e movimentação dos apêndices de satélite na simulação (Carrara e Hassmann, 2007). Para que esses parâmetros sejam carregados automaticamente foi criada a função `Set_DenavidHartenberg`(...). Esta função chama as subrotinas `set_number_bodies`(...), `set_body_k`(...) e `set_body_pos`(...) (Carrara e Hassmann, 2007) que são fundamentais para que os apêndices "existam" na simulação. Se essas funções não forem chamadas não há entendimento entre as rotinas da simulação, criando uma inconsistência, pois para que as respectivas matrizes de atitude sejam criadas é necessário que os apêndices "existam" antes de mais nada.

A função `set_number_bodies`(...) fixa o número de apêndices que deverão existir na simulação. As funções `set_body_k`(...) e `set_body_pos`(...) indicam as características físicas de cada apêndice e a posição e movimento que cada um terá, respectivamente.

## PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

**Teses e Dissertações (TDI)**

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

**Notas Técnico-Científicas (NTC)**

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programas de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

**Propostas e Relatórios de Projetos (PRP)**

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

**Publicações Seriadas**

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

**Pré-publicações (PRE)**

Todos os artigos publicados em periódicos, anais e como capítulos de livros.

**Manuais Técnicos (MAN)**

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

**Relatórios de Pesquisa (RPQ)**

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

**Publicações Didáticas (PUD)**

Incluem apostilas, notas de aula e manuais didáticos.

**Programas de Computador (PDC)**

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. Aceitam-se tanto programas fonte quanto os executáveis.