DEPTO. DE ENG. DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Relatório Técnico
DCA - 003/97

Uma Linguagem para Desenvolvimento de Roteiros de Animação

Alberto Barbosa Raposo
Léo Pini Magalhães

Universidade Estadual de Campinas (UNICAMP)
Faculdade de Engenharia Elétrica e de Computação (FEEC)
Depto. de Engenharia de Computação e Automação Industrial (DCA)
C.P. 6101 - 13083-970 - Campinas, SP, Brazil
Phone: +55 - 19 - 239-8385 - Fax: +55 - 19 - 239-1395
alberto, leopini@dca.fee.unicamp.br

Março 1997

ÍNDICE

RESUMO

Este relatório tem como objetivo apresentar a linguagem para desenvolvimento de roteiros de animação no TOOKIMA 2.0.

O uso de roteiros constitui um método prático e usual para automatizar o processo de animação. Um roteiro é uma sequência ordenada de comandos, interpretada de maneira não ambígua pelo sistema de animação, e que deve ser passada ao computador para que ele opere sobre os elementos que compõem a cena.

A linguagem proposta pretende ser intermediária entre a complexa linguagem do TOOKIMA (orientada ao computador e muito próxima da linguagem C) e a interface gráfica (cujo objetivo é permitir a construção interativa do roteiro na nova linguagem).

Essa nova sintaxe tem como objetivo permitir o desenvolvimento de roteiros em um nível mais alto de comandos (que se aproxime mais daqueles desenvolvidos pelos animadores profissionais). Isso eliminará a necessidade do conhecimento de comandos de baixo nível, oriundos da linguagem C, que existem na linguagem do TOOKIMA (que, na verdade, é uma extensão da linguagem C). Embora simples, a nova linguagem de roteiros é abrangente a ponto de possibilitar a utilização de todos os recursos importantes já existentes no TOOKIMA.

I - INTRODUÇÃO

I.1 - ProSIm - Um Sistema para Prototipação e Síntese de Imagens Foto-Realistas e Animação:

O ProSIm (/MAGA-91/) é um projeto bastante abrangente, para a implementação de diversos módulos/sistemas relacionados à Computação de Imagens. Estes módulos podem ser divididos em: Modelagem, Rendering, Interface e Animação. Esses módulos são independentes entre si, mas interdependentes de um sistema de funções básicas (o ProSIm), de forma a facilitar e incentivar a integração dos mesmos.

Em seu estágio atual, o ProSIm abrange basicamente 4 partes:

a) *Sistema de Modelagem*: Este sistema é responsável pela criação/modelagem de objetos, considerando suas características geométricas e topológicas (informações de relação entre as partes de um objeto). O sistema permite também uma pré-visualização dos objetos (visualização rápida e simplificada). É composto de três partes principais: um editor de primitivas (define objetos com superfícies poligonalizadas e com descrição CSG - Construtive Solid Geometry); uma câmera sintética (para visualização tridimensional da imagem gerada pelos outros módulos) e um compositor de imagens (posiciona, escala e orienta os objetos em coordenadas do mundo real, além de realizar as operações CSG entre objetos e as transformações nos mesmos; também exerce controle sobre a operação dos outros módulos).

b) *Sistema de Rendering* (visualização): Após a definição da geometria da cena e o acréscimo dos parâmetros para sua visualização final (fontes de luz, grau de transparência do material, etc), deve-se fazer o processamento final da imagem, através de um dos algoritmos de rendering existentes (Ray-Tracing, Scanline e Radiosidade).

c) *Interface* gráfica para modelagem geométrica /MALH-94/ e para auxílio na construção do script de animação /RAPO-96/.

d) *Animação*: É nesta parte do ProSIm que este trabalho se concentrará. O TOOKIMA é o suporte à abordagem para a animação cinemática direta.

I.2 - TOOKIMA - TOOl KIt for scripting computer Modeled Animation:

O TOOKIMA (/SILV-92/, /RAPO-96/, /RAPO-97/) define um conjunto de ferramentas para descrição algorítmica de animações de objetos modelados por computador. O TOOKIMA foi implantado no DCA-FEEC-UNICAMP.

O TOOKIMA se encaixa no Sistema de Animação do ProSIm. Um sistema de animação por computador é normalmente dividido em subsistemas:

- O subsistema de modelagem de objetos utilizado pelo TOOKIMA atualmente é o Geomod, um modelador geométrico de objetos por fronteiras, para o qual foi desenvolvida uma interface /MALH-94/.
- O subsistema de visualização é o SIPP /YNGV-94/, que é uma biblioteca para a renderização de cenas tridimensionais, usando um algoritmo de *scanline z-buffer* com muitos recursos (*wireframe*, mapeamento de texturas, *anti-aliasing*, sombras, etc). Também pretende-se usar o POV-Ray (persistence of Vision Ray tracer) para a visualização final das cenas /YOUN-96/.
- O subsistema de descrição e controle de animação é o próprio TOOKIMA, responsável pela integração dos dois subsistemas anteriores, além de prover uma linguagem para sincronização temporal e descrição das diversas cenas que comporão a animação.

II - A NOVA LINGUAGEM DE ROTEIROS

O uso de roteiros constitui um método prático e usual para automatizar o processo de animação. Um roteiro é uma sequência ordenada de comandos, interpretada de maneira não ambígua pelo sistema de animação, e que deve ser passada ao computador para que ele opere sobre os elementos que compõem a cena.

A nova linguagem proposta pretende ser intermediária entre a complexa linguagem do TOOKIMA /SILV-92/, /RAPO-93/, /RAPO-97/ (orientada ao computador e muito próxima da linguagem C) e a interface /RAPO-96/ (cujo objetivo é permitir a construção interativa do roteiro na nova "linguagem").

Essa nova sintaxe tem como objetivo permitir o desenvolvimento de roteiros em um nível mais alto de comandos (que se aproxime mais daqueles desenvolvidos pelos animadores profissionais). Isso elimina a necessidade do conhecimento de comandos de baixo nível, oriundos da linguagem C, que existem na linguagem do TOOKIMA (que, na verdade, é uma extensão da linguagem C). Embora simples, a nova linguagem de roteiros é abrangente a ponto de possibilitar a utilização de todos os recursos importantes já existentes no TOOKIMA.

Foi implementado um "interpretador" para o novo roteiro. Este “interpretador” traduz o roteiro para a linguagem do TOOKIMA, sendo possível a sua execução (o usuário comum não precisa ter conhecimento da linguagem do TOOKIMA e nem da linguagem C).

O sistema de animação, atualmente, possui, dentre outros, os seguintes recursos: pré-visualização em *wireframe* (onde apenas as arestas dos objetos são visíveis, sem a preocupação com texturas, iluminação, eliminação de superfícies escondidas, etc) e codificação dos quadros de uma animação no formato MPEG (Motion Picture Expert Group). Este formato é bastante compacto, embora apresente perdas na qualidade da imagem (além da animação MPEG, os quadros sem perdas continuam sendo gerados, para o caso de se desejar editá-los numa animação sem perdas). A codificação para o formato MPEG é feita usando o software “Berkeley MPEG-1 Video Encoder” /GONG-94/, de maneira transparente ao usuário (o usuário apenas escolhe a opção MPEG no TOOKIMA, e terá como resultado a animação e os quadros - sem perdas - separados da mesma; o TOOKIMA usa o MPEG-encoder “internamente”).

O sistema de animação do ProSIm permite três tipos de usuários: o usuário leigo (que utilizará a interface), o usuário mediano (que utilizará a interface, mas com conhecimento da linguagem de roteiro) e o usuário "especialista" (que conhecerá a linguagem do TOOKIMA e a linguagem C, podendo dispor da maior flexibilidade que uma linguagem de mais baixo nível permite).

A figura 1 sintetiza a estrutura do sistema de animação proposto para o ProSIm.

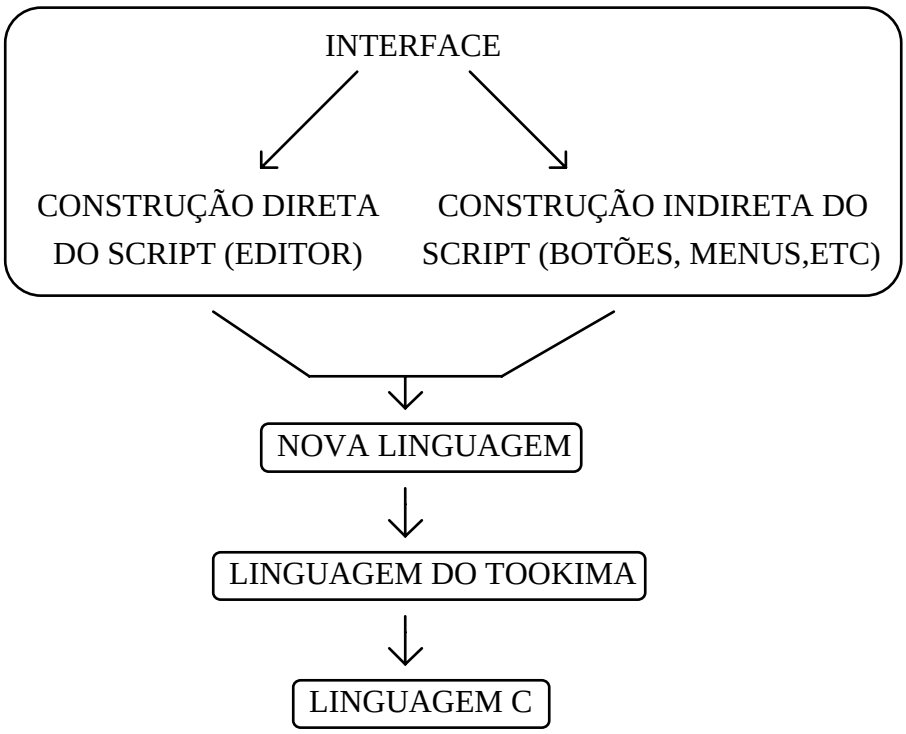

Figura 1: Estrutura do sistema de animação do ProSIm.

O roteiro proposto tem 8 módulos, que especificarão os parâmetros e funções referentes ao objeto do módulo. Todo módulo deve começar com o seu nome (`ACTORS`, `GROUPS`, etc) e terminar com a palavra `END`. Os módulos podem aparecer em qualquer ordem no roteiro (desde que o `GENERAL` seja o primeiro), ou até mesmo não aparecer (alguns módulos não são obrigatórios). Dentro de cada módulo, pode ser necessário manter uma ordem na descrição dos parâmetros (este relatório alertará, quando isso for necessário). Cada um desses módulos será detalhado a seguir.

O sinal " ; " indica que o resto da linha do roteiro será considerada como comentário.

1) GENERAL:

Neste módulo serão colocados os parâmetros gerais da animação: TotalTime (tempo total da animação, em segundos) e Rate (taxa de quadros por segundo). O número de quadros gerados será igual a Rate * TotalTime. É obrigatória a presença deste módulo (e dos parâmetros Rate e TotalTime) no início do roteiro de animação, caso contrário, o programa enviará uma mensagem de erro.

Se houver a necessidade de variáveis globais para a animação, elas deverão ser colocadas aqui.

O fragmento de roteiro a seguir mostra o módulo *general*:

```
GENERAL
     TOTALTIME = 10
     FR_RATE = 20
     double x1 = 7.5
     double x2 = read_file("/home/usr/input.in")
END
```

A animação do fragmento anterior terá 200 quadros (10 segundos de animação, a uma taxa de 20 quadros/s).

A variável `x1` é do tipo `double` e assume o valor 7.5. Já a variável `x2` será lida do arquivo `/home/usr/input.in` (a cada frame se lerá um valor do arquivo, que será armazenado nesta variável). A variável global pode também ser de tipos definidos pelo ProSIm: `Point`, `Vector` ou `Color` (que são estruturas com três valores reais, determinando as três coordenadas/componentes dos mesmos). A utilidade da variável lida em arquivo, é que este pode, por exemplo, conter o resultado numérico de uma simulação, cujos valores podem ser usados como parâmetros de movimentos de atores.

Uma variável global pode assumir ainda o valor dado pela função `following_track`, usada quando se deseja seguir uma trilha de pontos. A explicação da utilização de trilhas será dada no módulo *tracks*, que trata exclusivamente de trilhas.

2) ACTORS:

Este módulo define os objetos geométricos que participarão da cena e suas características. Deve ser dado o nome do arquivo com suas características geométricas (proveniente do modelador geométrico) e sua posição inicial, além de sua cor (superfície) e de seu método de tonalização (*shading*).

Este módulo também trata dos movimentos dos atores. Cada movimento (translação, rotação e escalamento) deve vir acompanhado dos parâmetros que o caracterizam (quantos graus o ator gira, qual a porcentagem do escalamento, etc), além do intervalo de tempo em que ocorrerão. A medida de tempo pode ser feita em segundos ou em quadros (*frames*). A conversão de *frames* para segundos é feita da seguinte maneira:

$$FRAME(i) = (i \,/\, Rate) \text{ segundos}$$

Quando se deseja especificar o final da animação, pode-se utilizar `END`, ao invés do número do último quadro ou do instante final da animação.

A definição das características "físicas" de cada ator e sua posição inicial podem ser dadas num arquivo em separado, que tem a seguinte forma:

```
BRP = "/proj/prosim/objects/esf.brp"
SHADE_TYPE = WOOD
SHADOW = FALSE
COLOR = "/home/usr/color/madeira1.cor"
INITIAL_POSITION = 0., 3., -5.
```

Os parâmetros desse arquivo podem vir em qualquer ordem. O parâmetro `BRP` define o arquivo B-rep, com as características geométricas do ator (este arquivo é gerado pelo modelador geométrico). `SHADE_TYPE` determina o tipo da textura do ator, pelo SIPP. Pode ser: `WOOD`, `MARBLE`, `GRANITE`, `STRAUSS` (superfícies metálicas), `PHONGS` ou `BASIC`. Esse parâmetro deve ser coerente com o arquivo de cor, definido por `COLOR` (por exemplo, se `SHADE_TYPE` for `MARBLE`, o arquivo de cor deve definir uma textura de mármore). Os parâmetros `BRP`, `SHADE_TYPE` e `COLOR` devem sempre estar presentes no arquivo de definição do ator. O parâmetro `SHADOW` indica se o ator terá ou não sombra (o SIPP apenas permite uma determinação de que todos os atores terão ou não sombra, não permitindo que certos atores tenham sombras e outros não; este parâmetro é usado apenas para facilitar a portabilidade com outros renderizadores). O parâmetro `INITIAL_POSITION` determina a posição inicial do objeto; se este parâmetro não estiver presente no arquivo, o programa assumirá que a posição inicial do ator será aquela na qual ele foi modelado (a que existe no arquivo .brp).

Com relação ao arquivo de cores (na verdade, texturas) para o SIPP, ele é apenas um arquivo numérico, com a primeira linha indicando o tipo de textura (1 para `BASIC`, 2 para `PHONG`, 3 para `STRAUSS`, 4 para `MARBLE`, 5 para `GRANITE` e 6 para `WOOD`). As linhas seguintes variarão, de acordo com o tipo de textura, como visto nos exemplos a seguir:

a)
```
1                /* BASIC */
.5 .65 .789      /* componentes rgb normalizados da cor */
1. 0.6  .99      /* opacidade para cada componente: 0 significa
                    transparente e 1 opaco */
0.7              /* ambiente: indica quanto da cor da superfície será
                    visível quando ela não está iluminada */
0.5              /* especular: quanto da luz incidente será refletido */
0.45             /* C3: quão brilhante a superfície é. 0 indica que é
                    muito brilhante; 1, nada brilhante */
```

b)
```
2                /* PHONGS */
.5 1. 0.76       /* cor */
1. 1. 1.         /* opacidade */
0.7              /* ambiente */
0.8              /* especular */
0.5              /* difuso: quanto da luz incidente será refletida
                    difusamente pela superfície */
35               /* varia de 1 a 200 (totalmente brilhante) */
```

c)
```
3                /* STRAUSS */
.5 .75 1.        /* cor */
1. 0.9 1.        /* opacidade */
0.654            /* ambiente */
0.75             /* smoothness: 1 significa superfície lisa e
```

```
                    brilhante */
    .87             /* metalness: 0 significa não metálico; 1,
                       completamente metálico */
```

d)
```
    4               /* MARBLE */
    .5 .7 .3        /* cor base */
    .9 .8 .65       /* cor das "estrias" do mármore */
    1. 1. 1.        /* opacidade */
    .8              /* ambiente */
    .5              /* especular */
    .65             /* C3 */
    3.              /* escala: um valor alto torna o padrão do mármore
                       mais "compacto" */
```

Obs.:

- Os arquivos para granito e madeira são idênticos ao do mármore, apenas trocando a primeira linha para o valor apropriado.
- Os comentários não são permitidos no arquivo de cores; estão aqui apenas para efeito didático.

Uma alternativa à definição do ator como um arquivo B-rep é defini-lo como uma sequência numerada de arquivos .brp, sendo que, a cada quadro, apenas um será mostrado. Para isso, ao invés de `BRP`, deve-se usar `GEO_MORPH` da seguinte maneira, no arquivo de definição do ator:

```
GEO_MORPH = "/proj/prosim/brp/esf", 10, FRAME(8)
```

O primeiro parâmetro indica o nome dos B-reps numerados (no exemplo, a sequência será `/proj/prosim/brp/esf000.brp`, `/proj/prosim/brp/esf001.brp`, ...). Note que o `".brp"` não é escrito. O segundo parâmetro indica o tamanho da sequência numerada (no exemplo, a sequência irá de 0 a 9). O terceiro parâmetro determina em que quadro a sequência começará a ser vista na animação (no exemplo, no quadro 8 será visto a `esf000.brp`, no quadro 9, a `esf001.brp`, e assim por diante, até o final da sequência).

É possível também criar uma sequência de texturas para um ator, de modo que em cada frame ele venha a ter uma textura numerada. Para isso, ao invés de `COLOR`, deve-se usar `CLR_MORPH` na definição da cor do ator.

```
CLR_MORPH = "/tmp/cor/texture", 20, FRAME(5)
```

O primeiro parâmetro indica o nome dos arquivos de cores numerados (no exemplo, a sequência será `/tmp/cor/texture000.cor`, `/tmp/cor/texture001.cor`, ...). Note que o `".cor"` não é escrito. O segundo parâmetro indica o tamanho da sequência numerada (no exemplo, a sequência irá de 0 a 19). O terceiro parâmetro determina em que quadro a sequência começará a ser vista na animação (no exemplo, no quadro 5 o ator terá a textura definida por `texture000.cor`, no quadro 6, a `texture001.cor`, e assim por diante).

Portanto, tem-se um arquivo de definição de atores, externo ao roteiro de animação, que referencia um arquivo de cores e um B-rep (ou sequência numerada de B-reps). O arquivo de definição será referenciado para cada ator da seguinte maneira no roteiro, dentro do módulo *actors*:

```
ACTORS
     A0: DEFINITION("def/actor0.adef")
     A1: DEFINITION("def/actor1.adef")
     A2: DEFINITION("def/actor1.adef")
END
```

No exemplo anterior, cada arquivo .adef é um arquivo de definição, como visto anteriormente. Como se pode perceber pelo exemplo, um mesmo arquivo de definição pode servir para dois atores (serão 2 atores iguais). Os atores devem ser chamados de `A0`, `A1`, ...

Obs.:

- Se o arquivo de definição de um ator se encontrar em um diretório diferente daquele onde se encontra o arquivo-roteiro, deve-se ter os "paths" do arquivo de cor e do brp relativos ao diretório onde se encontra o arquivo-roteiro, e não com relação ao diretório do arquivo de definição. Por exemplo, se o roteiro estiver no diretório `pai`, o arquivo de definição no diretório `filho` e o .brp no `neto`, o arquivo de definição deve ter: `BRP = filho/neto/a1.brp`, e não apenas `neto/a1.brp`.

Qualquer um dos parâmetros do arquivo de definição pode ser redefinido no roteiro, se colocado depois do `DEFINITION` do ator. Por exemplo, é possível colocar os dois atores iguais do exemplo anterior em posições iniciais diferentes e dar ao segundo uma outra cor:

```
ACTORS
     A0: DEFINITION("def/actor0.adef")
     A1: DEFINITION("def/actor1.adef")
     A2: DEFINITION("def/actor1.adef")
         INITIAL_POSITION = 5., 5., 5.
         COLOR = "/home/usr/color/madeira2.cor"
END
```

A figura 2 a seguir ilustra o relacionamento entre os arquivos usados neste módulo:

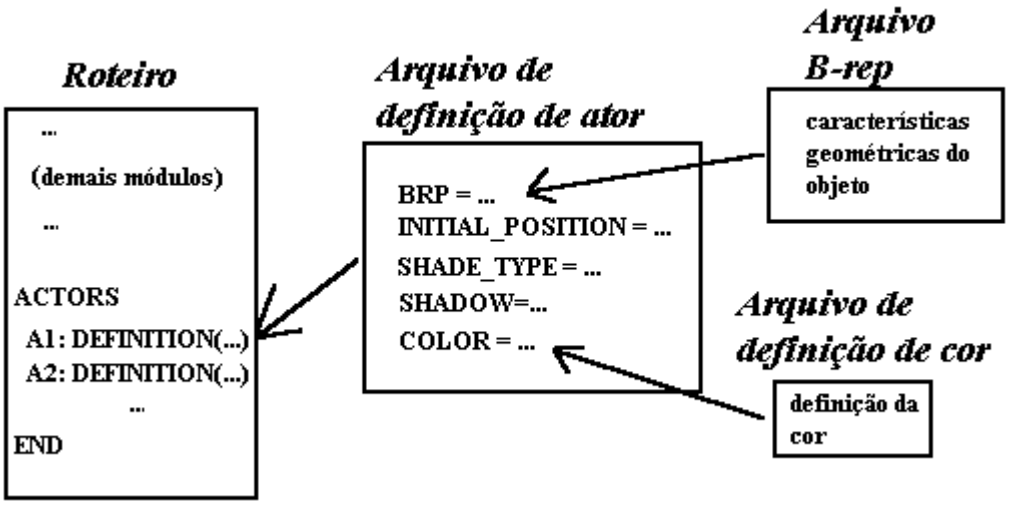

Figura 2: Módulo ACTORS

A geometria dos atores (arquivo B-rep) é definida pelo ProSIm, no módulo de Modelagem Geométrica.

Os movimentos dos atores também são especificados neste módulo; eles devem ser colocados após a definição de cada ator. O movimento deve ser sempre precedido pelo intervalo de tempo em que ocorrerá. Os delimitadores de intervalo usados são:

- `At`: determina que o movimento que se segue ocorrerá exatamente naquele instante (ou naquele quadro). Exemplos.: `At(2.0)`, `At(FRAME(60))`. O primeiro exemplo indica que o movimento ocorrerá para t = 2.0 s; o segundo indica o movimento no 60º. quadro. Os dois exemplos anteriores serão idênticos, se *Rate* (a taxa de quadros por segundo, definida no módulo `GENERAL`) for igual a 30.
- `After`: indica que o movimento ocorrerá após determinado instante (ou determinado quadro). Exemplos: `After(FRAME(95))`, `After(3.0)`.
- `Before`: O movimento ocorrerá do início até determinado instante (ou quadro). Exemplo: `Before(FRAME(22))`.
- `Between`: indica que o movimento ocorrerá entre dois intervalos de tempo (incluindo estes dois instantes). Exemplos: `Between (FRAME(4),FRAME(50))` indica que o movimento começa no quadro 4 e termina no 50; `Between(5.0,FRAME(300))` indica que o movimento vai do instante t = 5.0 s até o 300º. quadro. Aqui também pode ser usada a palavra-chave `END`, indicando o último quadro da animação, como usada no módulo `RENDER`. Assim, `Between(5.,END)` indica um movimento começando em t = 5.0 s e que vai até o fim da animação.

Obs.: Repare que apenas a primeira letra das palavras indicadoras de intervalo são maiúsculas.

Quando o intervalo se prolonga por mais de um quadro (em todos os casos, exceto quando se usa `At`), é necessário especificar se os valores dos parâmetros do movimento se referem ao movimento de um quadro para outro ou se referem ao movimento total do intervalo (por exemplo, se é dito que um ator deve girar 10^o, deve-se saber se ele vai girar 10^o por quadro ou se no final do intervalo ele deverá ter girado 10^o). Para isso existem as palavras-chave `EACH_FRAME` (indicando que o movimento desejado se dará a cada quadro) e `TOTAL` (indicando que o movimento desejado se completará no final do intervalo especificado). Quando o delimitador de intervalo é `At`, não se pode colocar `EACH_FRAME` ou `TOTAL`.

As linhas do roteiro que indicam os movimentos dos atores devem ter a seguinte forma:

delimitador de intervalo(...) movimento(...) `EACH_FRAME` (ou `TOTAL`)

Os possíveis movimentos de atores, com seus respectivos parâmetros são listados a seguir:

- `part`: sem parâmetros. Indica que o ator está ativo na cena. Esta função só tem sentido se for usada depois de `unpart`, pois por default, todos os atores definidos participam de toda a cena.

- `unpart`: indica que o ator não aparecerá na cena, no intervalo especificado. Para que o ator reapareça, é preciso usar `part`. Ex.:

```
ACTORS
     A0: DEFINITION("/home/usr/def/actor0.adef")
         Between(0.,FRAME(100)) unpart
         After(FRAME(100)) part
END
```

  No exemplo anterior, a animação terá apenas um ator, que não participará da cena até o quadro 100, só aparecendo a partir do 101^{o}.

- `translate_actor(A_R, x, y, z)`: realiza a translação do ator. O primeiro parâmetro é a condição `ABSOLUTE` ou `RELATIVE`. *Translação absoluta significa levar o ator exatamente para o ponto desejado. Translação relativa significa deslocar o ator de "um valor na(s) direção(ões) especificada(s) em relação à sua posição anterior (seu centro de gravidade)".* /SILV-92/. Os parâmetros `x`, `y` e `z` são do tipo *double* (da linguagem C) e representam as coordenadas do ponto para onde o ator se deslocará (se a translação for absoluta) ou o valor que o ator se deslocará em cada direção, a partir de seu centro de gravidade (se a translação for relativa). Veja o exemplo:

```
ACTORS
   A0: DEFINITION("actor0.adef")
       At(2.5) translate_actor(ABSOLUTE,0.,3.,2.)
   A1: DEFINITION("actor1.adef")
       At(2.5) translate_actor(RELATIVE,0.,3.,2.)
END
```

Neste exemplo, no instante 2.5 segundos, o ator `A0` será deslocado para o ponto (0, 3, 2). O ator `A1` sairá da posição onde estava e, no instante seguinte, terá deslocado 3 unidades na direção y e 2 unidades na direção z.

- `translate_actor_x(A_R, n)`: é um caso particular da função anterior, em que o deslocamento se faz de `n` unidades na direção x. É equivalente a se fazer o ponto da função anterior como sendo (n, 0., 0.). Assim, por exemplo, fazer: `translate_actor_x(ABSOLUTE,3.)` é a mesma coisa que fazer: `translate_actor(ABSOLUTE, 3.,0.,0.)`. Veja o exemplo:

```
ACTORS
   A0: DEFINITION("actor0.adef")
        At(2.5) translate_actor_x(ABSOLUTE,1.5)
   A1: DEFINITION("actor1.adef")
        At(2.5) translate_actor_x(RELATIVE,1.0)
END
```

No exemplo, o ator `A0` será deslocado para o ponto (1.5,0.,0), no instante 2.5 segundos. O ator `A1`, que sofreu translação relativa, vai se deslocar 1 unidade na direção x, a partir da posição onde estava.

- `translate_actor_y(A_R, n)`: equivalente à função anterior, porém fazendo-se o deslocamento na direção y. Portanto, fazer `translate_actor_y(RELATIVE, 1.4)` é o mesmo que fazer: `translate_actor(RELATIVE, 0., 1.4, 0.)`.

- `translate_actor_z(A_R, n)`: exatamente igual às duas anteriores, fazendo o deslocamento na direção z.

- `translate_actor_actor(A_R, first, second, n)`: `first` e `second` são atores previamente definidos. Se `A_R` for `ABSOLUTE`, o ator se moverá para junto de `second`. Se for `RELATIVE`, o ator se moverá `n` unidades na direção dada por `cg_first - cg_second` (onde cg_ significa centro de gravidade do ator).

- `rotate_actor(A_R, x, y, z, n)`: realiza a rotação do ator. O primeiro parâmetro é a condição `ABSOLUTE` ou `RELATIVE`. *Rotação absoluta "rotaciona o ator em relação ao eixo especificado em torno da*

origem, modificando também o posicionamento do ator no espaço". Rotação relativa "rotaciona o ator em relação ao eixo especificado em torno do seu centro de gravidade, só mudando seu direcionamento no espaço, não a sua posição". /SILV-92/. A rotação positiva é determinada pela "regra da mão direita". Os parâmetros `x`, `y` e `z` determinam o eixo de rotação; `n` é o valor em graus da rotação do ator. Por exemplo:

```
ACTORS
  A0: DEFINITION("actor0.adef")
    After(3.)
     rotate_actor(ABSOLUTE,1.,0.,2.,60.)
                                        EACH_FRAME
  A1: DEFINITION("actor1.adef")
    After(3.)
     rotate_actor(RELATIVE,1.,0.,2.,-20.)
                                          TOTAL
END
```

No exemplo, a partir do instante 3 segundos, o ator `A0` vai girar, a cada quadro, 60^o em torno da origem (rotação absoluta), tendo como eixo o vetor (1, 0, 2). O ator `A1` vai girar, até o final da animação, -20^o em torno de seu centro de gravidade, tendo o mesmo vetor como eixo.

- `rotate_actor_x(A_R, n)`: caso particular da função anterior, em que o eixo da rotação é o eixo x. O segundo parâmetro é o valor em graus da rotação. Assim, por exemplo: `rotate_actor_x(RELATIVE, 30.)` é o mesmo que `rotate_actor(RELATIVE, 1., 0., 0., 30.)`. Exemplo:

```
ACTORS
  A0: DEFINITION("actor0.adef")
       After(3.5) rotate_actor_x(ABSOLUTE,-50.)
                                        EACH_FRAME
  A1: DEFINITION("actor1.adef")
       After(3.5) rotate_actor_(RELATIVE,25.)
                                         EACH_FRAME
END
```

No exemplo, a partir do instante 3.5 segundos, `A0` vai girar, a cada quadro, -50^o em torno da origem (rotação absoluta), tendo como eixo de rotação o eixo x. O ator `A1` vai girar, a cada quadro, 25^o em torno de seu centro de gravidade, com o mesmo eixo de rotação (eixo x).

- `rotate_actor_y(A_R, n)`: semelhante à anterior, considerando o eixo y como o eixo de rotação.

- `rotate_actor_z(A_R, n)`: semelhante às duas anteriores, considerando o eixo z como o eixo de rotação.

- `free_rotate_actor(px, py, pz, vx, vy, vz, n)`: esta função permite novas alternativas para ponto de referência na rotação (na absoluta, era usada a origem como referência; na relativa, o centro de gravidade do ator). Os três primeiros parâmetros desta função são as coordenadas do ponto de referência desejado (em torno do qual o ator girará). Depois, vêm as coordenadas do vetor que determina o eixo de rotação. O último parâmetro é o valor em graus que o ator girará. Exemplo:

```
ACTORS
  A0: DEFINITION("/home/usr/def/esfera.adef")
        At(FRAME(10))   free_rotate_actor(2.,5.,6.,
                                          0.,1.,2., 45.)
END
```

No exemplo, no quadro 10, a esfera girará 45º em torno do ponto (2, 5, 6). A rotação terá como eixo o vetor (0, 1, 2).

Obs.: Se o ponto de referência for a origem (0, 0, 0), a função `free_rotate_actor` será equivalente à função `rotate_actor (ABSOLUTE, ...)`. Se o ponto de referência for o centro de gravidade do ator, ela será equivalente a `rotate_actor(RELATIVE, ...)`.

- `rotate_actor_actor(A_R, first, second, n)`: `first` e `second` são atores previamente definidos. O eixo de rotação é definido por `cg_first - cg_second` (onde cg_ significa centro de gravidade do ator). Se `A_R` for `ABSOLUTE`, o ator girará em torno de `second`. Se for `RELATIVE`, o ator girará em torno do seu centro de gravidade. O parâmetro `n` é o número de graus girados.

- `scale_actor(A_R, x, y, z)`: realiza o escalamento, que significa o aumento ou diminuição do tamanho do objeto em uma ou mais direções. O primeiro parâmetro é a condição `ABSOLUTE` ou `RELATIVE`. *"Escalamento absoluto aumenta ou diminui o tamanho do objeto em uma ou três direções em relação à origem, causando a movimentação do ator, além do escalamento. (...) Escalamento relativo aumenta ou diminui o tamanho do objeto em uma ou três direções em relação ao centro de gravidade do próprio ator, alterando somente seu tamanho"*. /SILV-92/. Os parâmetros `x`, `y` e `z` são coordenadas que representam o quanto o ator vai crescer ou diminuir em cada direção. Estas coordenadas devem ser dadas em valores percentuais. Valores negativos nas coordenadas significa diminuição de tamanho. Exemplo:

```
ACTORS
   A0: DEFINITION("actor.adef")
       At(0.) scale_actor(ABSOLUTE,30.,20.,0.)
       Between(FRAME(30), END)
          scale_actor(RELATIVE,-50.,0.,-60.)
```

```
                                                    TOTAL
END
```

No exemplo, no primeiro quadro da animação, o ator vai aumentar, com relação à origem (de maneira absoluta): 30% na direção x, 20% na direção y e nada na direção z. No final da animação, ele vai ter seu tamanho diminuído, com relação ao seu centro de gravidade (de maneira relativa): 50% na direção x e 60% na direção z, diminuição esta que começa no quadro 30.

- `scale_actor_x(A_R, n)`: caso particular da função anterior, em que o escalamento só se dá na direção x. O segundo parâmetro é a quantidade (em valores percentuais) que o ator aumetará ou diminuirá. Dessa maneira, `scale_actor_x(RELATIVE,35.)` é o mesmo que `scale_actor (RELATIVE, 35., 0., 0.)`.

- `scale_actor_y(A_R, n)`: semelhante à função anterior, sendo que o escalamento ocorre apenas na direção y.

- `scale_actor_z(A_R, n)`: semelhante às duas funções anteriores, sendo que o escalamento ocorre apenas na direção z.

- `growth(A_R, n)`: função que permite um crescimento igual nas três dimensões. O parâmetro n é o valor, em porcentagem, deste crescimento. Assim, por exemplo, `growth(ABSOLUTE,65.)` é o mesmo que `scale_actor(ABSOLUTE,65.,65., 65.)`.

- `shrink(A_R, n)`: semelhante a `growth`; permite uma diminuição de tamanho igual nas três dimensões. O segundo parâmetro (sempre maior que 0 e menor que 100) é o valor, em porcentagem, desta diminuição. Uma diminuição de 100% fará com que o ator desapareça. A comparação entre esta função e a anterior pode ser feita, por exemplo, da seguinte maneira: `shrink(RELATIVE, 40.)=growth(RELATIVE, -40.)`.

- `free_scale_actor(px, py, pz, vx, vy, vz)`: permite novas alternativas para ponto de referência no escalamento (se absoluto, era usada a origem como referência; se relativo, o centro de gravidade do ator). Os três primeiros parâmetros são as coordenadas do ponto de referência desejado. Os três parâmetros seguintes são as coordenadas que determinam o valor percentual do escalamento em cada direção. Exemplo:

```
ACTORS
  A0: DEFINITION("/home/usr/def/esfera.adef")
     At(FRAME(14))      free_scale_actor(0.,2.,6.,
                                         0.,10.,-36.)
END
```

No exemplo, no quadro 14, a esfera aumentará 10% na direção y e diminuirá 36% na direção z (coordenadas do vetor). A transformação se dará tendo como referência o ponto (0, 2, 6).

Obs.: Se o ponto de referência for a origem, `free_scale_actor` será igual a `scale_actor(ABSOLUTE,...)`. Se o ponto de referência for o centro de gravidade do ator, ela será igual a `scale_actor (RELATIVE,...)`.

- `scale_actor_actor(A_R, first, second, n)`: `first` e `second` são atores previamente definidos. O escalamento se dará na direção definida por `cg_first - cg_second` (onde cg_ significa centro de gravidade do ator). Se `A_R` for `ABSOLUTE`, o ator se moverá na direção do escalamento. Se for `RELATIVE`, o ator sofrerá o escalamento com relação ao seu centro de gravidade, não se movendo. O parâmetro `n` é a percentagem do escalamento.

- `repaint (texture_type, color)`: permite que num determinado instante, o ator seja "repintado", isto é, adquira uma nova textura, dada pelo arquivo, cujo nome é o segundo parâmetro. O parâmetro `texture_type` deve ser coerente com o arquivo de textura usado, e pode assumir os valores da variavel `SHADE_TYPE`, ou seja, `BASIC`, `PHONGS`, `STRAUSS`, `MARBLE`, `GRANITE` ou `WOOD`. Veja o exemplo a seguir, onde, no frame de número 19 o ator passará a ter uma textura de mármore, dada pelo arquivo `marb.cor`.

```
ACTORS
     A0: DEFINITION ("/tmp/actor1.adef")
         At(FRAME(19))
             repaint(MARBLE, "marb.cor")
END
```

Variáveis globais, declaradas no módulo *general* podem ser usadas como parâmetro para os movimentos, como mostrado no exemplo a seguir:

```
GENERAL
     TOTALTIME = 8.
     FR_RATE = 20.
     double x1 = read_file("angulos.out")
     double x2 = 10.
END

ACTORS
     A0: DEFINITION("def/actor0.adef")
         INITIAL_POSITION = 0., 5., -6.
         After(0.) translate_actor_z(RELATIVE, 3.)
                                                EACH_FRAME
     A1: DEFINITION("def/actor0.adef")
```

```
            At(0.) scale_actor_x(RELATIVE, x2)
            Before(4.) rotate_actor_y(RELATIVE,x1)
                                                EACH_FRAME
END
```

No exemplo anterior, tem-se um fragmento de roteiro de uma animação de 8 segundos, a uma taxa de 20 quadros/s. Ela tem dois atores. O ator `A0` tem sua posição inicial modificada em relação à posição dada no seu arquivo de definição (`def/actor0.adef`); ele se moverá 3 unidades na direção z, a cada quadro. No quadro 0, o ator `A1` aumentará seu tamanho na direção x de 10% (valor da variável `x2`) e até o instante t = 4 s, ele rotacionará um valor `x1`, em relação ao eixo y, a cada quadro. Este valor de `x1` será lido no arquivo `angulos.out` a cada quadro (significando que ele pode girar ângulos diferentes, de um quadro para outro).

Como observação final, deve ser dito que os movimentos envolvendo atores do tipo sequência de B-reps (`GEO_MORPH`, no arquivo de definição) devem ocorrer no interalo em que a sequência está presente na animação. Caso o movimento comece antes da aparição do primeiro B-rep da sequência, o resultado poderá não ser o esperado.

3) GROUPS:

Neste módulo, os atores são agrupados, de modo que os movimentos possam ser realizados não só com os atores individualmente, mas também com grupos deles. Se não houver interesse em agrupar atores, este módulo pode estar ausente no roteiro.

Num grupo de atores, devem ser especificados os atores que o compõem (devidamente definidos no módulo `ACTORS`) e o centro de gravidade do grupo (para que os movimentos do mesmo possam ser realizados relativamente a este ponto).

No que diz respeito aos intervalos em que estão ativos e à definição de seus movimentos, os grupos de atores são idênticos aos atores isolados (com exceção das funções `part` e `unpart`, que não são válidas para grupos).

Portanto, este módulo é muito parecido com o módulo `ACTORS`, se diferenciando dele apenas porque:

1. Os grupos devem ser nomeados, nesta ordem, de `G0`, `G1`, ... (ao invés de `A0`, `A1`,...).
2. Ao invés de usar a função `DEFINITION`, com o nome do arquivo com as características do ator, o grupo é definido com a função `COMPONENTS`, que tem como parâmetros os nomes dos atores que compõem o grupo. Por exemplo: `COMPONENTS(A0,A1,A2)` define um grupo composto por estes três atores. Um grupo também pode conter grupos previamente definidos. Por exemplo: `COMPONENTS(G0,A6,A7)` define um grupo composto pelos atores do grupo `G0` mais os atores `A6` e `A7` (a restrição é que `G0` tenha sido previamente definido).

3. Após a definição do grupo (com COMPONENTS), é preciso setar a variável GC que indica o centro de gravidade do grupo (pode ser as três coordenadas de um ponto qualquer ou o nome de um ator, indicando que o centro de gravidade dele será o centro do grupo).

O fragmento de roteiro a seguir, mostra os módulos ACTORS e GROUPS:

```
ACTORS
     A0: DEFINITION("def/actor0.adef")
     A1: DEFINITION("def/actor1.adef")
     A2: DEFINITION("def/actor3.adef")
         At(0.) growth(RELATIVE, 50.)
     A3: DEFINITION("def/actor4.adef")
END

GROUPS
   G0: COMPONENTS(A0,A1,A2)
       GC = 0., 0., 0.
       After(2.) rotate_actor_x(RELATIVE,9.)
                                          TOTAL
    G1: COMPONENTS(A0,A2)
       GC = A2
       At(FRAME(6))
          translate_actor_y(RELATIVE,3.)
   G2: COMPONENTS (G1, A3)
       GC = 0., 0., 0.
END
```

No exemplo, são definidos quatro atores, e o terceiro (A2) cresce 50% no quadro 0 (função growth). São também definidos três grupos: o primeiro é composto pelos três primeiros atores, o segundo por A0 e A2 e o terceiro por A0, A2 e A3 (ou seja, G1 mais A3). Os grupos G0 e G2 têm seus centros de gravidade na origem, e G1 no centro de gravidade do ator A2. O grupo G0 (portanto, seus três atores) vai ter girado 9^o até o final da animação (com o movimento iniciado depois de t = 2 s). Os atores do grupo G1 vão se mover 3 unidades na direção y, no quadro 6.

4) CAMERA:

Neste módulo, são especificados os parâmetros de câmera (observador, centro de interesse - coi -, distância focal, etc), como vistos no modelo da figura 3.

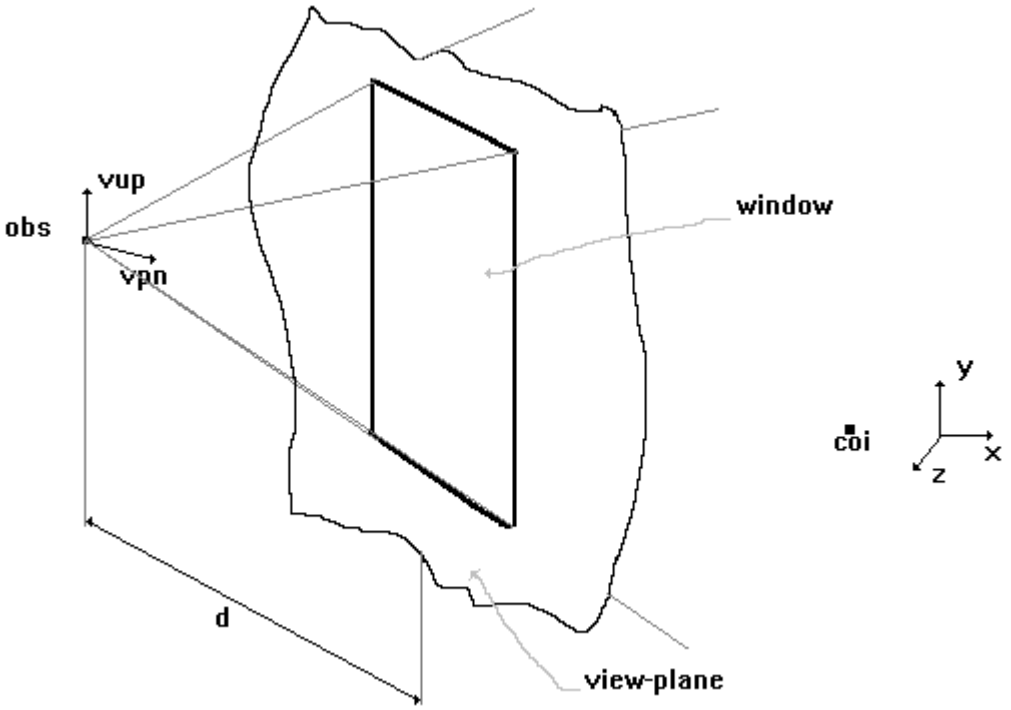

Figura 3: Modelo de câmera.

Na figura anterior:

- **obs** é a posição do observador (em projeção perspectiva é o centro de projeção).
- **coi** - *center of interest* - é o centro da atenção do observador (um ponto da cena para o qual ele está olhando).
- **vup** - *view up vector* - direção que identifica o que é "para cima". Isto é, se o ponto obs é o olho do observador, a direção vup orienta a cabeça do mesmo.
- ***view-plane*** é o plano de visualização, onde os objetos são projetados.
- **d** - focal distance - é a distância entre o observador e o plano de visualização.
- **vpn** - identifica a direção de visualização, dado por (vpn = coi - obs).

Além destes, ainda existem parâmetros que definem o enquadramento da imagem:

- **window** - "é uma porção retangular do plano de visualização que delimita a imagem de interesse e a pirâmide de visulização da mesma". /SILV-92/.
- **viewport** - porta de visualização - é a "porção retangular da tela do monitor que será efetivamente usada para expor a imagem contida na window. Esta é dimensionada em pixels e define as coordenadas bidimensionais do dispositivo (DC - Device Coordinate)" /SILV-92/.

Ainda neste módulo, poderão ser especificados os movimentos de câmera (alteração da posição do observador, do centro de interesse, *zoom*, *traveling*, etc) e os intervalos de tempo em que eles ocorrem, da mesma maneira que no módulo`ACTORS`.

Assim como no caso de atores, a câmera deve ter um arquivo de definições, onde seus parâmetros são definidos. O exemplo a seguir mostra um arquivo de definição de câmera:

```
OBS = 8., 2., 0.
COI = 0., 0.5, 0.
VUP = 0., 1., 0.
D = 1.
VIEWPORT = 5, 320, 5, 200
APERTURE = 32., 32.
```

O arquivo anterior define uma câmera (observador) no ponto (8, 2, 0), olhando para o ponto (0, .5, 0), sendo y o eixo apontando para cima (vup). Além disso, a distância focal é 1 e o ângulo de abertura da pirâmide de visualização é 32^o nas duas direções (este ângulo, juntamente com a distância `d`, define a `WINDOW`, que pode ser dada neste arquivo, alternativamente). A imagem gerada terá dimensões 320 x 200, definida por `VIEWPORT` (os valores 5 dados neste parâmetro servem apenas para indicar o local da tela onde a imagem seria vista, em versões antigas do programa).

Os valores podem aparecer em qualquer ordem no arquivo de definição e, para todos eles, existirão os valores default, dados a seguir:

```
OBS = 1.5, 1.5, 4.0
COI = 0., 0., 0.
VUP = 0., 1., 0.
D = 1.
VIEWPORT = 5, 100, 5, 100
WINDOW = 0.57735, 0.57735
```

Como default, a janela é especificada, ao invés do ângulo de abertura.

Assim como no caso dos atores, o arquivo de definição de câmera virá referenciado no módulo `CAMERA`, através da função `DEFINITION`:

```
CAMERA
     DEFINITION("def/camera.cdef")
END
```

No módulo `CAMERA`, apenas um arquivo de definição é usado. Mas qualquer um dos parâmetros do arquivo de definição pode ser redefinido no roteiro, se colocado depois do `DEFINITION`. Por exemplo, é possível trocar o observador e o centro de interesse definidos:

```
CAMERA
     DEFINITION("def/camera.cdef")
     OBS = 3., 6., -2.5
     COI = -3., 0., 0.
END
```

Com relação aos movimentos de câmera, eles seguem a mesma estruturação dos movimentos de atores ou grupos de atores:

delimitador de intervalo(...) movimento(...) `EACH_FRAME` (ou `TOTAL`)

A seguinte convenção é usada para os movimentos de câmera: movimentos começados com `go_` lidam com o observador, os que começam com `aim_` movem o centro de interesse e os que começam com `set_` redefinem os parâmetros de câmera. A lista de movimentos de câmera é dada a seguir:

- `go_forward(n)`: move o observador para frente, na direção do centro de interesse. O parâmetro `n` fornece o valor, em unidades, deste movimento. Seu efeito é o de um zoom, aumentando os detalhes da cena próximo ao coi. Se esta função for usada de tal modo que o observador ultrapasse o coi, o observador verá as "costas" do objeto. Exemplo:

```
CAMERA
     DEFINITION("camera1.cdef")
     After(FRAME(30)) go_forward(0.5) EACH_FRAME
END
```

Neste exemplo, o observador se moverá 0.5 unidades para a frente, em cada quadro, a partir do 31º.

- `go_backward(n)`: reverso de `go_forward`, ou seja, afasta o observador do centro de interesse.

Uma maneira de se considerar o movimento do observador em volta do coi é considerar o último como o centro de um cubo, no qual o observador anda paralelo aos seus lados. Para representar as direções nas quais ele se movimenta, são definidas as funções `go_up`, `go_down`, `go_right` e `go_left`, como mostra a figura 4:

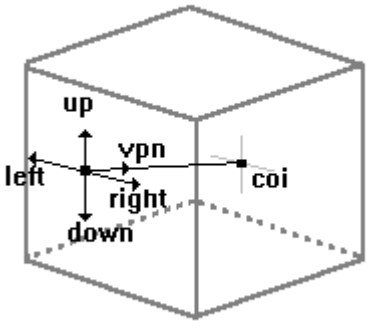

Figura 4: Coi considerado como centro de um cubo.

O plano que contém as direções up, down, left e right é o plano que contém o vetor vup, ou seja, o plano perpendicular a vpn. A direção up é determinada por vup, e as

outras, por consequência. Este plano não depende dos eixos cartesianos, mas do sistema de coordenadas de visualização (VC - View Coordinates).

- `go_up(n)`: desloca o observador na direção de vup. O valor deste deslocamento é dado pelo parâmetro `n`. Seu uso repetitivo faz com que o observador passe a olhar o objeto em coi cada vez mais de cima. Exemplo:

```
CAMERA
     DEFINITION("camera1.cdef")
     Between(FRAME(20),FRAME(40)) go_up(7.)
                                              TOTAL
END
```

Neste exemplo, entre o quadro 20 e o 40, o observador se deslocará 7 unidades para cima.

- `go_down(n)`: desloca o observador na direção oposta de vup. Seu uso repetitivo faz com que o observador passe a olhar o objeto em coi, cada vez mais de baixo.

- `go_right(n)`: desloca o observador para a direita da sua posição atual, sobre o plano perpendicular à direção de visualização (vpn), ou seja, paralelo ao plano de visualização. O valor deste deslocamento é dado pelo parâmetro `n`. Seu uso repetitivo provoca um deslocamento espiral crescente para a direita, uma vez que a direção de visualização é recalculada toda vez e, consequentemente, um novo plano perpendicular.

- `go_left(n)`: semelhante a `go_right`, só que deslocando o observador para a esquerda. Seu uso repetitivo provoca um deslocamento espiral crescente para a esquerda.

Existe uma outra maneira de considerar o movimento do observador em torno do coi. Esta maneira consiste em considerar o coi como o centro de uma esfera sobre a qual desloca-se o observador nas direções cardeais. Este método garante que o movimento mantém o raio da esfera constante. As funções são: `go_north`, `go_south`, `go_east` e `go_west`, como indica a figura 5:

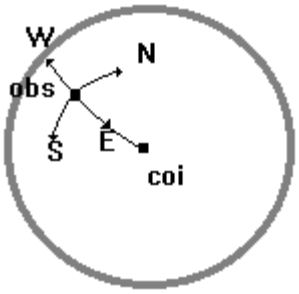

Figura 5: Coi como centro de uma esfera.

Por convenção, o Norte é indicado pelo vup e as outras direções são avaliadas considerando o observador olhando para o coi, tendo a sua direita o Leste.

- `go_north(n)`: desloca o observador na direção Norte. O valor em graus deste deslocamento é dado pelo parâmetro `n`. Seu uso repetitivo provoca uma volta sobre o objeto (coi) sem modificar a distância entre o coi e o observador. Deve ser usada para ângulos "pequenos" (entre -90^o e $+90^o$).

- `go_south(n)`: semelhante a `go_north`, sendo que o deslocamento se dará de `n` graus na direção oposta (Sul). Assim: `go_south(n)` = `go_north(-n)`.

- `go_east(n)`: desloca o observador na direção Leste (direita) de `n` graus. Seu uso repetitivo provoca uma volta sobre o objeto (coi) sem modificar a distância entre o coi e o observador.

- `go_west(n)`: semelhante a `go_east`, sendo que o deslocamento se dará na direção oposta (Oeste).

- `go_flying(x, y, z)`: permite o deslocamento do observador para uma posição arbitrária (semelhante a `set_obs`, que será vista adiante), mas garantindo que a direção de visualização seja recalculada, para se ajustar à direção deste movimento. Os parâmetros da função são as coordenadas da posição para a qual o observador se deslocará. O nome “go_flying" (vá voando) foi escolhido para tentar criar um paralelo entre a funcionalidade desejada e o movimento de um avião, por exemplo, cujo centro de interesse e direção de visualização estão sempre na atual direção de vôo.

- `go_crow(x, y, z)` : "arrasta" o observador para a posição especificada, mantendo vpn constante, isto é, "arrastando" também o coi. Os valores dos deslocamentos (translação) do observador e do coi são dados pelos parâmetros da função. Produz o efeito do carrinho, ou "traveling", para os animadores convencionais. Por exemplo:

```
CAMERA
     DEFINITION("camera1.cdef")
     Before(5.) go_crow(0.,1.,-1.) EACH_FRAME
END
```

No exemplo anterior, o observador e o coi se deslocam 1 unidade na direção y e 1 unidade na direção -z a cada quadro, antes do instante t = 5 segundos.

- `obs_actor(nome)`: especifica um ator, cujo centro de gravidade será a posição do observador. O parâmetro é o nome deste ator. Este ator não aparecerá na animação; seu objetivo é apenas servir de guia para a câmera. Veja o seguinte fragmento de roteiro:

```
ACTORS
     A0: DEFINITION("../def/actor1.adef")
END

CAMERA
     DEFINITION("../def/camera.cdef")
     At(4.) obs_actor(A0)
END
```

Neste exemplo, no instante t = 4 s, observador passará a se localizar no centro de `A0`, que desaparecerá.

- `aim_up(n)`: desloca o centro de interesse de `n` unidades na direção vup.

- `aim_down(n)`: desloca o centro de interesse de `n` unidades na direção oposta a vup.

- `aim_in(n)`: aproxima o coi do observador em `n` unidades. A primeira vista, essa função pode parecer idêntica a `go_forward`, sendo que quem se move é o coi. Entretanto, aproximar o observador significa ver a animação mais de perto e aproximar o coi pode apresentar efeito visual diferente.

- `aim_out(n)`: afasta o coi do observador de `n` unidades. Pode parecer idêntica a `go_backward`, mas afastar o observador significa ver a animação mais de longe e afastar o coi pode representar um efeito visual diferente (por exemplo, se o coi se afastar na reta que liga o observador a ele, nenhuma alteração visual ocorre).

- `aim_right(n)`: desloca o centro de interesse em `n` unidades para a direita do observador, paralelamente ao plano de visualização.

- `aim_left(n)`: desloca o centro de interesse para a esquerda do observador, paralelamente ao plano de visualização.

- `aim_north(n)`: desloca o centro de interesse de `n` graus na direção norte, sobre uma esfera com centro no observador.

- `aim_south(n)`: semelhante a `aim_north`, mas na direção sul. Assim: `aim_south(n) = aim_north(-n)`.

- `aim_east(n)`: desloca o centro de interesse de `n` graus.

- `aim_west(n)`: semelhante a `aim_east`, mas deslocando na direção oeste sobre a esfera centrada no observador.

- `aim_actor(nome)`: especifica um ator, cujo centro de gravidade será o centro de interesse. O parâmetro é o nome deste ator. Não pode haver `EACH_FRAME` ou `TOTAL` associados a esta função, pois não tem sentido falar em "mirar" o ator em cada quadro (uma vez colocado o ator como coi, ele só será modificado através de alguma das funções `aim_` ou `set_coi`, que será vista adiante). Veja o seguinte fragmento de roteiro:

```
ACTORS
     A0: DEFINITION("../def/actor1.adef")
END

CAMERA
     DEFINITION("../def/camera.cdef")
     At(4.6) aim_actor(A0)
END
```

Neste exemplo, no instante t = 4.6 s, o centro de interesse passará a ser o ator `A0`.

- `aim_scene`: determina que o centro de interesse vai ser o "centro de gravidade geral" da cena. Este "centro de gravidade geral" é especificado como sendo a média dos centros de gravidade dos atores. Não possui parâmetros. É uma maneira de "centralizar" a visualização de uma cena com muitos atores.

- `set_obs(x, y, z)`: função que muda a posição do observador. Esta função ainda verifica se o observador coincide com o centro de interesse do objeto (coi); se isto acontecer, aparece uma mensagem de erro. Exemplo:

```
CAMERA
     DEFINITION("camera1.cdef")
     At(4.0) set_obs(5.,3.,-3.)
```

```
END
```

No exemplo anterior, no instante t = 4 s, o observador se desloca para o ponto (5, 3, -3).

- `set_coi(x, y, z)`: altera a posição do centro de interesse. A função também verifica se o centro de interesse coincide com o observador; se isto acontecer, aparece uma mensagem de erro. Exemplo:

```
CAMERA
     DEFINITION("camera1.cdef")
     At(FRAME(244)) set_obs(15.,0.,-1.)
END
```

Neste exemplo, no quadro 244, o centro de interesse muda para (15, 0, -1).

- `set_vup(x, y, z)`: altera a posição de vup. Os três parâmetros são as coordenadas do novo vetor vup. A função retornará mensagens de erro se: os três parâmetros forem nulos ou se vup for colinear a vpn (onde vpn = coi - obs).

- `set_d(n)`: altera a distância focal. O valor do parâmetro não pode ser zero.

- `set_viewport(x_corner, x_length, y_corner, y_length)`: função que redefine a viewport. Os parâmetros são inteiros, dados em coordenadas de tela do dispositivo (DC - device coordinates). Assim:

```
CAMERA
     DEFINITION("camera1.cdef")
     At(FRAME(0)) set_viewport(5,300,5,300)
END
```

O exemplo anterior permite a visão da animação em janelas de 300x300 (DC) na tela.

- `set_window(u_min, u_max, v_min, v_max)`: altera a *window* (parte do plano de projeção que contém a imagem que nos interessa). A dimensão destes parâmetros deve ser compatível com o valor do parâmetro `d`, para garantir um cone de projeção perspectiva coerente. Além disso `u_min` e `v_min` devem ser, respectivamente, iguais a -`u_max` e -`v_max`.

- `set_aperture (alfa_u, alfa_v)`: redefine os ângulos de abertura da pirâmide de visualização. Assim, se pode definir a janela (*window*), sem a preocupação com o valor de `d`.

Existem ainda funções com nomes relacionados aos termos usados em cinema. Estas funções são, na maioria, idênticas a funções já apresentadas.

- `zoom_in(n)` : efetua um aumento nos detalhes da imagem, através da alteração do tamanho da *window* (mantendo *viewport* constante). O tamanho da janela é modificado na porcentagem desejada (parâmetro `n`). Se o parâmetro for 100 (%), a janela se reduz a zero, não aparecendo a imagem. Esta função assemelha-se a `go_forward`, sendo que a última não altera a pirâmide de visualização, mas a posição efetiva do observador. Cabe ao usuário escolher qual o melhor efeito para maior detalhamento da cena.

- `zoom_out(n)`: efetua um encolhimento nos detalhes da imagem, através da alteração do tamanho da *window* (mantendo *viewport* constante). É o oposto de `zoom_in`, assemelhando-se a `go_backward`.

- `spin_clock(n)`: modifica (roda) a direção de vup de `n` graus sobre o eixo vpn. A rotação é feita no sentido horário visto pelo observador que olha na direção vpn. O efeito é a rotação da imagem (semelhante ao efeito de torcer a cabeça para visualizar a cena).

- `spin_Cclock(n)`: parecida com a função anterior, sendo que a rotação é feita no sentido anti-horário. Assim: `spin_Cclok(n) = spin_clock(-n)`.

- `tilt_up(n)`: o mesmo que `aim_north(n)`. Definida pelo movimento de elevar a cabeça para alcançar a visualização de alguma coisa acima (na vertical).

- `tilt_down(n)`: o mesmo que `aim_south(n)`. Definida pelo movimento de abaixar a cabeça para alcançar a visualização de alguma coisa abaixo.

- `pan_right(n)`: o mesmo que `aim_east(n)`. Executa um movimento para visualização na linha do horizonte (linha panorâmica) para a direita.

- `pan_left(n)`: o mesmo que `aim_west(n)`. Executa um movimento para visualização na linha do horizonte para a esquerda.

- `traveling(x, y, z)`: o mesmo que `go_crow(x, y, z)`: modifica a posição do observador, sem modificar a direção de visualização vpn; ou seja, a mesma translação é aplicada ao observador e ao coi.

- `see_track(trilha)`: permite a visualização da trilha dada como parâmetro. É necessário que a trilha seja definida no módulo `TRACKS`. Após esta função não deve ser colocado `EACH_FRAME` ou `TOTAL`.

5) LIGHTS:

Trata da iluminação. Neste módulo são definidas as cores do fundo (*background*) e da iluminação ambiente (*environment*), com possibilidade de *fade-in* e *fade-out*, em intervalos de tempo especificados.

Este módulo também trata das fontes de luz adicionais que podem ser colocadas na animação. Existem dois tipos de fontes de luz (se classificadas com relação à sua área): fontes puntuais e fontes extensas. Fontes de luz puntuais são representadas por um ponto no espaço emitindo energia radiante (na prática, aquelas cuja dimensão é desprezível com relação às dimensões dos objetos iluminados). Fontes extensas são aquelas cuja dimensão não é desprezivel frente às dimensões dos objetos iluminados.

Os tipos de fontes de luz possíveis são:

- `POINT` (lâmpada), é uma fonte puntual e que irradia energia luminosa em todas as direções. É modelada pela sua cor e posição no espaço.
- `SUN` (sol), é uma fonte extensa e que produz raios de energia luminosa paralelos entre si. É modelada pela cor e pela direção dos raios luminosos paralelos.
- `SPOT`, é uma fonte puntual direcional (isto é, tem uma direção principal na qual ocorre a máxima concentração de energia luminosa; fora desta direção, ocorre uma atenuação desta energia). O *spot* é modelado através de sua cor, posição, direção de atuação e ângulo de abertura. O SIPP ainda permite um recurso adicional, que é escolher entre o *spot* sem atenuação da energia luminosa, ou seja, com uma terminação abrupta (`SHARP`), ou o *spot* com atenuação da luz (`SOFT`), que é mais realista.

As fontes podem ser acesas ou apagadas em qualquer instante da animação. Também podem ter qualquer uma de suas características alteradas (cor, posição e direção).

Assim como os atores e a câmera, a iluminação deve ser definida num arquivo externo ao roteiro. Neste arquivo devem estar definidas as cores da iluminação ambiente e do fundo da imagem, através de suas componentes r, g e b, da seguinte maneira:

```
ENVIRONMENT = 65000., 65500., 48700.
BACK = 65000., 55000., 10000.
```

Pela convenção do ProSIm, os valores de cada componente das cores variam de 0. a 65535.

Além da definição das cores de fundo e da iluminação ambiente, o arquivo de definições deve listar as fontes de luz da animação, com suas características (as fontes devem ser chamadas de `L0`, `L1`, e assim por diante). O exemplo a seguir mostra um arquivo de definição de iluminação, com três fontes de luz: uma lâmpada (`POINT`), um sol (`SUN`) e um *spot*. Observe que cada tipo de fonte exige a definição de parâmetros diferentes:

```
ENVIRONMENT = 63600., 60000., 58000.
```

```
BACK = 65000., 65000., 65500.

L0:  TYPE = POINT
     POSITION = 25., 4., -1.
     COLOR = 60000., 60000., 60000.

L1:  TYPE = SUN
     DIRECTION = 0., -1., 1.
     COLOR = 0., 63000., 63000.

L2:  TYPE = SPOT
     POSITION = -3., 3., 1.5
     DIRECTION = 3., -4., -2.
     COLOR = 65535., 0., 0.
     SOLID_ANGLE = 35.
     SPOT_TYPE = SOFT
```

O arquivo de definição de iluminação é referenciado no roteiro da mesma maneira que o dos atores e o de câmera:

```
LIGHTS
     DEFINITION("def/luzes1.ldef")
END
```

Por default, todas as fontes definidas no arquivo referenciadas estão acesas. Para apagá-las, deve-se usar a função `TURN_OFF`, que tem como parâmetro o nome da fonte a ser apagada (obviamente, esta fonte deve estar definida no arquivo de definições). Para reacendê-las, deve-se usar `TURN_ON`. Veja o exemplo a seguir, onde a fonte `L0` será apagada no início da animação e acesa apenas no quadro 150:

```
LIGHTS
     DEFINITION("def/luzes1.ldef")
     At(0.0) TURN_OFF(L0)
     At(FRAME(150)) TURN_ON(L0)
END
```

Obs.: `TURN_ON/OFF` exige que o delimitador de tempo seja `At` (se, por exemplo, uma fonte for desligada, ela permanecerá desligada até que seja novamente ligada; portanto, não tem sentido falar em `After(...) TURN_OFF(...)`, por exemplo).

Também é possível mudar a posição, a direção e/ou a cor de qualquer uma das fontes de luz, num determinado instante de tempo. Para isso, são usadas as mesmas palavras-chaves do arquivo de definição, mas tendo como parâmetro o nome da fonte de luz que será modificada. O exemplo a seguir usa o arquivo de definição apresentado anteriormente e altera a cor da lâmpada (no instante t = 1.5 s), a direção do sol (no quadro 100) e a posição do *spot* (no quadro 126):

```
LIGHTS
     DEFINITION("luzes.ldef")
```

```
        At(1.5) COLOR(L0) = 0., 65500., 10000.
        At(FRAME(100)) DIRECTION(L1) = 1., -1., 0.
        At(FRAME(126)) POSITION(L2) = 2., 3.6, -2.
END
```

A cor de fundo também pode ser alterada de maneira semelhante. No exemplo a seguir, em 3.6 s a cor de fundo passará a ser azul:

```
LIGHTS
        DEFINITION("/tmp/light.ldef")
        At(3.6) BACK = 0., 0., 65535.
END
```

O módulo *lights* ainda possui funções que realizam os *fades* de iluminação (*fade-in* significa ir acendendo a luz, do preto até a sua cor e *fade-out*, o contrário). É possível realizar *fades* de qualquer uma das fontes definidas no arquivo através das funções `FADE_IN` e `FADE_OUT`, que têm como parâmetro o nome da fonte que sofrerá o *fade*. Como os *fades* ocorrem em intervalos de tempo (e não em um único instante), o delimitador de tempo usado para estas funções é sempre `Between`. Exemplo:

```
LIGHTS
        DEFINITION("luzes.ldef")
        Between(0.,3.) FADE_IN(L1)
        Between(20.,END) FADE_OUT(L1)
END
```

No exemplo anterior, a fonte `L1` sofrerá um *fade-in* durante os três primeiros segundos de animação e um *fade-out* a partir do 20º. segundo.

Os *fades* não são restritos às fontes de luz; a iluminação ambiente e a cor de fundo também podem sofrê-los. Para isso, existem as seguintes funções (sem parâmteros): `FADE_IN_ENV`, `FADE_OUT_ENV`, `FADE_IN_BACK` e `FADE_OUT_BACK`. O exemplo a seguir mostra um fragmento de roteiro de uma animação que se inicia com um *fade-in* da iluminação ambiente e termina com um *fade-out* da cor de fundo:

```
LIGHTS
        DEFINITION("luzes.ldef")
        Between(0.,2.5) FADE_IN_ENV
        Between(16.,END) FADE_OUT_BACK
END
```

Ainda é possível fazer um *fade* generalizado, englobando todas as fontes definidas, a iluminação ambiente e a cor de fundo, através das funções `FADE_IN_ALL` e `FADE_OUT_ALL`. Estas duas funções também não possuem parâmetros.

Assim como o observador ou o centro de interesse, fontes de luz também podem "seguir" um determinador ator. A função `light_follow_actor(L0, A1)`, por exemplo, faz com que a fonte `L0` se mova de acordo com o ator `A1` (que não aparece). Nesse caso, a fonte não pode ser do tipo `SUN` (que não tem posição). A função

`spot_aim_actor(L1, A3)`, faz com que o spot `L1` esteja sempre direcionado para o ator `A3`, enquanto a função `sun_direct_actor(L2, A4, A7)` faz com que a fonte direcional `L2` se direcione sempre para o eixo dado por `cg_A4 - cg_A7` (centro de gravidade dos atores).

6) RENDER:

Determina os intervalos da animação que serão passados ao subsistema de *rendering*. A determinação destes intervalos é importante porque a etapa de *rendering* é a mais demorada do processo de criação de uma animação por computador e, portanto, a limitação dos quadros a serem gerados pode dividir o processo em partes menores, além de agilizar a etapa de testes, quando só é preciso visualizar determinados trechos da animação.

Este módulo se organiza como no exemplo a seguir:

```
RENDER
      FROM 2.0
      TO 3.0
END
```

No exemplo acima, só serão gerados os quadros entre os instantes t = 2 e t = 3 segundos.

Uma opção à determinação do intervalo de geração em segundos, é a definição do mesmo em quadros (frames), como no exemplo a seguir:

```
RENDER
      FROM FRAME(40)
      TO FRAME(60)
END
```

No exemplo anterior, a geração começará no quadro de número 40 e terminará no de número 60.

Se este módulo não estiver presente no roteiro, o programa assumirá que todos os quadros da animação devem ser gerados (valores default: `FROM 0.0` e `TO END`).

Outro fator que pode agilizar a etapa de testes, é gerar apenas um quadro em cada grupo de *n* quadros consecutivos (por exemplo, gerar um a cada cinco quadros), dentro do intervalo de *rendering* determinado. Para isso, usa-se `ONE_FR_IN`. Apenas os quadros de números múltiplos de `ONE_FR_IN` serão gerados. Exemplo:

```
RENDER
      FROM 0.
      TO FRAME(80)
      ONE_FR_IN 4
END
```

No exemplo anterior, só serão gerados os quadros 0, 4, 8, 12, ..., 76 e 80.

7) OUTPUT:

Neste módulo é dado o nome dos arquivos que conterão as imagens e a qualidade das imagens geradas. O sistema de animação, integrado ao SIPP /YNGV-94/, possibilita como saída a visualização da animação em MPEG /GONG-94/. É também possível ter como saída apenas os quadros que comporão a animação (com a renderização completa ou na forma de *wireframe*).

O fragmento de roteiro a seguir ilustra o módulo *output*:

```
OUTPUT
      OUTPUT_PATH = "/tmp/quadros"
      QUALITY = SIPPPPM
      SIPP_SHADE = FLAT_SIPP
      SIPP_SHADOW = TRUE
      SIPP_SAMPLE = 2
END
```

No exemplo anterior, os quadros da animação serão gerados como `/tmp/quadros0.ppm`, `/tmp/quadros1.ppm`, e assim por diante. Se o nome dos arquivos de saída não forem especificados neste módulo, o programa assume o nome default de `animacao`, no diretório do mesmo.

O parâmetro `QUALITY` indica o tipo/qualidade da renderização. Este pode assumir os seguintes valores:

- `DEBUG`: nenhum quadro é gerado; apenas o programa é "debugado". Útil apenas para o programador mais experiente, que conhece a linguagem do TOOKIMA.
- `SIPPTGA`: as imagens serão renderizadas com o SIPP /YNGV-94/. O tipo de renderização a ser realizada por ele é determinada por parâmetros adicionais (`SIPP_SHADE`, `SIPP_SHADOW` e `SIPP_SAMPLE`), que serão vistos a seguir. A saída será no formato TGA não comprimido.
- `SIPPPPM`: a saída terá o formato PPM (ou PBM, se for usada a opção `LINE` no parâmetro `SIPP_SHADE` - ver parágrafo seguinte).
- `SIPPMPEG`: o resultado será a animação MPEG e seus quadros PPM.

O parâmetro `SIPP_SHADE` determina o tipo de tonalização (*shading*) da cena. Pode assumir os seguintes valores: `PHONG_SIPP` (default; usa o método de Phong para o *shading* /ROGE-85/), `GOUR_SIPP` (usa Gouraud *shading*), `FLAT_SIPP` (usa Flat *shading*), `LINE` (visualização em *wireframe*) ou `LINE_ENVELOPE` (também *wireframe*, mas só é desenhado o envoltório convexo tridimensional - *bounding box* - de cada ator).

As opções `LINE` e `LINE_ENVELOPE` gerarão quadros PBM, se a qualidade for `SIPPPPM` ou `SIPPMPEG`; elas não são compatíveis com a qualidade `SIPPTGA`. Estas duas opções apresentam renderização mais rápida, e devem ser usadas para pré-visualização da animação.

O parâmetro `SIPP_SHADOW` pode assumir os valores `TRUE` ou `FALSE`, indicando se é desejada ou não a visualização de sombras nas imagens. É válido ressaltar também que apenas fontes de luz do tipo *spot* podem gerar sombras (isso será visto no módulo *lights*).

O parâmetro `SIPP_SAMPLE` é um inteiro que indica o tamanho da superamostragem (*oversampling)* que o SIPP fará para evitar *alias*. Cada pixel será renderizado internamente como uma matriz de (`SIPP_SAMPLE` x `SIPP_SAMPLE`) subpixels, cuja cor média representará o pixel na imagem final. O valor default é 1. Esse parâmetro é ignorado se a qualidade de render (`SIPP_SHADE`) for `LINE`.

Como todos os parâmetros deste módulo têm valores default, sua presença é opcional (embora aconselhável) no roteiro.

8) TRACKS:

Este módulo permite a utilização de um recurso poderoso do TOOKIMA, que é a definição de trajetórias ("caminho no espaço a ser percorrido por um determinado elemento da cena" /SILV-92/) e a geração de trilhas (*tracks*). A trajetória é definida através da interpolação de curvas, a partir de pontos de controle. Em seguida, são geradas as trilhas, sobre as quais o ator poderá se movimentar. O objetivo das trilhas é tornar os movimentos mais suaves e naturais /RAPO-97/.

Neste módulo é definido o tipo de trajetória desejada, ou seja, se é uma trajetória "conhecida" (que usa uma função matemática conhecida, como linear, quadrática, etc para a determinação de seus pontos de controle) ou se é uma trajetória "livre" (cujos pontos de controle são dados num arquivo). Se for uma trajetória "conhecida", devem ser dados os pontos iniciais e finais da mesma. A partir destes pontos, a função escolhida determina os pontos de controle que servirão para o cálculo de uma spline cúbica (trilha). Se for uma trajetória "livre", deve ser dado o nome do arquivo onde são definidos os pontos de controle.

As trajetórias não tratam apenas de pontos. Podem existir trilhas interpolando vetores, valores de cores (base RGB) e valores reais (por exemplo, ângulo de giro de um objeto).

As trilhas definidas neste módulo devem ser denominadas `T0`, `T1`, etc. Após o nome da trilha definida, deve ser dado o tipo da trilha (`TYPE`) no roteiro. Este tipo pode assumir os seguintes valores: `LINEAR`, `QUADR`, `CUBIC` ou `EXP`. Estes valores indicam que os valores serão interpolados segundo uma função matemática conhecida (x, x^2, x^3 ou $x.e^{(x-1)}$, respectivamente). Uma outra alternativa, é dar os pontos de controle da trilha num arquivo externo (a primeira linha deste arquivo deve ser o número de pontos de controle). Nesse caso, o parâmetro `TYPE` assume o nome do arquivo de pontos de controle.

Outro parâmetro que deve ser definido para cada trilha é o seu valor (VALUE), que indica o tipo de elemento que compõe a trilha. Esse valor pode ser: POINT (para as posições sucessivas dos atores, ou da câmera, por exemplo), DOUBLE (para ângulos de rotação, por exemplo), VECTOR (para eixos de rotação) ou COLOR.

Quando se tratar de trilhas baseadas em funções matemáticas (ou seja, quando o parâmetro TYPE não for nome de arquivo), é ainda necessário definir os parâmetros START e STOP (que serão pontos, reais, vetores, ou cores, de acordo com o valor do parâmetro VALUE).

O exemplo a seguir ilustra a definição de trilhas:

```
TRACKS
     T0:  TYPE = "pontos_trilha.in"
          VALUE = POINT

     T1:  TYPE = QUADR
          VALUE = DOUBLE
          START = 1.5
          STOP = 30.

     T2:  TYPE = EXP
          VALUE = COLOR
          START = 0., 0., 0.
          STOP = 65535., 65535., 65535.
END
```

Neste exemplo, a trilha T0 é uma trilha de pontos (VALUE = POINT), lida a partir do arquivo pontos_trilha.in. Este arquivo deve ter as três coordenadas de cada um dos pontos de controle (se fosse uma trilha de reais, por exemplo, o arquivo deveria conter os valores destes números reais). A trilha T1 é uma trilha de reais, que começa em 1.5 e vai até 30., e os valores serão interpolados segundo uma função quadrática. Já a trilha T2 é uma trilha de cores, que começa no preto (0,0,0) e termina no branco (65535, 65535, 65535). As cores serão interpoladas segundo a função exponencial.

Para usar os valores de uma trilha como parâmetros para movimentos de atores, câmera, etc, é necessário definir uma variável global (no módulo *general*) com a função following_track. Esta função deve ser definida da seguinte maneira:

- following_track(A_R,trilha,aceleracao,A_R,begin,end): o primeiro parâmetro é ABSOLUTE ou RELATIVE, e indica o modo de caminhamento pela trilha. *"Caminhar por uma trilha de maneira ABSOLUTA (relativa à origem) significa ir para um determinado ponto na curva da trajetória, pois é usada uma translação absoluta para esta tarefa. Caminhar de maneira RELATIVA (relativa à última posição) significa perguntar ao procedimento (...) quanto variou o parâmetro do ator entre a última 'posição' e a atual. (...) Caso escolha-se ABSOLUTE, o ator segue rigorosamente a curva definida. (...) Caso escolha-se RELATIVE (...), o ator segue a curva, mas a partir da posição em que o elemento está inicializado (como se a curva tivesse sido*

definida a partir da posição do ator).” /SILV-92/. O segundo parâmetro desta função é o nome da trilha que se deseja seguir. O terceiro parâmetro indica a “aceleração” com que a trilha será percorrida; ele pode ter os seguintes valores:

- `linear`: indica movimento uniforme.
- `slow_in`: movimento acelerado.
- `slow_out`: movimento desacelerado.
- `acc_dec`: movimento acelerado até a metade do percurso (também metade do tempo) e desacelerado na segunda metade.
- `dec_acc`: desacelerado até a metade e depois acelerado.

O quarto parâmetro é mais uma vez a condição `ABSOLUTE` ou `RELATIVE`, indicando se *“o valor de retorno é para ser calculado a partir do começo ou se é para obter apenas a variação no último instante, respectivamente”*. /SILV-92/. Quando se deseja que o ator se desloque a cada instante para o ponto definido pela trilha, deve-se usar `ABSOLUTE`. A condição `RELATIVE` leva a resultados dificilmente previsíveis. Os dois últimos parâmetros indicam o intervalo de tempo em que a trilha será percorrida (podem ser valores reais, indicando o tempo em segundos, ou em quadros da animação - `FRAME(n)`) . É aconselhável que o intervalo do movimento (translate, rotate, etc) que usará um parâmetro proveniente de `following_track` seja o mesmo especificado nesta função. Caso contrário, o resultado poderá ser inesperado.

O fragmento de roteiro a seguir mostra os módulos *general*, *tracks* e *actors*:

```
GENERAL
     TOTALTIME = 10.
     FR_RATE = 10.
     Point trackp = following_track(ABSOLUTE,
               T0, slow_in, ABSOLUTE, 0., END)
                         ; definida de 0. ao fim
     double trackd = folowing_track(ABSOLUTE,
              T1, linear, ABSOLUTE, 0. FRAME(50))
                         ; definida entre 0. e o
                         ; quadro 50
END

ACTORS
     A0: DEFINITION(“def/actor0.adef”)
         Between(0., END)
            translate_actor(ABOLUTE, trackp.x,
               trackp.y, trackp.z) EACH_FRAME
                         ; o mesmo intervalo de
                         ; definição de T0
     A1: DEFINITION(“def/actor1.adef”)
       Between(0., FRAME(50))
        scale_actor_y(RELATIVE,*trackd)
                                   EACH_FRAME
                         ; o mesmo intervalo de
```

```
                                   ; definição de T1
END

TRACKS
     T0:  TYPE = LINEAR
          VALUE = POINT
          START = 10., 0., -1.
          STOP = 1.5, 13., -6.5
     T1:  TYPE = "trilha.in"
          VALUE = DOUBLE
END
```

No exemplo anterior, são definidas duas variáveis globais. A primeira delas (trackp) é do tipo `Point` e é usada como valores para a translação absoluta do ator `A0`. Ela assumirá os valores da trilha `T0` (trilha de pontos, intercalados linearmente, entre os valores de `START` e `STOP`), percorrendo-a de maneira acelerada (`slow_in`), em todo o intervalo da animação (de `0.` a `END`). A segunda variável global (`trackd`) é real e é usada para o escalamento de `A1`. Ela assumirá os valores da trilha `T1`, que é uma trilha de reais, controlada pelos valores dados no arquivo `trilha.in`. `T1` será percorrida com movimento uniforme (parâmetro `linear` em `following_track`), na primeira metade da animação (de `0.` a `FRAME(50)`).

III - EXEMPLOS

Para completar a exposição da linguagem para a composição do roteiro de animação, serão dados a seguir dois exemplos comentados:

EXEMPLO 1:

```
GENERAL
      TOTALTIME = 10.          ; Animação de 10 s
      FR_RATE = 2              ; Taxa de 2 quadros/s
END

ACTORS
      A0:   DEFINITION("def/actor1.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
      A1:   DEFINITION("def/actor2.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
      A2    DEFINITION("def/actor3.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
      A3:   DEFINITION("def/actor4.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
      A4:   DEFINITION("def/actor5.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
      A5:   DEFINITION("def/actor6.adef")
            After(0.) rotate_actor_y(ABSOLUTE, 7.) EACH_FRAME
END                        ; Animação com 6 atores, que girarão 7º em torno de
                           ; y, a cada quadro

CAMERA
      DEFINITION = "def/cam2.cdef"
      VIEWPORT = 5,512,5,486  ; Viewport é alterado em relação ao definido
                              ; no arquivo acima. A imagem terá dimensões
                               ; 512 x 486
      After(0.) go_north(60.) TOTAL
                              ; No final da animação, o observador terá ido
                              ; 60% para o norte, em relação ao quadro 0
END

LIGHTS
      DEFINITION = "def/light2.ldef"
      Between(FRAME(18),END) FADE_OUT_ALL
                           ; Todas as luzes vão se apagando, a partir do quadro 18
      At(0.) TURN_OFF(L1)
      At(0.) TURN_OFF(L2)      ; Fontes 1 e 2, definidas em def/light2.def
                               ; permanecerão apagadas
END

RENDER
      FROM FRAME(0)
      TO FRAME(10)         ; Só serão renderizados os 10 primeiros quadros
END
```

```
OUTPUT
      OUTPUT_PATH = "bolas"          ; A saída será uma animação
      QUALITY = SIPPMPEG             ; no formato MPEG
      SIPP_SHADE = PHONG_SIPP
      SIPP_SHADOW = TRUE
      SIPP_SAMPLE = 3
END
```

EXEMPLO 2:

```
GENERAL
      TOTALTIME = 10.          ; Animação de 8 s
      FR_RATE = 15             ; Taxa de 15 quadros/s
      double x1 = read_file("valores.in")
                               ; x1 terá valores lidos em arquivo
      Point trackp = following_track(ABSOLUTE, T0,
                     slow_out, ABSOLUTE, 0., 7.5)
                               ; Seguirá trilha T0, desaceleradamente, até
                               instante 7.5 seg
      Color trackc = following_track(RELATIVE, T1,
                     linear, ABSOLUTE, 4.5, END)
                               ; Seguirá T1, uniformemente, a partir do
                               instante 4.5 seg
END

ACTORS
      A0:   DEFINITION("def/actor1.adef")
            BRP = "/tmp/new_obj.brp"
            At(0.) shrink(RELATIVE, 50.)
                               ; A0 terá seu arquivo .brp modificado em relação
                               ; ao do seu arquivo de definição e reduzirá seu
                               ; tamanho em 50%, no início da animação
      A1:   DEFINITION("def/actor2.adef")
            Before(7.5) translate_actor(ABSOLUTE, trackp.x,
                                trackp.y, trackp.z) EACH_FRAME
                               ; A1 se moverá para os pontos dados pela trilha  T0
      A2:   DEFINITION("def/actor3.adef")
            After(2.) unpart
                               ; A2 não participará mais da cena, depois de 2
                               segundos
END

GROUPS
      G0:   COMPONENTS(A0, A1, A2)
                               ; Grupo composto dos 3 atores da animação
            GC = 0., -1. ,2.
            After(3.) free_rotate_actor(1.,1.,1.,0.,-1.,-1.,20.)
                                                              TOTAL
      G1:   COMPONENTS(A0, A1)
            GC = A0
            At(3.6) scale_actor_z(RELATIVE, -5.)
                               ; Este grupo terá o centro de gravidade no centro
                               ; de A0 e se reduzirá 5% na direção z, no instante
                               ; 3.6 segundos. Atenção porque o escalamento será
                               ; relativo ao centro de A0, podendo causar efeito
```

```
                                    ; inesperado em A1
END

CAMERA
      DEFINITION = "def/cam1.cdef"
      After(7.): go_flying(1., 1., 1.) EACH_FRAME
                                    ; A partir de 7 segundos, será executado o
                                    ; go_flying, a cada quadro
      Between(FRAME(15),END) aim_down(0.5) EACH_FRAME
                                    ; O coi se deslocará 0.5 unidades para baixo,
                                    ; a partir do quadro 15
END

LIGHTS
      DEFINITION = "def/light1.ldef"
      Between(FRAME(100),FRAME(130)) FADE_OUT_BACK
                                    ; O fundo sofre fade-out, neste intervalo
      At(0.) TURN_OFF(L1)
      At(5.) TURN_ON(L1)            ; Fonte 1, definida em def/light1.def só se
                                    ; acenderá em t = 5 s
      At(6.) POSITION(L0) = 5., 6., -3.6
                                    ; Em 6 s, a posição da fonte 0 é alterada
      After(4.5) COLOR(L0) = trackc.r, trackc.g, trackc.b
                                    ; A cor de L0 passa a mudar, de acordo com os
                                    ; valores da trilha T1 (trilha de cores)
END

OUTPUT
      OUTPUT_PATH = "/tmp/teste"
      QUALITY = SIPPTGA             ; Saída em quados TGA
      SIPP_SHADE = GOUR_SIPP
      SIPP_SHADOW = FALSE
      SIPP_SAMPLE = 1
END

TRACKS
      T0:   TYPE = CUBIC
            VALUE = POINT
            START = 10., 10., 10.
            STOP = =10., -10., 10
                                    ; T0 é trilha de pontos, com interpolação
                                    ; dada por função cúbica
      T1:   TYPE = "/tmp/trilha.tr"
            VALUE = COLOR
                                    ; T1 é trilha de cores, lida em arquivo
END
```

IV - DESCRIÇÃO FORMAL DA LINGUAGEM

A definição formal da sintaxe de uma linguagem de programação é importante tanto para o usuário quanto para o desenvolvedor. Para o usuário, ela serve como referência, pois é uma descrição clara da linguagem. Para o desenvolvedor, ela facilita a manutenção da linguagem.

A BNF ("Backus Normal Form") é uma notação para escrever "gramáticas" /DONO-72/, /GEAR-74/ e será utilizada para descrever formalmente a sintaxe da linguagem de roteiros do ProSIm, informalmente descrita nas seções precedentes. A BNF consiste de sentenças que definem a maneira em que a linguagem de programação deve ser escrita. Ela é chamada de *meta-linguagem*, e usa caracteres diferentes daqueles que são usados na linguagem por ela descrita. Na BNF, os símbolos não-terminais (isto é os símbolos da meta-linguagem) são delimitados por < > e representam estágios intermediários no processo de descrição da linguagem.

O sinal **::=** significa: "é substituído por". Assim, a sentença: <*digito_0*> *::=* 0 é lida: "o meta-símbolo <*digito_0*> é substituído por 0".

O símbolo **|** também é usado na BNF, indicando alternativa ("ou"). Assim, para representar um digito qualquer, tem-se: <*digito*> *::=* 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

Para facilitar a descrição da linguagem de roteiros do ProSIm, foram acrescentados novos símbolos à meta-linguagem. Esses símbolos se baseiam na meta-linguagem usada por /PRES-92/ para a definição de um dicionário de dados. Os símbolos da meta-linguagem a ser usada e seus respectivos significados são descritos na tabela a seguir.

Símbolo	Significado
< >	delimitam meta-símbolo
::=	"é substituído por"
|	"ou"
[ ]	opcional
{ }^{n}	"n repetições de"
⇒	continuação de linha
/* */	delimitam comentários

A linguagem de roteiros do ProSIm pode ser assim descrita:

```
/* Definições gerais */

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<inteiro> ::= <digito> | <inteiro> <digito>
                          /* representa qualquer número
                          inteiro*/
<v_inteiro> ::= <inteiro> | <string>
                          /* variável inteira: valor ou nome de
                          variável (que pode ser usada como
```

```
                              parâmetro de movimento), com a
                              restrição de que a variável assuma
                              valores do tipo desejado */
<real_pos> ::= <inteiro> | <inteiro>. | .<inteiro> |
               ⇒ <inteiro>.<inteiro>
<v_real_pos> ::= <real_pos> | <string>
<real_neg> ::= -<real_pos>
<v_real_neg> ::= <real_neg> | <string>
<real> ::= <real_pos> | <real_neg>
<v_real> ::= <real> | <string>
<letra> ::= a | b | c | ... | z | A | B | C | ... | Z
                              /* qualquer letra minúscula ou
                              maiúscula/
<c_especial> ::= - | _ | . | # | ...
                              /* qualquer caracter não alfa-
                              numérico que possa fazer parte
                              do nome de um arquivo (com
                              as restrições do sistema; i.e.,
                              número de caracteres, etc) */
<string> ::= <letra> | <string> <letra> | <string> <inteiro> |
             ⇒ <string> <c_especial>
                              /* qualquer combinacao de caracteres
                              comecada com letra */
<path> ::= / | <string> | <c_especial> |
           ⇒ <path>/| <path> <string> |
           ⇒ <path> <c_especial>
                              /* o "path" de um arquivo, com as
                              restrições do sistema (i.e., os tipos
                              de caraceteres permitidos, etc) */
<a_r> ::= ABSOLUTE | RELATIVE
<t_f> ::= TRUE | FALSE
<e_t> ::= EACH_FRAME | TOTAL
<tempo> ::= <real_pos> | FRAME(<inteiro>)
                              /* indica tempo em segundos ou em
                              frames */
<tempo_2> ::= <tempo> | END
<t_mov_1> ::= At(<tempo_2>)
                              /* nao precisa <e_t> */
<t_mov_2> ::= Before(<tempo_2>) | After(<tempo>) |
              ⇒ Between(<tempo>, <tempo_2>)
                              /* exigem <e_t> */


/* Módulo GENERAL */

<mod_general> ::= GENERAL <totaltime_def> <rate_def>
                  ⇒ { [<ext_var>] }^n END
                              /* n é um número qualquer */
<totaltime_def> ::= TOTALTIME = <real_pos>
<rate_def> ::= FR_RATE = <inteiro>
<ext_var> ::= <var_type> <string> = <var_value>  |
```

```
                ⇒ <var_type_pr> <string> = <var_value_pr>
                                /* variável externa: <tipo> <nome> =
                                <valor> */
<var_type> ::= int | float | double | ...
                                /* qualquer tipo de variável da
                                linguagem C */
<var_type_pr> ::= Point | Vector | Color
                                /* tipos de variáveis do ProSIm */
<var_value> ::= <inteiro> | <real> | <fcao_read_file> |
                  ⇒ <fcao_foll_track>
                                /* a definição aqui não é livre de
                                contexto; é preciso saber o tipo da
                                variável para saber se ela assumirá
                                um valor inteiro ou real */
<var_value_pr> ::= <real>, <real>, <real> | <fcao_read_file> |
                    ⇒ <fcao_foll_track>
<fcao_read_file> ::= read_file("<path>")
                                /* <path> deve indicar um arquivo com
                                valores para a variavel */
<fcao_foll_track> ::= following_track(<a_r>, <tr_name>, <acc>,
                       ⇒ <a_r>, <tempo>, <tempo_2> )
                                /* <tr_name> será definido no módulo
                                TRACKS */
<acc> ::= linear | slow_in | slow_out | acc_dec | dec_acc


/* Módulo ACTORS */

<mod_actors> ::= ACTORS {<actor_name>: [<actor_def>]
                  ⇒ {[<actor_param>]}^s {[<actor_mov>]}^t}^n END
                                /* n atores (n, s e t quaisquer) */
<actor_name> ::= A<inteiro>
                                /* os atores devem ser numerados em
                                ordem crescente, a partir do 0 */
<actor_def> ::= DEFINITION("<path>")
                                /* <path> deve indicar um arquivo com
                                o formato de definição de atores:
                                <arq_def_atores> - ver OBS. no final
                                deste módulo */
<actor_param> ::= <brp_def> | <shade_def> | <color_def> |
                   ⇒ <p0_def>
<brp_def> ::= BRP = "<path_1>" | GEO_MORPH = "<path_2>",
                    ⇒ <inteiro>, <tempo_2>
                       /* onde <path_1> indica um arquivo do
                       formato B-Rep, com a extensão .brp e
                       <path_2> indica uma sequência numerada de
                       arquivos B-rep, sem a extensão .brp */
<shade_def> ::= SHADE_TYPE = <shade_type>
<shade_type> ::= BASIC | PHONGS | STRAUSS |
                  ⇒ MARBLE | GRANITE | WOOD
<shadow_def> ::= SHADOW = <t_f>
```

```
<color_def> :: = COLOR = "<path>"  | CLR_MORPH = "<path_2>",
                 ⇒ <inteiro>, <tempo_2>
                          /* onde <path_1> indica um arquivo de
                          cor, com a extensão .cor e <path_2>
                          indica uma sequência numerada de
                          arquivos de cor, sem a extensão
                          .cor */
<p0_def> ::= INITIAL_POSITION = < real>, <real>, <real>
<actor_mov> ::= <t_mov_1> <a_movs>  | <t_mov_2> <a_movs> <e_t>
<a_movs> ::= <parts> | <translates> | <rotates> | <scales> |
             ⇒ <repaint>
<parts> ::= part | unpart
<translates> ::= translate_actor(<a_r>, {<v_real>,}^2 <real>) |
                 ⇒ translate_actor_<xyz>(<a_r>, <v_real>) |
                 ⇒ translate_actor_actor(<a_r>, <actor_name>,
                 ⇒ <actor_name>, <real>)
<xyz> ::= x | y | z
<rotates> ::= rotate_actor(<a_r>, {<v_real>,}^3 <v_real>) |
              ⇒ rotate_actor_<xyz>(<a_r>, <v_real>) |
              ⇒ free_rotate_actor({<v_real>,}^6 <v_real>) |
              ⇒ rotate_actor_actor(<a_r>, <actor_name>,
              ⇒ <actor_name>, <real>)

<scales> ::= scale_actor(<a_r>, <v_real>, <v_real>, <v_real>) |
             ⇒ scale_actor_<xyz>(<a_r>, <v_real>) |
             ⇒ growth(<a_r>, <real>) | shrink(<a_r>, <v_real>) |
             ⇒ free_scale_actor({<v_real>,}^5 <v_real>) |
             ⇒ scale_actor_actor(<a_r>, <actor_name>,
             ⇒ <actor_name>, <real>)

<repaint> ::= repaint(<shade_type>, "<path>")
                          /* onde <path> indica um arquivo de
                           cor */

/* OBS.: Formato do arquivo de definição de atores: */
<arq_def_atores> ::= <brp_def> <shade_def> <color_def>
                     ⇒ [<p0_def>]


/* Módulo GROUPS */

<mod_groups> ::= GROUPS {<group_name>: <group_def>
                 ⇒ <gc_def> {[<group_mov>]}^t}^n END
                          /* n grupos (n e t quaisquer) */
<group_name> ::= G<inteiro>
                          /* os grupos de atores devem ser
                          numerados em ordem crescente, a
                          partir do 0 */
```

```
<aORg_name> ::= <actor_name> | <group_name>
<group_def> ::= COMPONENTS([<aORg_name>,]^n <aORg_name>)
                          /* os n+1 nomes de atores do grupo
                          devem ser diferentes */
<gc_def> ::= GC = <v_real>, <v_real>, <v_real> |
          ⇒ GC = <actor_name>
<group_mov> ::= <t_mov_1> <g_movs> | <t_mov_2> <g_movs> <e_t>
<g_movs> ::= <translates> | <rotates> | <scales>
                          /* <translates>, <rotates> e <scales>
                          já definidos no módulo ACTORS */


/* Módulo CAMERA */

<mod_camera> ::= CAMERA [<cam_def>] {[<cam_param>]}^s
               ⇒ {[<cam_mov>]}^t END
                          /* s e t quaisquer */
<cam_def> ::= DEFINITION("<path>")
                          /* <path> deve indicar um arquivo com
                          o formato de definição de câmera:
                          <arq_def_camera> - ver OBS. no final
                          deste módulo */
<cam_param> ::= <obs_def> | <coi_def> | <vup_def> | <d_def> |
               ⇒ <viewport_def> | <aperture_def> | <window_def>

<obs_def> ::= OBS = <real>, <real>, <real>
<coi_def> ::= COI = <real>, <real>, <real>
                          /* aqui também a linguagem não é
                          livre de contexto; é preciso que o
                          COI seja diferente do OBS */
<vup_def> ::= VUP = <real>, <real>, <real>
<d_def> ::= D = <real_pos*>
                          /* por <real_pos*> entenda-se um real
                          positivo diferente de 0 */
<viewport_def> ::= VIEWPORT = {<inteiro>,}^3 <inteiro>
<aperture_def> ::= APERTURE = <real_pos>, <real_pos>
<window_def> ::= WINDOW = <real_pos>, <real_pos>
<cam_mov> ::= <t_mov_1> <c_movs> | <t_mov_2> <c_movs> <e_t> |
            ⇒ <t_mov_1> aim_actor(<actor_name>) |
            ⇒ <t_mov_1> see_track(<tr_name>)
<c_movs> ::= <gos> | <aims> | <sets> | <standards>
<gos> ::= go_forward(<v_real>) | go_backward(<v_real>) |
        ⇒ go_up(<v_real>) | go_down(<v_real>) |
        ⇒ go_right(<v_real>) | go_left(<v_real>) |
        ⇒ go_north(<v_real>) | go_south(<v_real>) |
        ⇒ go_east(<v_real>) | go_west(<v_real>) |
        ⇒ go_flying(<v_real>, <v_real>, <v_real>) |
        ⇒ go_crow(<v_real>, <v_real>, <v_real>) |
        ⇒ obs_actor(<aORg_name>)
```

```
<aims> ::= aim_up(<v_real>) | aim_down(<v_real>) |
          ⇒ aim_in(<v_real>) | aim_out(<v_real>) |
           ⇒ aim_right(<v_real>) | aim_left(<v_real>) |
           ⇒ aim_north(<v_real>) | aim_south(<v_real>) |
           ⇒ aim_east(<v_real>) | aim_west(<v_real>) |
          ⇒ aim_actor(<aORg_name>) | aim_scene
<sets> ::= set_obs(<v_real>, <v_real>, <v_real>) |
          ⇒ set_coi(<v_real>, <v_real>, <v_real>) |
          ⇒ set_vup(<v_real>, <v_real>, <v_real>) |
          ⇒ set_d(<v_real_pos*>) |
          ⇒ set_viewport({<v_inteiro>,}3 <v_inteiro>) |
          ⇒ set_aperture(<v_real_pos>, <v_real_pos>) |
          ⇒ set_window(<v_real_neg>, <v_real_pos>,
          ⇒ <v_real_neg>, <v_real_pos>)
                          /* outra dependência do contexto: é
                          preciso que cada par <v_real_neg> e
                          <v_real_pos> tenha o mesmo valor
                          absoluto */
<standards> ::= zoom_in(<v_real>) | zoom_out(<v_real>) |
               ⇒ spin_clock(<v_real>) | spin_Cclock(<v_real>) |
               ⇒ tilt_up(<v_real>) | tilt_down(<v_real>) |
               ⇒ pan_right(<v_real>) | pan_left(<v_real>) |
               ⇒ traveling(<v_real>, <v_real>, <v_real>)

/* OBS.: Formato do arquivo de definição de câmera: */
<arq_def_camera> ::= [<obs_def>] [<coi_def>] [<vup_def>]
                   ⇒ [<d_def>] [<viewport_def>]
                   ⇒ [<apert_win_def>]
<apert_win_def> ::= [<aperture_def>] | [<window_def>]
```

/* Módulo LIGHTS */

```
<mod_lights> ::= LIGHTS [<lights_def>] {[<lights_param>]}s
                ⇒ {[<lights_mov>]}t END
                          /* s e t quaisquer */
<lights_def> ::= DEFINITION("<path>")
                          /* <path> deve indicar um arquivo com
                          o formato de definição de iluminação:
                          <arq_def_lights> - ver OBS. no final
                          deste módulo */
<lights_param> ::= <env_def> | <back_def> | <source_def>
<env_def> ::= ENVIRONMENT = <real_pos>, <real_pos>, <real_pos>
<back_def> ::= BACK = <real_pos>, <real_pos>, <real_pos>
                          /* sempre que se fala em cor, deve-se
                          lembrar que a faixa de valores para
                          cada componente (rgb) varia de 0 a
```

```
                              65535 */
<source_def> ::= <source_name>: <point>  |
                 ⇒ <source_name>: <sun>  |
                 ⇒ <source_name>: <spot>
<source_name> ::= L<inteiro>
                              /* as fontes de luz devem ser
                              numeradas em ordem crescente, a
                              partir do 0 */
<point> ::= TYPE = POINT
            ⇒ POSITION = <real>, <real>, <real>
            ⇒ COLOR = <real_pos>, <real_pos>, <real_pos>
<sun> ::= TYPE = SUN
          ⇒ DIRECTION = <real>, <real>, <real>
          ⇒ COLOR = <real_pos>, <real_pos>, <real_pos>
<spot> ::= TYPE = SPOT
           ⇒ POSITION = <real>, <real>, <real>
           ⇒ DIRECTION = <real>, <real>, <real>
           ⇒ COLOR = <real_pos>, <real_pos>, <real_pos>
           ⇒ SOLID_ANGLE = <real_pos>
           ⇒ SPOT_TYPE = <sharp_soft>
<sharp_soft> ::= SHARP | SOFT
<lights_mov> ::= <t_mov_1> <turns>  | <t_mov_1> <redefs>  |
                 ⇒ Between(<tempo>, <tempo_2>) <redefs>  |
                 ⇒ Between(<tempo>, <tempo_2>) <fades>  |
                 ⇒ <t_mov_2> <follows>
                              /* <turns> só pode vir com At;
                               <fades> deve vir com Between;
                               <follows> pode vir com qualquer */
<turns> ::= TURN_ON(<source_name>) | TURN_OFF(<source_name>)
<redefs> ::= POSITION(<source_name>) = {<v_real>,}² <v_real>  |
           ⇒ DIRECTION(<source_name>) = {<v_real>,}² <v_real>  |
           ⇒ COLOR(<source_name>)={<v_real_pos>,}² <v_real_pos>
           ⇒|BACK = <real_pos>, <real_pos>, <real_pos>
<fades> ::= FADE_IN(<source_name>) | FADE_OUT(<source_name>)  |
            ⇒ FADE_IN_ENV | FADE_IN_BACK | FADE_IN_ALL  |
            ⇒ FADE_OUT_ENV | FADE_OUT_BACK | FADE_OUT_ALL
<follows> ::= light_follow_actor(<source_name>, <aORg_name>)  |
              ⇒ spot_aim_actor(<source_name>, <aORg_name>)  |
              ⇒ sun_direct_actor(<source_name>, <aORg_name>)
                              /* Obs: No primeiro caso a fonte
                              não pode ser SUN, no segundo caso
                              ela deve ser SPOT e, no terceiro,
                              SUN */

/* OBS.: Formato do arquivo de definição de i luminação: */
<arq_def_lights> ::= [<env_def>] [<back_def>] {[<source_def>]}ⁿ
```

```
/* Módulo RENDER */

<mod_render> ::= RENDER [FROM <tempo>] [TO <tempo_2>]
                 ⇒ [ONE_FR_IN <int>] END


/* Módulo OUTPUT */

<mod_output> ::= OUTPUT [<out_path_def>] [<quality_def>]
                 ⇒ [<s_shade_def>] [<s_shadow_def>]
                 ⇒ [<s_sample_def>] END
<out_path_def> ::= OUTPUT_PATH = "<path>"
<quality_def> ::= QUALITY = <quality_type>
<quality_type> ::= DEBUG | SIPPTGA | SIPPPPM | SIPPMPEG
<s_shade_def> ::= SIPP_SHADE = <s_shade_type>
<s_shade_type> ::= PHONG_SIPP | GOUR_SIPP | FLAT_SIPP | LINE
<s_shadow_def> ::= SIPP_SHADOW = <t_f>
<s_sample_def> ::= SIPP_SAMPLE = <digito>
                            /* não é aconselhável usar dígito
                            maior que 3 */


/* Módulo TRACKS */

<mod_tracks> ::= TRACKS {[<tr_name>: <tr_def>]}^n END
<tr_name> ::= T<inteiro>
                            /* as trilhas ser numeradas em ordem
                            crescente, a partir do 0 */
<tr_def> ::= <tr_1> | <tr_2>
                            /* <tr_1> é trilha lida em arquivo e
                            <tr_2> é trilha "conhecida" */
<tr_1> ::= TYPE = "<path>" <value_def_1> |
           ⇒ TYPE = "<path>" <value_def_2>
                            /* onde <path> indica o arquivo de
                            pontos de controle da trilha */
<value_def_1> ::= VALUE = <value_type_1>
<value_def_2> ::= VALUE = <value_type_2>
<value_type_1> ::= POINT | VECTOR | COLOR
<value_type_2> ::= DOUBLE
<tr_2> ::= TYPE = <tr_type> <resto_1> |
           ⇒ TYPE = <tr_type> <resto_2>
<tr_type> ::= LINEAR | QUADR | CUBIC | EXP
<resto_1> ::= <value_def_1>
              ⇒ START = <real>, <real>, <real>
              ⇒STOP = <real>, <real>, <real>
<resto_2> ::= <value_def_2>
              ⇒ START = <real>
              ⇒ STOP = <real>
```

```
/* Um roteiro de animação é composto dos módulos anteriormente
definidos */

<ROTEIRO_DE_ANIMACAO> ::= <mod_general> [<mod_actors>]
                          ⇒ [<mod_groups>] [<mod_camera>]
                          ⇒ [<mod_lights>] [<mod_render>]
                          ⇒ [<mod_output>] [<mod_tracks>]
                            /* o roteiro será válido se conter
                            apenas o módulo GENERAL, pois todos
                            os outros módulos têm valores
                            default */
```

Para finalizar, alguns comentários devem ser feitos a respeito da formalização da linguagem de roteiros do ProSIm:

1. A notação BNF tem limitações no que diz respeito à dependência do contexto. Isto é, as classes de símbolos são expandidas sem referência ao contexto no qual elas estão inseridas /DONO-72/. Entretanto, a BNF foi usada em virtude de sua simplicidade e do pequeno número de situações em que há dependência do contexto na linguagem de roteiros do ProSIm (nessas situações, os comentários na descrição formal tentam suprir a limitação da BNF).

2. Como visto nas seções anteriores, a linguagem de roteiros apresentada tem grande flexibilidade no que diz respeito à ordem em que os módulos (ou os elementos internos a eles) podem ser apresentados. Para efeito de simplificação, entretanto, essa flexibilidade não é considerada na formalização utilizada, a qual "obriga" a apresentação dos módulos (e dos elementos internos a eles) em uma sequência específica. Essa ordem de apresentação foi tomada como padrão e usada em quase todos os exemplos das seções anteriores.

3. As linhas com comentários (começadas com " ; ") não foram incluídas na BNF.

V - BIBLIOGRAFIA

/DONO-72/: Donovan, J. J.
"Systems Programming"
Computer Science Series - McGraw-Hill Book Co., 1972

/GEAR-74/: Gear, C. W.
"Computer Organization and Programming"
2nd. ed. - Computer Science Series - McGraw-Hill Co., 1974

/GONG-94/: Gong, K. L.
"Berkeley MPEG-1 Video Encoder - User's Guide"
Computer Science Division - University of California - Berkeley, CA, 1994

/MAGA-91/ : Magalhães, L. P. e Silva, M. H.
"ProSIm - Um Sistema para Prototipação e Síntese de Imagens Foto-Realistas e Animação"
Relatório Interno DCA - 030/91 - FEE - UNICAMP

/MALH-94/: Malheiros, M. de G.
Relatório Técnico: *"Uma interface Gráfica para ProSIm"*
FAPESP - 1994

/PRES - 92/: Pressman, R. S.
"Software Engineering - A Practitioner's Approach"
3rd. ed. - McGraw-Hill, Inc., 1992

/RAPO-93/: Raposo, A. B.
"TOOKIMA"
Relatório de Iniciação Científica - CNPq
DCA - FEE - UNICAMP - 1993

/RAPO-96/: Raposo, A. B.
Tese de Mestrado: *"Um Sistema Interativo de Animação no Contexto ProSIm"*
DCA - FEEC - UNICAMP - 1996

/RAPO-97/ Raposo, A. B. e Magalhães, L. P.
"TOOKIMA, uma Ferramenta Cinemática para Animação"
Relatório Interno - 002/97 - DCA - FEE - UNICAMP

/ROGE-85/: Rogers, D. F.
"Procedural Elements for Computer Graphics"
McGraw-Hill Book Co., 1985

/SILV-92/: Silva, M. H.
Tese de Mestrado: *"TOOKIMA: Uma ferramenta para Animação Modelada por Computador"*
DCA - FEE - UNICAMP - 1992

/YNGV-94/: Yngvesson, J. and Wallin, I.
"User's Guide to SIPP - a 3D Rendering Library - Version 3.1"
1994

/YOUN-96/: Young, C. et al..
"Persistence of Vision Ray Tracer (POV-Ray) - User's Documentation 3.0.10"
1996