DEPTO. DE ENG. DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Relatório Técnico
DCA - 002/97

TOOKIMA:
Uma Ferramenta Cinemática para 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 trabalho teve por objetivo principal a elaboração de um manual para a linguagem de mais baixo nível do TOOKIMA (TOOl KIt for scripting computer Modeled Animation). Este relatório pretende ser, ao mesmo tempo, um manual de referência e um guia para o processo de criação de uma animação. Para isso, todas as rotinas do TOOKIMA foram estudadas e testadas.

O manual, apresentado em anexo, está dividido em 6 capítulos, de acordo com os módulos em que se divide o TOOKIMA (*time, script, actor, motion, camera* e *light*). Cada um destes capítulos mostra todas as rotinas contidas nos arquivos do respectivo módulo. O manual também apresenta animações e fragmentos de animações, visando tornar claro o uso das rotinas e a estrutura de uma animação.

O manual termina com apêndices que explicam algumas rotinas de outros módulos do ProSIm (sistema para Prototipação e Síntese de Imagens foto-realistas e animação), que se relacionam diretamente com o TOOKIMA.

I - INTRODUÇÃO:

Computação Gráfica é o termo mais utilizado em português para designar "Computer Graphics", do inglês. Entretanto, é preferível designar a área de atuação deste trabalho como Computação de Imagem (/SANT-89/). Este termo é preferível ao anterior porque traz maior clareza à abrangência da área de pesquisa, uma vez que *Computação* indica o ferramental utilizado e *Imagem* indica o objeto do trabalho computacional.

As atividades em Computação de Imagem iniciaram-se no DCA-FEE-UNICAMP como apoio à area denominada, naquela época (1976), Automação Industrial. Hoje, as atividades nessa área têm também a Arte como alvo, uma vez que ela oferece um rico campo de experimentação.

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

d) Interface: Há uma interface gráfica para o módulo de modelagem /MALH-94/ e outra para o sistema de animação /RAPO-96/.

I.2: Animação:

Animação (/PUEY-88/) é um tema estudado na área de Computação de Imagens. Ela incorpora a noção de movimento e sua reprodução através da superposição rápida e progressiva de "quadros" (imagens), simulando a idéia de tempo transcorrendo.

As ferramentas computacionais podem auxiliar na produção de animações, automatizando algumas das tediosas e longas tarefas da produção de filmes de desenho animado (edição, colorização, etc). Esse tipo de animação é chamada *"Animação Auxiliada por Computador"*.

O tipo de animação referida neste trabalho, no entanto, é a chamada *"Animação Modelada por Computador*", que propõe modelos de representação das entidades gráficas que compõem a cena (modelo geométrico do objeto, modelo para a câmera, modelo para a iluminação, etc). O computador deve armazenar estes modelos e então computar as diferentes ações que podem ser realizadas nestas entidades (isto é, nos modelos).

Os modelos de animação podem ser implementados através de:

* Técnicas Cinemáticas
* Técnicas Dinâmicas

As *técnicas cinemáticas* são usadas principalmente na animação convencional (cartoons) e podem ser divididas em: técnicas de cinemática direta e cinemática inversa. Nas técnicas de cinemática direta o animador especifica o que deve ser apresentado em cada quadro. Quando são utilizadas técnicas de cinemática inversa, o animador especifica as posições inicial e final a serem tomadas pelas entidades em cena, cabendo a algum procedimento o cálculo das interpolações entre tais posições.

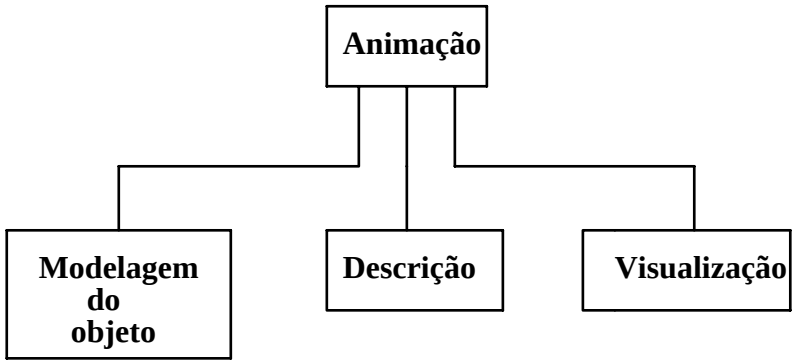
O *modelo dinâmico* (/ISAA-87/) cria animações bem mais realistas utilizando técnicas dinâmicas, as quais podemos desmembrar em:

1. *Dinâmica Direta:* São dadas as forças que agem sobre uma entidade da cena (ator) e observa-se seu comportamento.
2. *Dinâmica Inversa:* São determinadas as restrições físicas do objeto e do ambiente, bem como o estado inicial do sistema objeto-ambiente, cabendo a algum procedimento o cálculo das forças que levam o sistema a obedecer tais restrições.

I.3: TOOKIMA - TOOl KIt for scripting computer Modeled Animation:

O TOOKIMA (/SILV-92/) 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 sub-sistemas, como pode ser visualizado a seguir:

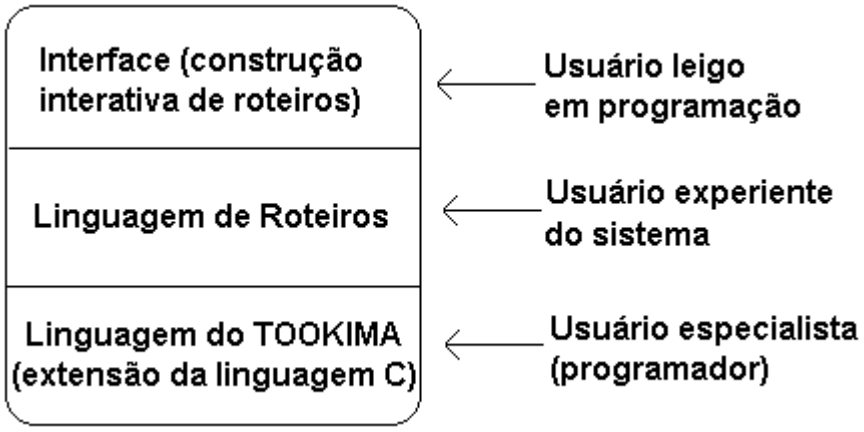


Os sub-sistemas estão integrados no ProSIm.

No estágio atual de desenvolvimento, é possível acessar as seguintes funcionalidades através da interface ProSIm:

- Modelagem:
  - Modelagem Geométrica: definição de objetos por instanciação ou importação de arquivos, remoção de objetos ou parte deles e exportação de objetos em arquivos.
  - Modelagem de Cenas: definição e posicionamento de câmeras e posicionamento de objetos, através da entrada de valores.
  - Modelagem de Propriedades Físicas: definição de cores sólidas para os objetos de cena.
- Descrição de animações: desenvolvimento de um script de animação interativamente; pré-visualização em *wireframe* e MPEG; geração dos quadros em formato PPM ou TGA, para posterior gravação em vídeo.
- *Rendering*: SIPP, que é uma bilbioteca para o *rendering* de cenas tridimensionais, usando um algoritmo de *scanline Z-buffer* com diversos recursos, tais como, *wireframe*, mapeamento de texturas, *anti-aliasing* e sombras /YNGV- 94/.

A linguagem do TOOKIMA para a composição de roteiros de animação (apresentada neste relatório) é uma extensão da linguagem C, provendo tipos de dados e uma biblioteca de funções que permitem o desenvolvimento de animações. Entretanto, a dificuldade que esta linguagem apresenta para usuários leigos em programação levou ao desenvolvimento de uma linguagem de mais alto nível para a composição de roteiros (esta linguagem de roteiros, de mais alto nível, é detalhada em /RAPO-97/). A nova linguagem de roteiros, por sua vez, serviu de base para o desenvolvimento de uma interface, onde o roteiro pode ser construído interativamente (a interface é analisada em /RAPO-96/). A figura a seguir ilustra os possíveis níveis para o desenvolvimento de uma animação utilizando o TOOKIMA, e os usuários característicos de cada um dos níveis.



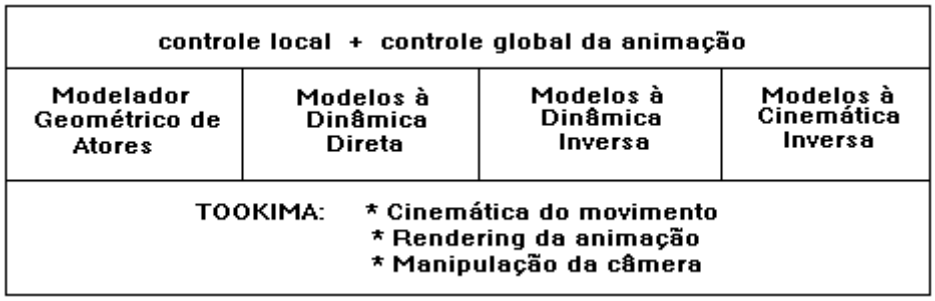
I.4: O futuro do Sistema de Animação do ProSIm:

Pretende-se transformar o Sistema de Animação do ProSIm em um sistema capaz de descrever modelos de cinemática direta e inversa e de dinâmica (também direta e inversa).

Os modelos dinâmicos seriam desenvolvidos em blocos à parte. O modelo à dinâmica direta já está implementado em /RODR-93/. O trabalho voltado à cinemática inversa foi implementado em /CAMA-92/ e /CAMA-95/.

O TOOKIMA passaria a ser então apenas a ferramenta de representação e visualização de movimentos (translada ator, rotaciona ator, move câmera, fade, shade, zoom, etc).

O resultado final desejado é esquematizado em camadas:



II - OBJETIVOS:

O objetivo deste trabalho é o estudo da funcionalidade do TOOKIMA, através do domínio do processo de criação de uma animação. O resultado será expresso na implementação de um manual para a linguagem de mais baixo nível do TOOKIMA. O manual (apresentado em anexo) pretende ser, ao mesmo tempo, um manual de referência e um guia para o processo de criação de uma animação com o TOOKIMA.

O estudo também teve por objetivo o teste de todas as rotinas do TOOKIMA, uma vez que ele foi implementado contendo muitas rotinas que ainda não haviam sido utilizadas. Os aperfeiçoamentos que, durante os testes, se mostraram necessários para as rotinas também foram objetivos deste trabalho.

Pretende-se, com este estudo e com a documentação decorrente, tornar a linguagem do TOOKIMA acessível a qualquer usuário da rede, embora seja o caminho mais "difícil" de se implementar uma animação no ProSIm, já que ele dispõe de uma linguagem de mais alto nível e uma interface gráfica /RAPO-96/, /RAPO-97/.

III - METODOLOGIA:

A primeira parte deste trabalho consistiu no contato inicial com o TOOKIMA. Nesta etapa, procurou-se o entendimento de algumas animações já implementadas (por exemplo, em /RODR-92/).

A etapa seguinte foi a implementação de pequenas animações, para um maior aprofudamento no sistema. Até então não havia metodologia específica, o objetivo era apenas um conhecimento geral do TOOKIMA.

A partir do momento que já havia um conhecimento razoável do sistema, partiu-se para os testes, propriamente ditos. Estes testes eram realizados criando-se pequenas animações que usavam as rotinas que se desejavam testar. Se as rotinas testadas não funcionavam da maneira esperada (descritas em /SILV-92/), se fazia necessário um aprofundamento no algorítmo das mesmas, para seu aperfeiçoamento. As falhas foram encontradas, em sua maior parte, em funções pouco usadas pelo animador e que, por esta razão, haviam sido pouco testadas. Ao final dos testes, todas as falhas encontradas foram corrigidas.

Paralelamente aos testes, o manual era escrito, procurando descrever de forma simples e objetiva a utilização e limitação das rotinas do TOOKIMA (baseado na experiência obtida com os testes e também nas informações da bibliografia).

O resultado deste trabalho é apresentado em anexo (o manual do TOOKIMA). O manual apresenta algumas rotinas que foram aperfeiçoadas neste trabalho; entretanto, elas não estão destacadas no texto (o interesse está na utilização do TOOKIMA revisado, e não em saber o que estava errado anteriormente).

IV - COMENTÁRIOS FINAIS:

O sistema é limitado no que diz respeito à interação entre os atores (um ator não tem conhecimento da existência do outro). Dessa maneira, no sistema, dois ou mais atores podem ocupar o mesmo lugar no espaço, no mesmo instante. Isso pode trazer resultados indesejáveis.

Outra limitação do sistema é relativa à interação usuário-máquina. Como se perceberá pelo manual, o programa é de difícil utilização para animadores não acostumados com a linguagem C e com o sistema UNIX. As interfaces gráficas (respeito /MALH-94/ e /RAPO-96/) permitem que o TOOKIMA tenha um público alvo mais generalizado.

Finalmente, ainda existe a dificuldade da transposição da animação do computador para o cinema ou o vídeo. Ainda não se chegou a uma conclusão sobre qual a melhor maneira de se fazer esta transposição, que depende, sobretudo, dos recursos disponíveis.

Apesar destas limitações (e outras, que aparecerão ao longo do manual), o TOOKIMA é um abrangente sistema para animação modelada por computador.

ANEXO:

Manual da Linguagem do TOOKIMA:

(TOOl KIt for scripting computer Modeled Animation)

INTRODUÇÃO:

Este manual pretende auxiliar no processo de criação de uma animação utilizando o TOOKIMA. Para isso, explica-se o funcionamento e o modo de utilização de todas as rotinas do programa. Também são apresentadas algumas animações-exemplo.

Recomenda-se que o animador tenha o mínimo de conhecimento do sistema UNIX e da linguagem C (ou alguma linguagem de programação). Entretanto, o manual foi escrito da maneira menos técnica possível, numa tentativa de tornar o programa acessível a qualquer usuário.

O TOOKIMA é um conjunto de ferramentas para descrição algorítmica de animações de objetos modelados por computador. O TOOKIMA foi desenvolvido no LCA (Laboratório de Engenharia da Computação e Automação Industrial) do GRUPO DE COMPUTAÇÃO DE IMAGENS - DCA - FEEC - UNICAMP, como tese de mestrado /SILV-92/.

Na verdade, um sistema de animação é um conjunto que integra 3 sub-sistemas:

- *Modelador Geométrico*: sub-sistema responsável pela criação de objetos que servirão de "atores" na animação.
- *Sistema de Visualização (rendering)*: responsável pelo processamento final da imagem, após a definição da geometria da cena e o acréscimo de parâmetros para sua visualização final.
- *Sistema de Descrição e Controle da Animação*: quem realiza estas tarefas é o TOOKIMA, propriamente dito. É especificamente para esta parte do sistema que este manual foi feito.

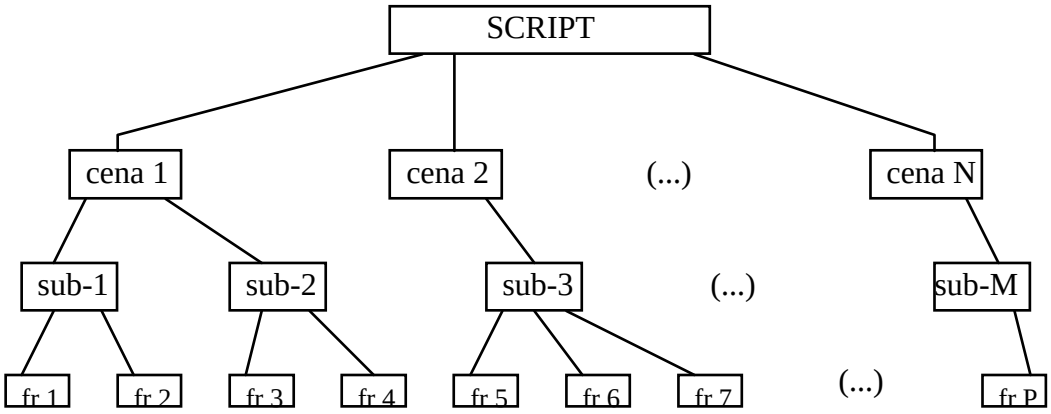
O TOOKIMA está integrado ao projeto ProSIm (sistema para *Pro*totipação e *S*íntese de *Im*agens foto-realistas e animação). O TOOKIMA integra a parte de animação do projeto (mais especificamente, a parte de animação baseada no modelo de cinemática direta).

Na criação de uma animação com o TOOKIMA, são utilizados alguns conceitos importantes:

Script (roteiro) - é um programa principal que gerencia as cenas, sendo responsável pelo macro controle de animação. Utilizando uma hierarquia (árvore) para representar o processo de descrição de uma animação, o script seria a raiz (contém uma visão geral da animação, de forma sucinta).

Cena - desdobramento do script, que descreve a animação com maior detalhamento. Os desdobramentos das cenas serão chamados sub-cenas, aumentando cada vez mais o nível de detalhes...

Frames - são as folhas da árvore, onde não são mencionadas as mudanças mas, mostradas quadro a quadro. A figura a seguir ilustra a estrutura de uma animação.



"Os níveis script, cena e frame são níveis obrigatórios que descrevem uma animação no TOOKIMA." /SILV-92/

Cue - outro conceito muito utilizado. Cue é um intervalo de tempo associado a uma cena (intervalo de tempo em que a cena acontecerá). "Cada cena está associada a uma cue e a cada cue estão associadas uma ou mais cenas". Cada cue possui dois valores que identificam o instante em que a cena começa (cue_start) e o instante em que ela termina (cue_stop).

Relógio Absoluto (Tglobal) - varia de 0 a TotalTime (variável definida no script; é o tempo total da animação), com passo de 1/Rate (Rate é a taxa de quadros por segundo de animação, também é definida no Script).

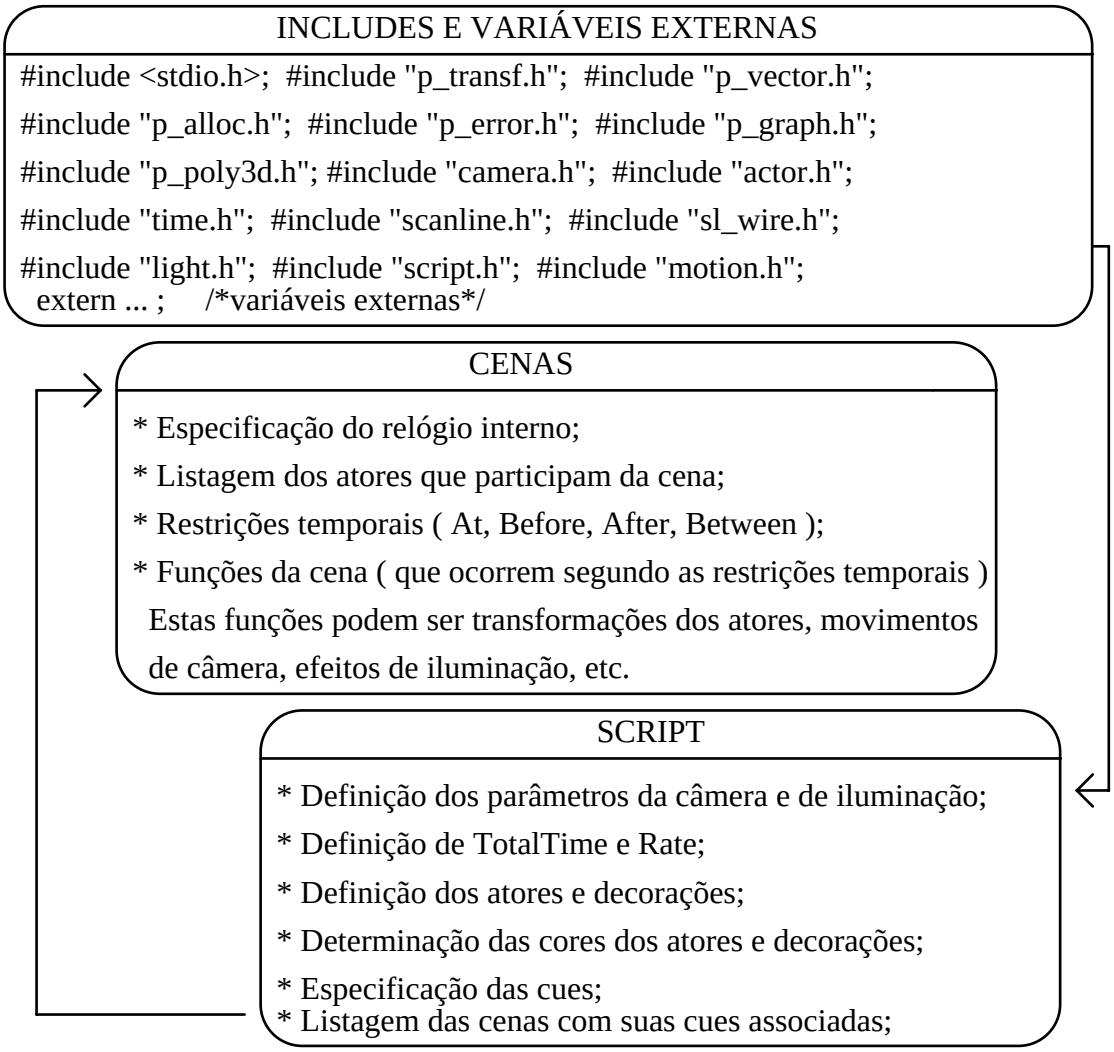
Relógio Local (Tlocal) - relógio determinado internamente à cena utilizando o comando CLOCK no começo da descrição da cena (ver módulo time.h).

Ator x Decoração - um objeto definido como ator, pode sofrer as transformações permitidas (rotação, translação, escalamento, etc), enquanto que a decoração não pode ter nenhuma movimentação. A decoração é limitada a objetos poliédricos, enquanto que o ator pode também ser um ponto, uma cor, uma face, etc. (ver módulo actor.h).

Camera Sintética - permite efeitos semelhantes aos obtidos com uma câmera de cinema (zoom, traveling, etc.). Maiores detalhes serão dados no módulo camera.h.

Iluminação - permite a criação de fontes luminosas de três tipos: pontual (situada em um ponto e emitindo luz em todas as direções), spot (situada em um ponto e emitindo luz em uma direção especificada) e sol (emite raios paralelos). Permite também o efeito de fade-in e fade-out.

Com o conhecimento dos conceitos acima explicados, pode-se criar um quadro com a estrutura de uma animação:



Este manual está dividido em 6 capítulos, de acordo com os arquivos (módulos) .h do TOOKIMA. Os arquivos .h definem estruturas e macros que serão usadas em funções contidas nos arquivos .c. Cada módulo .h possui um conjunto de arquivos .c, que se inicia com a primeira letra do nome do módulo (por exemplo, o módulo actor.h possui os arquivos a_define.c, a_rotate.c, e assim por diante).

Cada capítulo está dividido em sub-seções que tratam separadamente das estruturas, das macros e das funções do módulo. A sub-seção que trata das funções, também está dividida de acordo com os arquivos .c do módulo.

No final do manual, existem dois apêndices que apresentam rotinas do ProSIm e do Scanline[1] (apêndices A e B, respectivamente). Estas rotinas são apresentadas porque são importantes no processo de criação de uma animação (por ser um sistema interdependente, muitas vezes são usadas nas animações funções do ProSIm - e não do TOOKIMA).

[1] O Scanline (/SILV-90/, /PRET-91/ e /PRET-93/) era a ferramenta de rendering inicialmente utilizada pelo TOOKIMA. Para manter a completude deste manual, optou-se por manter a descrição destas funções, mesmo não sendo elas mais utilizadas.

1 - MÓDULO TIME.H:

O módulo time define o relógio local de uma cena da animação (estrutura que determinará a duração das cenas e servirá como referência para a execução de funções de outros módulos - isto é, as funções serão realizadas em determinado tempo do relógio). Este módulo também define uma estrutura chamada "cue", que descreve a localização temporal das cenas (seu começo e seu término). A cada cena deve estar associada pelo menos uma cue. Neste texto, muitas vezes usa-se cue e cena como sinônimos, embora exista uma diferença em termos de algorítmo.

As funções defindas neste módulo tratam unicamente do tempo (relógio) e da manipulação de cues (geometria temporal).

1.1 - Estruturas do time.h:

Cue - a estrutura Cue tem apenas dois parâmetros reais (double): start e stop.

Da listagem do programa:

```
typedef struct { double start, stop; } Cue;
```

Watch - (relógio); possui 5 parâmetros reais (double):

Da listagem do programa:

```
typedef struct { double    start,
                           stop,
                           inv_dura,      /* inverso da duração */
                           t_local,       /* tempo local convertido
                                             para a dimensão temporal
                                             (relógio) da cena */
                           d_time;        /* equivalente a um Tick do
                                             relógio global, convertido
                                             para o relógio local */
               } Watch;
```

1.2 - Macros do time.h:

Tick - é a "bateria" do relógio. Acrescenta uma unidade de tempo ao Tglobal (tempo global da animação). A unidade de tempo acrescentada é 1/Rate, onde Rate é o número de quadros (frames) por segundo da animação (o valor de Rate deve ser definido no script da animação). Esta macro é acionada "automaticamente" pelo programa com o comando LIST_SCENES (ver módulo script.h), não sendo necessária sua utilização pelo animador. Seu equivalente no relógio da cena é o d_time.

SET_CUE (cue_in, início, fim) - esta macro tem como parâmetros: cue_in, o tempo de início e o tempo de término da mesma. Cria-se a estrutura Cue, chamada cue_in, e colocam-se os dois parâmetros seguintes como cue_in.start e cue_in.stop, respectivamente. Esta macro deve ser usada no script da animação (ver módulo script.h), semelhante a macros de inicialização do ProSIm (SET_POINT, SET_COLOR, etc - ver apêndice A).

Exemplo: `SET_CUE ( cue[1], 0.0, 2.5 );`

Isso equivale a definir a variável cue[1] como sendo uma estrutura do tipo Cue e fazer: `cue[1].start = 0.0;`
`cue[1].stop = 2.5;` indicando que qualquer cena relacionada à cue[1] vai ser realizada entre 0 e 2.5 segundos da animação.

"Cada cena deve estar associada a pelo menos uma cue e, a cada cue estão associadas uma ou mais cenas" /SILV-92/. Portanto, antes de se listar as cenas e suas cues associadas no script (ver módulo script.h), deve-se definir a cue com SET_CUE ou com alguma das próximas SET_CUE_... .

SET_CUE_COND (cue_in, fim) - esta macro cria cues condicionais, que estão associadas a cenas condicionadas, ou seja, cenas cujas atividades são determinadas por outras cenas. A única diferença entre SET_CUE e SET_CUE_COND, é que a última não define o tempo de início da cue, fazendo necessário uma função para ativá-la dentro de outra cena (ver Active_ccue e Active_ccue_until, neste módulo). Por isso, ela tem apenas 2 parâretros: cue_in e fim. (Na verdade é dado um valor maior para ccue.start, de modo que ccue.start > ccue.stop, fazendo com que a ccue não seja ativada até que haja uma mudança no ccue.start através das funções Active_ccue ou Active_ccue_until.)

Exemplo: `SET_CUE_COND ( ccue[3], 1.0 );` cria uma ccue[3], que não é acionada até o uso de Active_ccue ou Active_ccue_until e terminará no tempo 1.0, a não ser que se use Kill_ccue ou Kill_ccue_at (ver estas funções neste módulo).

Obs.: Deve-se fazer um vetor ccue[ ] para cues condicionais, enquanto que para cues normais e/ou cíclicas faz-se o vetor cue[ ].

SET_CUE_CICLIC (cue_in, início, fim) - cria cues cíclicas, que vão estar associadas a cenas cíclicas (cenas que, uma vez iniciadas, se repetem indefinidamente com o período estabelecido na cue cíclica). Em termos de programação, é idêntica a SET_CUE, tendo inclusive os mesmos parâmetros; mas o tempo de início e tempo de término da cue vão definir o período com o qual a cena cíclica associada a ela vai se repetir. Entretanto, diferentemente de SET_CUE, a cena só vai começar no frame seguinte ao do início da cue cíclica. Por exemplo:

`SET_CUE ( cue[2], FRAME(4), FRAME(8) )`, faz com que a cena relacionada a ela comece a ser passada ao sub-sistema de rendering (geração de imagens) a partir do frame (4).

`SET_CUE_CICLIC ( cue[2], FRAME (4), FRAME (8) )`, a cena relacionada a ela começa a ser passada ao sub-sistema de rendering a partir do frame (5).

Active_ccue (N) - ativa imediatamente uma cue condicional (ativação simples), dentro de uma outra cena. O único parâmetro é um número inteiro N, que determina a ccue[N] a ser ativada. A ccue[N] deve ter sido definida como condicional (SET_CUE_COND).

Exemplo: `Active_ccue ( 3 );`
É equivalente a: `ccue[3]->start = Tglobal;`

Active_ccue_until (N, T) - difere da anterior porque determina o momento em que a cena condicional deve ser desativada. Tem 2 parâmetros: um inteiro N (que determina a ccue[N] a ser ativada) e um real T (instante em que a ccue será desativada). Novamente, ccue [N] deve ter sido definida como condicional (SET_CUE_COND).

Exemplo: Active_ccue_until (3, 4.0);
É equivalente a: ccue[3]->start = Tglobal;
ccue[3]->stop = 4.0;

Kill_ccue (N) - macro que deve ser usada para cenas condicionais. Desativa a cena imediatamente (isto é, no frame seguinte ela já está desativada). O único parâmetro é o inteiro N, que identifica a ccue a ser desativada.

Exemplo: Kill_ccue (3);
Equivale a: ccue[3]->stop = Tglobal.

Kill_ccue_at (N, T) - A única diferença com relação a Kill_ccue é que Kill_ccue_at não desativa a ccue (condicional) imediatamente, mas no tempo T (no frame seguinte ao do tempo T, a cena já está desativada). T é o segundo parâmetro da função.

Exemplo: Kill_ccue_at (3, 4.0);
Equivale a: ccue[3]->stop = 4.0.

Kill_cue - Desativa a cena corrente imediatamente (sem a necessidade de ser uma cena condicional); no frame seguinte ela já está desativada. Não possui parâmetros.

Exemplo: Kill_cue;
Equivale a: cue->stop = Tglobal.

Kill_cue_at (T) - A única diferença com relação a Kill_cue é que Kill_cue_at não desativa a cena corrente imediatamente, mas no tempo T (no frame seguinte ao do tempo T, a cena já está desativada).

Exemplo: Kill_cue_at (3.5);
Equivale a: cue->stop = 3.5

CLOCK (dura) - macro que deve ser usada em cada cena, para definir o relógio da mesma. Tem um único parâmetro real, que define a duração do relógio local da cena.

Exemplo: CLOCK (dura);
Equivale a:

```
clock.start = cue.start;
clock.stop = cue.stop;
clock.inv_dura = 1/dura;
clock.t_local = [(Tglobal - cue.start)*dura] /
                    (cue.stop - cue.start);
clock.d_time = [dura / (cue.stop - cue.start)]
        * ( 1/Rate);
```

A variável clock.t_local é um relógio interno à cena, que varia de 0 a "dura", incrementado em unidade dada por clock.d_time.

NO_CLOCK - equivalente a se fazer CLOCK (cue.stop - cue.start). Esta macro foi criada porque o relógio fica mais simples de entender usando-se "dura" como sendo o tamanho da cue (cue.stop - cue.start).

Exemplo: NO_CLOCK;
Equivale a:

```
clock.start = cue.start;
clock.stop = cue.stop;
clock.t_local = Tglobal - cue.start;
clock.inv_dura = 1/(cue.stop - cue.start);
clock.d_time = 1/Rate;
```

Tlocal - macro definida como sendo igual ao clock.t_local. Esta variável vai ser muito usada em outros módulos. Tlocal é o tempo contado a partir de cue.start em unidades iguais à unidade do Tglobal (1/Rate) multiplicada por dura/(cue.stop - cue.start), onde "dura" é o parâmetro da função CLOCK. Repare que, usando-se NO_CLOCK, o Tlocal tem a mesma unidade do Tglobal. Tlocal varia de 0 a "dura" numa cena.

2 - MÓDULO SCRIPT.H:

Este módulo contém funções relativas à organização da animação como um programa em linguagem C. Suas funções variam desde funções temporais (criando loops condicionais com o tempo de relógio) a outras mais complexas, responsáveis pelo "rendering" (visualização) da imagem gerada. Este módulo torna clara a relação entre algumas necessidades do roteiro e a implementação das mesmas usando recursos da linguagem C.

2.1: Estruturas do script.h:

Neste módulo, nenhuma nova estrutura é definida.

2.2: Macros definidas em script.h:

TOTAL (n) - tem um único parâmetro n, que vai ser multiplicado por uma função linear do clock (TOTAL (n) é equivalente a n*clock.d_time*clock.inv_dura, elementos da estrutura relógio) - ver função linear( ) no arquivo m_dynamic.c, do módulo motion.h e estrutura Watch do módulo time.h.

FRAME (n) - Seu objetivo é converter um inteiro n, referente ao n-ésimo quadro da animação, para tempo absoluto (Tglobal). Esta conversão é feita através da definição: FRAME(n) = n./Rate. Desta maneira, por exemplo, FRAME(Rate) é equivalente a 1 segundo. Sempre que se mudar o valor de Rate, deve-se mudar os valores dos FRAME's, para manter a animação inalterada no tempo. Por outro lado, é bom evitar manipular Rate após o uso de declarações com FRAME.

At (local_t) - Cria um loop condicional ("if") que só permitirá a execução de funções se Tlocal (ver time.h) for *igual* a local_t. O parâmetro local_t pode ser dado em tempo absoluto (double) ou em FRAME (n), que vai ser convertido em double, como visto anteriormente. Deve ser usado nas cenas, e não no script.

Exemplo (fragmento de uma animação):

```
Rate = 24;
        .
        .
        .
At (FRAME(48))   /*Equivalente a At (2.0)*/
{
        função 1 ( ); função 2 ( );
}
```

Neste exemplo, as funções 1 e 2 vão ser executadas apenas no 48º quadro (no tempo = 2 segundos, já que Rate é 24).

After (t) - cria loop condicional, cuja condição é que Tlocal seja *maior que* t (as funções internas a esse loop só se realizarão após o tempo t). Novamente t pode ser tempo absoluto ou FRAME (n).

Exemplo (fragmentos de uma animação):

```
Rate = 24;
        .
        .
```

```
.
After (FRAME(48))   /*Equivalente a After (2.0)*/
{
        função 1 ( ); função 2 ( );
}
```

Neste exemplo, as funções 1 e 2 vão ser executadas apenas a partir do 49º quadro (ou para Tlocal > 2.0).

Um detalhe importante é que, a partir do 49º quadro, as funções 1 e 2 serão executadas a cada quadro e não uma vez só até o final da cena. Assim, se a função 1 for, por exemplo, uma função de translação e se deseja que o ator se desloque 3 unidades de distância até o final da cena, o parâmetro relativo ao deslocamento do ator (na função 1) deve ser 3/(número de quadros restantes até o final da cena). Dessa maneira, ao final da cena, o ator terá se deslocado 3 unidades. Se o parâmetro fosse 3, no 49º quadro o ator se deslocaria 3 unidades, no 50º quadro se deslocaria mais 3 e assim sucessivamente.

Before (t) - cria loop condicional, cuja condição é que Tlocal seja *menor que* t (as funções internas a esse loop só se realizarão antes do tempo t). Novamente t pode ser tempo absoluto ou FRAME (n).

Exemplo (fragmentos de uma animação):

```
Rate = 24;
        .
        .
        .
Before (FRAME(48))   /*Equivalente a Before (2.0)*/
{
        função 1 ( ); função 2 ( );
}
```

Neste exemplo, as funções 1 e 2 vão ser executadas apenas até o 47º quadro (ou para Tlocal < 2.0).

A mesma observação quanto aos parâmetros das funções internas ao loop After são válidas para o Before (as funções também serão executadas uma vez a cada quadro).

Between (t1,t2) - cria loop condicional, cuja condição é que Tlocal seja *maior ou igual a t1 e menor ou igual a t2* (as funções internas a esse loop só se realizarão após o tempo t1 e antes de t2). Novamente t1 e t2 podem ser tempo absoluto ou FRAME (n).

Exemplo (fragmento de uma animação):

```
Rate = 36;
        .
        .
        .
Between (FRAME(54),FRAME(72))
                /*Equivalente a Between (1.5,2.0)*/
{
        função 1 ( ); função 2 ( );
}
```

Neste exemplo, as funções 1 e 2 vão ser executadas apenas a partir do quadro 55 (ou para Tlocal > 1.5), até o quadro 71 (ou Tlocal <2.0).

A mesma observação quanto aos parâmetros das funções internas aos loops After e Before são válidas para o Between (as funções internas ao loop serão executadas uma vez a cada quadro).

RenderInterval (t1, t2, tj) - esta função contribui para agilizar os testes na elaboração de uma animação, controlando qual pedaço da animação será passado ao sub-sistema de "rendering" (o "pedaço da animação", no caso, são todas as cenas ativas no período entre t1 e t2). O parâmtero tj indica de quantos em quantos quadros será feita a geração de imagens (por exemplo, se tj for 5, só serão gerados os quadros múltiplos de 5, entre t1 e t2). O sub-sistema de "rendering" é o responsável pela geração das imagens. Usada no Script.

Exemplo: `RenderInterval ( FRAME(0), FRAME(9), 1 );`

Neste exemplo, só são gerados os 10 primeiros quadros da animação (tj = 1 indica que todos os quadros no intervalo serão gerados).

BEGIN_SCENE - usada uma vez em cada cena. É um loop condicional que só vai permitir que a cena (funções internas a este loop) ocorra no intervalo estabelecido pela cue relativa à cena (ver LIST_SCENES neste módulo). Esta macro também invoca as variáveis externas Tglobal (double) e Rate (inteiro), que serão usadas no decorrer da cena. Estas variáveis são internas ao script (ver BEGIN_SCRIPT neste módulo).

END_SCENE - deve ser usada no final de uma cena, indicando o final da mesma.

Exemplo:

```
void cena1 (cue)
Cue *cue;            /*Definição da função cena1, que tem como
                      parâmetro um apontador para uma Cue */
{
 BEGIN_SCENE   /*Equivalente a:
                se (cue.start < = Tglobal <=cue.stop),
                faça ... */
 NO_CLOCK;     /*Após o BEGIN_SCENE é
                necessário definir o relógio.*/
        .
        .      /*Funções da cena*/
        .
 END_SCENE     /*Final de cena*/
}
```

BEGIN_COND_SCENE - quando a cena for condicional, deve-se usar esta macro, ao invés de BEGIN_SCENE. Sua utilização é idêntica à anterior.

END_COND_SCENE - deve ser usada no final de uma cena condicional, indicando o final da mesma.

BEGIN_CICLIC_SCENE - quando a cena for cíclica, deve-se usar esta macro, ao invés de BEGIN_SCENE. Sua utilização é idêntica à anterior.

END_CICLIC_SCENE - quando a cena for cíclica, deve-se usar esta macro, ao invés de END_SCENE. É esta macro que permite que a cena cíclica seja repetida continuamente, pois, em Tlocal = (cue.stop - cue.start), ela faz com que cue.start seja igual a Tlocal e cue.stop seja (Tlocal + tamanho da cue). Em outras palavras, ao terminar uma cue cíclica, ela faz com que a cue seja recomeçada a partir daí.

Exemplo:

```
void cena1 (cue)
Cue *cue;        /*Definição da função cena1, que tem
                 como parâmetro um apontador para
                 Cue */
```

```
{
 BEGIN_CICLIC_SCENE
                    /*Equivalente a:
                    se (cue.start < = Tglobal <=cue.stop),
                    faça ... */
 NO_CLOCK;     /*Após o BEGIN_CICLIC_SCENE
                    é necessário definir o relógio.*/
          .
          .          /*Funções da cena cíclica.*/
          .
 END_CICLIC_SCENE
                    /*Equivale a fazer: se
                    Tlocal = (cue.stop - cue.start), então:
                          {aux=cue.stop - cue.start;
                           cue.start = cue.stop;
                           cue.stop = aux + cue.stop; }*/
}
```

BEGIN_SCRIPT (n) - Após a definição de todas as cenas (funções), é necessário fazer um script, que é o equivalente à função main( ) da linguagem C. No script devem ser definidos os parâmetros da câmera (ver módulo camera.h), os atores (ver módulo actor.h), a iluminação (ver light.h), outros parâmetros como Rate (Rate tem default = 24) e TotalTime (tempo total da animação), além de funções do próprio módulo script.h (ver arquivo s_define.c, neste módulo). Dentro do script também são chamadas as cenas com suas respectivas cues (ver LIST_SCENES, neste módulo).

Nesta macro são declaradas as seguintes variáveis externas: um vetor de caracteres chamado picture_name (usado na função set_studio do arquivo s_define.c deste módulo) e um inteiro chamado fr_counter (vai contar o número de frames). Ainda são declaradas as variáveis internas Tglobal, TotalTime, InitTime, Rate e os vetores cue[ ] (para cues normais e cíclicas) e ccue[ ] (para cenas condicionais).

O parâmetro n desta macro é o número de cenas da animação. Este valor é necessário pois BEGIN_SCRIPT (n) cria dois vetores estáticos de Cue's com n elementos cada (cue[ ] e ccue[ ]). Assim, cada cena normal ou cíclica pode estar associada a uma cue diferente (cue[0], cue[1], ...) e cada cena condicional pode estar associada a uma ccue (ccue[0], ccue[1], ...).

END_SCRIPT - Macro que indica fim do script. Usa a função p_free_all ( ) do ProSIm (ver apêndice A), desalocando todos os ponteiros alocados.

LIST_SCENES - Esta macro é responsável pela criação de um "loop" ("for") que insistentemente passará o controle para as cenas listadas sequencialmente. A cada iteração, o relógio é incrementado (Tick - ver módulo time.h), até o final do tempo destinado à animação (TotalTime).

END_LIST - Ao final da lista de cenas é usada esta macro. Ela vai chamar a função shoot(), responsável pelo "rendering" dos frames (ver arquivo s_define.c neste módulo), caso a animação esteja no seu intervalo de rendering (RenderInterval) e também a função do_mpeg(), responsável pela geração da animação MPEG a partir dos frames (ver s_define.c). Esta macro também é responsável pelo incremento do fr_counter (contador de frames).

Exemplo: BEGIN_SCRIPT(3) /*script com três cenas*/

```
        .
        .                   /*funções relativas à câmera, aos
        .                   atores, etc.*/
TotalTime = 4;
Rate = 30;                  /*Esta animação vai ter 120 frames,
                            já que se tem 4 segundos de anima-
                            ção a uma taxa de 30 frames/s*/
SET_CUE (cue[0],FRAME(0),FRAME(60));
SET_CUE (cue[1],2.0,4.0);
                    /*Estabeleceram-se os limites das duas cues: a
                    primeira ocorre nos primeiros 2 segundos e a
                    segunda, nos dois segundos finais*/
SET_CUE_COND (ccue[0],4.0);
                    /*Define-se a cue condicional com final no tempo
                    4.0 segundos*/
LIST_SCENES         /*Listagem das cenas com suas cues
                    associadas.*/
        cena0 (cue[0]);
        cena1 (cue[1]);
        cena2 (ccue[0]);
                    /*Cenas devem ser definidas antes do script*/
END_LIST            /*Fim da listagem*/
END_SCRIPT          /*Fim do script*/
```

OBS.:

1. As macros CLOCK, NO_CLOCK, TOTAL, At, After, Before, Between, Active_ccue..., Kill_ccue..., Kill_cue..., BEGIN_SCENE, BEGIN_CICLIC_SCENE, BEGIN_COND_SCENE, END_CICLIC_SCENE, END_COND_SCENE e END_SCENE, devem ser usadas dentro da definição de uma cena.

2. As macros FRAME, LIST_SCENES, END_LIST, BEGIN_SCRIPT, END_SCRIPT, RenderInterval e SET_CUE... devem ser usadas na definição do roteiro (script), e não na cena.

2.3: Funções do script.h:

As funções definidas neste módulo estão nos arquivos s_define.c e s_render.c:

2.3.1: Arquivo s_define.c:

set_studio (String *, int, String *) - função que deve constar obrigatoriamente no script. Determina a inicialização da câmera (ver módulo camera.h) e da iluminação (ver módulo light.h).

O primeiro parâmetro é o "path" do arquivo de configuração (como toda String, deve vir entre aspas). A função então chama a função "read_cfg", que lê o arquivo. Esta função será vista logo a seguir. Existe a opção (mais comumente utilizada) de usar o primeiro parâmetro como NULL. Assim, a animação terá uma configuração default:

- *configuração da câmera*: a configuração default da câmera é dada pela função "set_cam_defaults" do arquivo c_define.c, módulo camera.h.

- *configuração default da iluminação*:

cor de fundo: é (36000., 53000., 60000.). Segundo a convenção do ProSIm (os três valores são respectivamente as componentes *r* - red -, *g* - green - e *b* - blue - da cor; estes valores variam de 0 a 60000), a cor especificada é azul-claro;

luz ambiente: é branca (60000., 60000., 60000.) e o meio tem índice de refração 1;

O segundo parâmetro é um inteiro que definirá a qualidade de visualização (as qualidades possíveis serão vistos na função "shoot", neste módulo).

O último parâmetro será o "path" da imagem que se deseja gerar.

Após a determinação da configuração, são chamadas as seguintes funções: "border_viewport" e "camera" (ambas do módulo camera.h) e "p_init_graph" do ProSIm (ver apêndice A).

Exemplos:

1. set_studio ("../config.in", 6, "../images/scan/imag");
   /* A configuração está no arquivo ../config.in; as imagens se chamarão imag e estarão no "path" determinado. */
2. set_studio (NULL, 6, "../scan/anim1");
   /* Usa-se a configuração default e as imagens geradas se chamarão anim1 e estarão no "path" determinado. */

read_cfg (String *) - não é usada diretamente pelo animador. É chamada internamente a "set_studio", quando é especificado o nome de um arquivo. O parâmetro é o nome do arquivo a ser lido. O arquivo de configuração deve ter a seguinte forma:

```
obs_x  obs_y  obs_z
coi_x  coi_y  coy_z
vup_x vup_y vup_z
d
u_min  u_max  v_min  v_max
x_corner  x_length  y_corner  y_length
proj
back_r  back_g   back_b
amb_r  amb_g  amb_b
air_index
n_lights
(...)
```

Onde:

obs_... - coordenadas x, y e z da posição do observador. São dadas em valores reais (double);

coi_... - coordenadas x, y e z do centro de interesse (ponto para onde o observador olha). São dadas em valores reais ;

vup_... - coordenadas de vup (vetor que indica a direção do que é para cima - ver camera.h). São dadas em valores reais (double);

d - distância focal da camera. É um valor inteiro.

u_min, u_max, v_min e v_max - definem a janela (parâmetro de enquadramento). São valores reais. (Ver camera.h);

x_corner, x_length, y_corner, y_length - definem a "viewport" (outro parâmetro de visualização). São valores inteiros, dados em coordenadas da tela do dispositivo (DC). (Ver camera.h);

proj - inteiro que estabelece o tipo de projeção desejada: 0, se for ortogonal e 1, se for perspectiva;

back_... - valores reais que dão as coordenadas r, g e b da cor de fundo;

amb_... - valores reais que dão as coordenadas r, g e b da luz ambiente;

air_index - valor real que especifica o índice de refração do ambiente;

n_lights - valor inteiro que estabelece o número de fontes luminosas especificadas;

(...) - dependendo do valor de n_lights, deve-se especificar as fontes. O primeiro valor de cada fonte é um inteiro que identifica o tipo de fonte (0, se for do tipo lâmpada; 1, se for do tipo sol e 2, se for do tipo spot). A seguir, são colocadas as três coordenadas da cor da fonte (valores reais). Finalmente, dependendo do tipo de fonte, são colocados outros valores reais que a caracterizam: se for lâmpada, deve-se colocar as 3 coordenadas de sua posição; se for sol, deve-se colocar as 3 coordenadas da direção dos raios luminosos; se for spot, deve-se colocar primeiro as coordenadas da posição, depois as da direção e por último, um parâmetro associado ao ângulo de espalhamento (ver light.h).

Exemplo de arquivo de configuração:

```
4. 0. 0.                /* observador no ponto ( 4, 0, 0 ) */
0. 1. 0.                /* centro de interesse = ( 0, 1, 0 ) */
0. 1. 0.                /* vup = direção y, dada pelo vetor ( 0, 1, 0 ) */
1.                      /* distância focal = 1 */
-0.7 0.7 -0.7 0.7       /* extremidades da janela */
5 120 5 120             /* extremidades da viewport, em DC */
1                       /* indica projeção perspectiva */
60000. 50000. 0.        /* cor de fundo */
40000. 60000. 60000.    /* luz ambiente */
1.                      /* índice de refração do ar */
3                       /* número de fontes de luz */
0                       /* a primeira fonte é do tipo lâmpada */
60000. 60000. 60000.    /* cor da fonte */
3. 3. 3.                /* posição da lâmpada */
1                       /* a segunda fonte é do tipo sol */
0. 60000. 0.            /* cor da fonte */
0. -1. 0.               /* direção dos raios luminosos */
2                       /* a terceira fonte é do tipo spot */
60000. 50000. 0.        /* cor da fonte */
4. 6. -2.               /* posição do spot */
0. -1. -1.              /* direção do spot */
5.                      /* parâmetro do ângulo de espalhamento */
```

OBS.: No arquivo de configuração não pode ser colocado nenhum tipo de comentário. Eles foram colocados no exemplo acima apenas para facilitar o entendimento do mesmo.

write_cfg (String *) - faz o contrário da função anterior: coloca a configuração da animação em um arquivo, cujo "path" é o parâmetro da função. O arquivo a ser escrito terá a mesma forma do arquivo lido na função "read_cfg".

apply_transformations ( ) - função sem parâmetros e que não é usada diretamente pelo animador, pois é chamada internamente à função "shoot". É esta função que "aplica as transformações acumuladas pelas cenas aos atores devidos" /SILV-92/.

prepar_to_render ( ) - função que prepara os dados para o "rendering" (geração das imagens). Não possui parâmetros. É chamada internamente à função "shoot".

reset_all_activities ( ) - "desativa todos os elementos da cena para o próximo instante"./SILV-92/. Também não possui parâmetros e é chamada internamente, ao final da função "shoot".

list_act_actors ( ) - lista todos os atores ativos em um instante. É chamada internamente à função "shoot".

list_act_decors ( ) - lista todos as decorações ativas em um instante. É chamada internamente à função "shoot".

list_act_lights ( ) - lista todas as características (cor, posição, etc) das fontes de luz ativas em um instante.

shoot ( ) - função que é chamada com a macro END_LIST. É a função que chama o "rendering" . Inicialmente, ela chama as funções "list_act_..." e a função "camera_list" (módulo camera.h). Depois chama a função "aplly_transformations". Então, ela executa a visualização, segundo a qualidade especificada pelo segundo parâmetro da função "set_studio":

Se ele for *0*: há só o "debug" do programa, sem geração de imagens.

Se for *1*: utiliza o SIPP (renderizador), e os quadros são gerados no formato PPM.

Se for *2*: usa o SIPP, e são gerados os quadros PPM e a animação MPEG (chama a função “do_mpeg”, vista a seguir).

Se for *3*: usa o SIPP, e são gerados quadros no formato TGA não-comprimido.

Se for *5, 6, 7* ou *8*: há a geração de imagens pelo Scanline (desatualizados!).

Na prática, pode-se dizer que só são usadas as qualidades de 0 (quando não se deseja ver a imagem, mas apenas detectar erros no programa da animação) a 3.

do_mpeg( ) - gera a animação MPEG utilizando o “Berkeley's MPEG Encoder” /GONG-94/, quando a qualidade for 2. Esta função necessita de um arquivo de parâmetros (extensão .par) no mesmo diretório que a animação e com o mesmo nome. Quando se usa a interface gráfica do TOOKIMA, tudo isso é feito automaticamente.

2.3.2: Arquivo s_render.c:

write_brp (Polyhedron *pol, char arq_name[NAME_LENGTH]) - função que escreve um arquivo .brp a partir de uma estrutura poliédrica (primeiro parâmetro). O segundo parâmetro é o nome do arquivo a ser escrito.

write_envelope_brp (Polyhedron *pol, char arq_name[NAME_LENGTH]) - semelhante à anterior, com a diferença de que será escrito no arquivo .brp o envelope convexo 3D da estrutura poliédrica (menor parelelepípedo que engloba o poliedro).

script_scanline (char arquivo[NAME_LENGTH]) - escreve num arquivo um script para ser executado pelo scanline. Este script será gerado em cada quadro da animação e será dado como entrada para o renderizador.

script_scanline (char arquivo[NAME_LENGTH], int quality) - semelhante à função anterior, mas escreve no arquivo um script para ser executado pelo SIPP. Através do código desta função é possível entender o formato deste script. O parâmetro "quality" é o mesmo de função shoot(), definindo a qualidade dos quadros gerados.

OBS.: As funções aqui apresentadas utilizam estruturas que serão definidas em módulos ainda não apresentados neste texto. Portanto, algumas delas podem ter ficado obscuras para o leitor. Os capítulos seguintes as tornarão mais claras.

3 - MÓDULO ACTOR.H:

O módulo actor.h define estruturas relacionadas com os atores e decorações da cena. Pela definição em /SILV-92/ : ator é "o elemento componente da cena que tem expressividade e movimento" e decoração é "o elemento componente da cena que não tem movimento".

O TOOKIMA dispõe de vários tipos de atores, divididos em duas classes: ator *referência* e ator *terminal*:

Um ator referência pode ser: DOUBLE (valor real usado, por exemplo, como um ângulo entre braço e antebraço de um ator); POINT (ponto com coordenadas retangulares - x y z - , podendo servir de referência para diversas transformações ou podendo ser parte de um outro ator terminal - ponto de um poliedro, por exemplo); VECTOR (vetor com coordenadas direcionais retangulares, podendo, entre outras coisas, servir de eixo para uma rotação); COLOR (cor, com as três coordenadas R, G e B - red, green e blue - normalizadas entre 0 e 60000, segundo convenção do ProSIm; pode representar a cor de uma fonte de luz, por exemplo).

Um ator terminal pode ser: FACE (não pode ser uma face solta no espaço, mas uma face de um ator tipo POLY); POLY (poliedro criado por um Modelador Geométrico, por exemplo o Geomod - ver /MADE-92/; é o tipo de ator que é usado na maioria das vezes); ENUM (sequência numerada de poliedros; em cada quadro aparecerá um poliedro desta sequência) e NICKY (não incorpora nenhuma estrutura, é apenas o "apelido" de uma aglutinação de vários atores; por exemplo: um ator poliedro representando o braço e outro representando o antebraço, podem ser aglutinados, com o apelido de "membro_superior").

Deve ser notado que o uso de atores do tipo POINT e FACE como parte de um ator POLY "pode gerar falta de integridade geométrica" (não planaridade de faces, por exemplo) no último. "Isto ocorre porque o acesso e manipulação do animador a quaisquer elementos geométricos de um objeto é feita de maneira direta e irrestrita" /SILV-92/.

O uso do tipo NICKY já demonstrou a existência de uma hierarquia entre os atores. De fato, podem ser definidos atores *mestres* e atores *escravos*. Qualquer transformação sofrida pelo mestre (seja ele um NICKY ou um ator com estrutura geométrica) é propagada para seus escravos, e assim sucessivamente, até a posição mais baixa da hierarquia. Um escravo, entretanto, pode sofrer transformações independentemente do mestre.

Quanto à decoração, não existem todos esses tipos: *decoração só pode ser poliedro* (POLY).

Este módulo trata da definição e caracterização de atores e decorações; em outras palavras: o que é o ator (POLY, POINT, etc), qual sua cor, se ele participa da cena ou não... . Além disso estão definidas em actor.h as funções responsáveis pela criação da hierarquia entre os atores (quem é mestre e quem é escravo).

Também estão neste módulo as funções que determinam as Transformações Geométricas Lineares Rígidas (ver /BADL-87/). As tranformações disponíveis são: translação, rotação e escalamento.

3.1 - Estruturas do actor.h:

A_R - É uma enumeração que, usada como parâmetro de uma função, define se ela vai ter condição absoluta ou relativa:

```
typedef enum { ABSOLUTE, RELATIVE } A_R;
```

ValueType - Outra enumeração que define o tipo de valor tratado:

```
typedef enum {  INT,  DOUBLE,  POINT,  VECTOR,  COLOR,
                FACE,  POLY,  ENUM, NICKY  }  ValueType;
```

FacePlus - Estrutura que caracteriza uma face de um ator poliédrico:

```
typedef struct  {   int a;
                    int fn;
                    int nvert;  }  FacePlus;
```

O primeiro campo (inteiro "a") identifica o ator ao qual pertence a face definida (os atores são numerados). O segundo campo (inteiro "fn") é o número da face na estrutura poliédrica (ator "a"); as faces dos atores poliédricos também são numeradas. O último campo (inteiro "nvert") indica o número de vértices da face definida.

PolyPlus - Define um poliedro:

```
typedef struct  {  Polyhedron        *p;
                   SIPP_Surface      *sipp_s;
                   Surface           *v;
                   SIPP_ShadeType    s;
                   SIPP_TextureType sipp_texture;
                   Boolean           shadow;
                   int               a_list, r_list;
                   int               size_list;
                   Wireframe         wire_list;
                   Boolean wf_calculated;  }  PolyPlus;
```

O primeiro campo é de Polyhedron, uma estrutura definida no ProSIm. É o poliedro propriamente dito (tem o número de faces e vértices, além da lista de faces e vértices). Ver a descrição desta estrutura no apêndice A.

O segundo campo é do tipo SIPP_Surface, estrutura definida em light.h., responsável pela definição das características de cor de um ator ou decoração a serem usadas pelo renderizador (SIPP).

O campo a seguir (do tipo Surface) é equivalente ao anterior, mas usado para o antigo renderizador Scanline.

O campo do tipo SIPP_Shadetype, também definido no módulo light.h, determina o modelo de tonalização - "shading" (pode ser LINE, LINE_ENVELOPE, FLAT_SIPP, GOUR_SIPP ou PHONG_SIPP - ver mais detalhes em light.h).

O tipo SIPP_TextureType (campo “sipp_texture”) também é definido em light.h e indica o tipo de textura do objeto (o SIPP permite 6 tipos de texturas: BASIC, PHONGS, MARBLE, GRANITE, STRAUSS e WOOD).

O campo shadow indica se o ator terá ou não sombras. Na versão atual ele não é utilizado, pois os renderizadores utilizados não permite dar esta característica aos objetos.

Os campos de inteiros (a_list, r_list e size_list) serão valores para tamanho de vetores que listarão os vértices e outros dados a respeito da estrutura.

O campo wire_list é uma Wireframe, definida no Scanline (ver apêndice B).

O último campo é uma variável booleana que indica se o campo Wireframe já foi calculado ou não.

Element - união de todos os tipos possíveis para um ator (real, ponto, vetor, cor, face e poliedro). Vai ser muito utilizada daqui para frente.

```
typedef union  {  double          *d;
                  Point           *p;
```

```
                Vector          *v;
                Color           *c;
                FacePlus        *f;
                PolyPlus*o;  }  Element;
```

Element é utilizada em campos que podem receber qualquer um destes tipos (por exemplo, o campo "master" da estrutura "Actor", que será definida a seguir).

Actor - estrutura que descreve o ator:

```
typedef struct  {  String          *name;
                   Boolean         activity;
                   ValueType       t;
                   ActorType       a_t;
                   Element         *master;
                   Matrix          atm;
                   int             n_slaves;
                   int             slaves[N_MAX_SLAVES];
                   Point           gc;
                   Boolean         gc_changed, gc_calculated;
                }          Actor;
```

O primeiro campo ("name") é uma String que define o nome do ator.

O segundo campo é uma variável booleana ("activity"), que identifica se o ator está presente (ativo) em alguma cena, num instante avaliado.

O campo "t" é do tipo ValueType, anteriormente definida, que identifica o tipo de ator (POINT, FACE, etc).

O campo “a_t” é do tipo ActorType, definido pelo antigo Scanline, e indica se o ator é “aberto” ou “fechado”.

O campo "master" é do tipo Element e "conterá efetivamente a estrutura geométrica e topológica do ator". /SILV-92/

"atm" é do tipo Matrix, definido no ProSIm (apêndice A). É a chamada "Accumulated Transformation Matrix", matriz que acumula as transformações sobre o ator no intervalo de 1/Rate segundos (intervalo de tempo de cada quadro).

O campo "n_slaves" identifica o número de escravos ("slaves") dependentes do ator.

"slaves[ ]" é uma lista de inteiros que identifica cada um dos escravos do ator (cada ator está associado a um inteiro).

O ponto "gc" identifica a "referência básica para acesso ao ator e que determinará as suas transformações relativas". /SILV-92/. Normalmente, "gc" representa o centro de gravidade do polígono (ator FACE) ou do poliedro (ator POLY). Para ator POINT, "gc" é o próprio ponto. O valor "gc" pode ser mudado usando-se um ator tipo NICKY (que aglutinará atores com um "gc" especificado).

Os dois últimos campos são variáveis booleanas para identificar se o valor de "gc" já foi calculado ("gc_calculated") e se foi mudado ("gc_changed").

Decor - descreve a decoração. É semelhante à estrutura "Actor" e, como decorações não apresentam movimentos, eliminam-se os campos relacionados com transformações (todos os campos a partir de "atm"). Como decoração só pode ser do tipo POLY, também não há a necessidade do campo "t", que identificava o tipo de ator. Além disso, pelo mesmo motivo, o campo Element pode ser substituído por um campo PolyPlus. Assim, temos a estrutura Decor:

```
typedef struct  {  String          *name;
                   Boolean         activity;
```

```
    PolyPlus*decor;
} Decor;
```

3.2: Macros do actor.h:

N_MAX_ACTORS - definida como sendo N_MAX_OBJ, sendo esta última definida no Scanline como 1000. Isso significa uma capacidade de até 1000 atores numa animação.

N_MAX_DECORS - definida como sendo 1000 (pode-se ter até 1000 decorações numa animação).

N_MAX_SLAVES - definido como 200 (um ator pode ter até 200 escravos).

PONTO (N, O) - na função que define o ator ponto ("cast_point", do arquivo a_define neste módulo), pode-se usar esta macro, quando tratar-se de um ponto de um ator poliedro. Por exemplo:

```
PONTO ( 1, "cubo");
```

Neste exemplo, está sendo referido o ponto 1 do ator "cubo". (O problema maior é saber qual é o ponto 1 do cubo, pois este depende da maneira como o cubo foi criado pelo modelador geométrico. A melhor maneira de saber qual o número do ponto desejado num ator poliédrico é por "tentativa e erro".)

A função "cast_point" ficaria então (seu primeiro parâmetro é o nome do ator ponto e o segundo é o próprio ponto):

`cast_point ( "ponto1", PONTO( 2, "cubo" ) );` que é equivalente à forma mais complexa:

`cast_point ( "ponto1", & ( Ppoly ( id_actor("cubo")) ->p_list [ 2] ) );`

OBS.: Os pontos referidos são vértices do poliedro, portanto têm número limitado (por exemplo, num cubo, só existirão os pontos de 0 a 7).

Cada uma das macros a seguir tem como parâmetro um valor inteiro que identifica uma estrutura Actor. Dependendo da função usada, se chega ao campo desejado do ator. Como um exemplo:

`p = Ppoint ( 2 );` significa que a variável p vai receber o valor do campo p do elemento "master", que por sua vez é um campo do ator número 2.

Ppoint (id) - equivale a: actors[(id)]->master->p.

Pvector (id) - equivale a: actors[(id)]->master->v.

Pcolor (id) - equivale a: actors[(id)]->master->c.

Pface_ (id) - equivale a: actors[(id)]->master->f.

Ppoly (id) - é equivalente a: actors[(id)]->master->o->p.

Ppoly_ (id) - é equivalente a: actors[(id)]->master->o.

Pint (id) - é equivalente a: actors[(id)]->master->i.

Pdouble (id) - equivale a: actors[(id)]->master->d.

3.3: Funções do actor.h:

As funções deste módulo estão em 5 arquivos: a_define.c (funções relacionadas à definição de atores e decorações); a_miscel.c (miscelânea de funções, dentre elas as que hierarquizam os atores - criam e anulam a relação mestre/escravo entre atores); a_transl.c (funções de translação de atores); a_rotate.c (funções de rotação) e a_scale.c (funções de escalamento de atores).

3.3.1: Arquivo a_define.c:

No início deste arquivo, é declarado um vetor de "Actor", com N_MAX_ACTORS elementos e também um vetor de "Decor", com N_MAX_DECORS elementos:

```
Actor  *actors[ N_MAX_ACTORS ];
Decor  *decors [ N_MAX_DECORS ];
```

Também são declaradas duas variáveis inteiras: n_actors (número de atores) e n_decors (número de decorações).

As funções deste arquivo são descritas a seguir:

id_actor (String) - função que identifica um ator. Seu único parâmetro é o nome do ator (como toda String, entre aspas). Esta função faz uma verificação na lista de atores e retorna o inteiro correspondente ao ator desejado. Geralmente é usada com funções ou macros que têm como parâmetros valores inteiros correspondentes aos atores. Por exemplo:

```
Ppoly_ ( id_actor("cubo1") );
```

Esta macro, como já foi visto, acessa o campo "o" do elemento "master" do ator cubo1.

id_decor (String) - exatamente igual à anterior, só que deve ser usada para identificar decorações, e não atores.

As funções "cast", vistas a seguir, são as que definem os atores. São elas que colocam os valores determinados nos respectivos campos da estrutura Actor. Por definição, o primeiro parâmetro é sempre uma String que representa o nome do ator a ser definido (esta String é colocada no campo "name" da estrutura Actor).

cast_point (String, Point) - define ator do tipo ponto (no campo "t" da estrutura Actor é colocado POINT). Se o ator for um ponto de um poliedro (vértice de um ator POLY), o segundo parâmetro pode ser uma macro do tipo PONTO ( ). O ator POINT também pode ser um ponto qualquer do espaço, sem necessariamente fazer parte de um poliedro. Veja os exemplos:

```
cast_point ( "ponto1", PONTO(0,"cubo") );  /* Ponto 0 do ator cubo.*/
```

```
cast_point ( "ponto2", p );                    /* Ponto p qualquer, previamente
                                                  definido com a função SET_POINT. */
```

cast_vector (String, Vector) - define ator do tipo vetor (no campo "t" da estrutura Ator é colocado VECTOR). O segundo parâmetro é uma variável (do tipo Vector), com o valor que vai ser recebido pelo ator. Por exemplo:

```
(...)   BEGIN_SCRIPT
                .                       /* Tem que ser no script, e não
                .                        na cena. */
                .
        Vector v;
        SET_VECTOR ( v, 1.0, 1.0, 3.0 );
                .
                .
                .
        cast_vector ( "vetor1", v );
```

Neste exemplo, o ator "vetor1" vai receber, na sua definição, o valor da variável "v" (1, 1, 3).

cast_color (String, Color) - define ator do tipo COLOR. O segundo parâmetro é uma variável (tipo Color), com o valor que vai ser recebido pelo ator. Por exemplo:

```
(...)   BEGIN_SCRIPT
                .                       /* Tem que ser no script, e não
                .                        na cena. */
                .
        Color c;
        SET_RGB ( c, 10000., 10000., 50000. );
                .
                .
                .
        cast_color ( "cor1", c );
```

Neste exemplo, o ator "cor1" vai receber, na sua definição, o valor da variável "c" (10000, 10000, 50000).

cast_double (String, double) - define ator do tipo DOUBLE. O segundo parâmetro é simplesmente um valor real:

```
cast_double ( "doub", 50. );
```

Neste exemplo, o ator doub receberá, na definição, o valor 50.

cast_face (String, String, int) - define ator do tipo FACE. O segundo parâmetro é uma String que determina o ator ao qual pertence a face (uma face, ao contrário de um ponto, não pode ser livre no espaço; tem que ser uma face de um ator tipo POLY). O último parâmetro é o valor inteiro que identifica qual face do ator POLY está sendo referida (este valor, obviamente, é limitado pelo número de faces do poliedro). Exemplo:

```
cast_face ( "face1", "cubo", 1 );
```

Neste exemplo, o ator "face1" é definido como sendo a face número 1 do ator "cubo".

cast_poly (String, String) - define ator do tipo POLY, que é o tipo mais usado em quase toda animação. O segundo parâmetro é o "path" do poliedro desejado. Exemplo:

```
cast_poly ( "piram", “/usr/prosim_objects/pyr.brp” );
```

Esta função define o ator "piram" como o poliedro "pyr.brp", da biblioteca /usr/prosim_objects.

cast_enum (String, String, int) - define uma sequência numerada de atores, onde em cada quadro aparecerá um ator desta sequência. O primeiro parâmetro é o nome (único) do ator na animação. O segundo parâmetro é o nome dos arquivos .brp numerados (por exemplo, se o segundo parâmetro for "bola", o ator será definido por bola000.brp, bola001.brp, bola002.brp, etc., nos respectivos quadros). O último parâmetro é o número de elementos da sequência numerada. Exemplo:

cast_enum ("piram", "/brp/piram", 6);

Esta função define o ator "piram" como sendo /brp/piram000.brp no quadro 0, /brp/piram001.brp no quadro 1, ..., até /brp/piram005 no quadro 5 da animação.

cast_nicky (String, double, double, double) - define ator do tipo NICKY. Os três últimos parâmetros determinam, respectivamente, as coordenadas x, y e z do ponto de referência para o ator (é o equivalente ao centro de gravidade de um poliedro). É em relação a este ponto que vão ser realizadas as transformações relativas do ator NICKY definido.

furniture (String, String) - É o equivalente da função "cast_poly", só que para decoração. O primeiro parâmetro é o nome da decoração e o segundo é o "path" do poliedro que vai ser a decoração.

As funções "paint", vistas a seguir, são as que determinam a cor e o modelo de tonalização (FLAT, PHONG ou GOUR - ver módulo light.h) do ator ou decoração. É importante saber que *somente atores do tipo POLY podem ser pintados.*

sipp_paint_actor (String, SIPP_TextureType, String) - O primeiro parâmetro é o nome do ator a ser pintado. O segundo parâmetro determina o modelo da textura (BASIC, GRANITE, etc.). O último parâmetro é o "path" da textura escolhida:

paint_actor ("esfera", STRAUSS, “/usr/prosim_colors/laranja.cor”);

Neste exemplo, o ator "esfera" terá a cor definida no arquivo "laranja.cor" da biblioteca /usr/prosim_colors. Esta textura é do tipo STRAUSS (metálica).

Apenas atores do tipo POLY podem ser pintados por esta função.

sipp_paint_enum (String, SIPP_TextureType, String) - Idêntica à anterior, mas pinta atores do tipo ENUM.

sipp_paint_morph (String, SIPP_TextureType, String, int, double) - Cria um “morphing” de cores para o ator (cujo nome é o primeiro parâmetro). A cada quadro uma cor diferente é aplicada ao ator. O segundo parâmetro indica o tipo de textura aplicada, e o terceiro é o “path” e o nome dos arquivos de cor (sem a extensão .cor). O penúltimo parâmetro é o número de elementos da sequência numerada de cores e o último parâmetro é o instante em que começa o “morphing”:

sipp_paint_morph(“A1”, STRAUSS, “/usr/prosim_colors/laranja”, 10, 20.);

No exemplo acima, a partir do instante 20. (último parâmetro), o ator A1 receberá as cores laranja000.cor, laranja002.cor, ..., laranja009.cor, uma por quadro. Antes disso ele tinha a cor laranja000 e ao final, terá a cor laranja010.

paint_actor(String, ShadeType, String) - Equivalente a “sipp_paint_actor”, usada no antigo renderizador Scanline.

sipp_paint_decor(String, SIPP_TextureType, String) - Equivalente a “sipp_paint_actor”, usada para pintar decorações.

paint_decor (String, ShadeType, String) - Semelhante à anterior, usada pelo antigo Scanline.

sipp_render(SIPP_ShadeType, int, Boolean) - Função que deve ser usada antes das “sipp_paint_...”. Ela define o tipo de tonalização a ser usada na cena (o SIPP, diferentemente do Scanline, não aceita um tipo de tonalização diferente para cada ator). O segundo parâmetro define a taxa de superamostragem (quanto maior, mais demorada e com menos alias será feita a renderização); não é aconselhável usar um valor maior que 3 aqui. O último parâmetro define se serão ou não geradas sombras.

part (String) - função que define a participação do ator em determinada cena. É responsável pela colocação de TRUE no campo "activity" da estrutura Actor (e de todos os seus escravos), indicando participação na cena. Exemplo:

```
BEGIN_SCENE          /* Início da cena. */
NO_CLOCK;            /* Definição do relógio local. */
part ( "cubo1" );
part( "ponto1" );
part ( "esfera" );   /* Os atores cubo1, ponto1 e esfera
                        participam da cena. */
```

part_enum (String, double) - define a participação do ator (tipo ENUM, definido por "cast_enum") em determinada cena. O primeiro parâmetro é o nome do ator e o segundo, o quadro a partir do qual a sequência de .brp numerados começará Exemplo:

```
BEGIN_SCENE          /* Início da cena. */
NO_CLOCK;            /* Definição do relógio local. */
part _enum( "cubos", 0. ); /* cubo000.brp no quadro 0, cubo001.brp no quadro 1, ... */
part ( "esfera", FRAME(4) ); /* esfera000.brp no quadro 4, esfera001.brp no 5, ... */
```

unpart (String) - encerra a participação do ator em determinada cena. Coloca FALSE no campo "activity" da estrutura Actor (e de todos os seus escravos).

decor (String) - exatamente igual à função "part". Entretanto, esta é usada para determinar a participação de uma decoração na cena.

undecor (String) - equivalente a "unpart", só que usada para objetos de decoração.

OBS.:

1. As funções *cast, furniture* e *paint* são usadas só no *script.*

2. As funções *part, unpart* e *decor* são usadas no início da cena (após o BEGIN_SCENE e a definição do relógio interno da cena - ver módulos time.h e script.h).

3.3.2: Arquivo a_miscel.c:

Funções relacionadas à hierarquia entre atores:

group (String, String) - o primeiro parâmetro é o nome do ator mestre e o segundo, o nome do ator que será seu escravo. Esta função coloca o número do ator escravo no campo "slaves" do ator mestre.

grasp (String, String) - semelhante à função "group", sendo que antes é verificado se já não existe uma relação mestre/escravo entre os dois atores especificados como parâmetros desta função. Outra diferença importante é que, nesta função, o primeiro parâmetro é o nome do escravo e o segundo, o nome do ator mestre (logo, invertido). Além disso, existe mais uma diferença entre "group" e "grasp": é que a segunda pode ser "aplicada durante a animação e pode ser desfeita, enquanto que a primeira é usada para definir um personagem como uma aglutinação de partes (atores) de maneira permanente". /SILV-92/.

Como exemplo:

```
grasp ( "ponto1", "cubo1" );
```

Neste exemplo, o ator "ponto1" é definido como escravo de "cubo1".

OBS.: Não se pode colocar diretamente um ponto como escravo de um ator. O ponto deve ser declarado como ator e então poderá ser escravo de outro.

ungrasp (String, String) - função booleana usada para desfazer uma relação mestre/escravo entre o primeiro (mestre) e segundo (escravo) parâmetros. Se a relação for desfeita, ela retorna TRUE. Caso contrário (se o segundo não for escravo do primeiro, por exemplo), ela retorna FALSE. Veja o exemplo:

```
(...)   Boolean x;
            .
            .
            .
        x = ungrasp ( "cubo1", "ponto1" );
                                /* ponto1 já não é mais escravo de cubo1.*/
        if ( x ) {
            .
            .                   /* Funções realizadas se a função ungrasp
            .                    retornar TRUE. */
            }
```

grasped (String, String) - função booleana. Testa se o primeiro parâmetro é mestre do segundo. Se assim for, retorna TRUE, caso contrário, retorna FALSE. Usada para evitar declarações "grasp" ambíguas (um escravo sendo declarado como mestre de seu mestre !).

broad_actor (Matrix, int) - função que propaga as transformações sofridas por um ator para todos os seus escravos. O primeiro parâmetro é a matriz de acumulação, que determinou a transformação do mestre. O segundo parâmetro é o número que identifica o escravo que sofrerá a mesma transformação. Esta função não é usada pelo animador, mas é chamada internamente ao final de cada transformação, para propagá-la aos escravos.

OBS.: As funções *grasp*, *group* e *ungrasp* devem ser usadas dentro da cena.

Outras funções:

gc_actor (String) - retorna um ponto, que é o centro de gravidade do ator (cujo nome é o parâmetro da função). Este ponto (baricentro), vai ser o ponto de referência para transformações relativas. Ao final da sua execução, ela coloca TRUE no campo gc_calculated do ator.

transf_actor (int) - tem como único parâmetro o inteiro que identifica o ator que sofrerá uma transformação. A transformação é efetuada a partir de uma matriz homogênea como parâmetro. Não é usada diretamente pelo animador.

shade_actor (String, ShadeType) - determina o modelo de tonalização do ator tipo POLY (cujo nome é o primeiro parâmetro). O modelo usado é o segundo parâmetro. Pode ser usada tanto no script (apesar do modelo já ser determinado na função "paint_actor") quanto na cena (neste caso, quando quiser mudar o modelo a partir de certo instante). Por exemplo, dentro de uma cena:

```
After ( 4.0 )
        {       shade_actor ( "esfera", PHONG );        }
```

Nesta cena, após o instante 4 segundos, o ator "esfera" vai ser tonalizado com o modelo PHONG, mesmo que na função "paint_actor" tenha sido usado outro modelo.

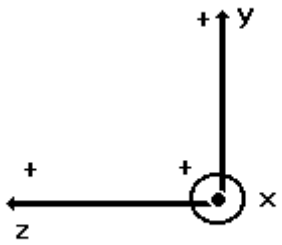
type_actor (String, ActorType) - define se ator será aberto ou fechado (definição usada pelo antigo renderizador).

3.3.3: Arquivo a_transl.c:

Neste arquivo, estão funções relacionadas à translação (transformação geométrica linear rígida). Por convenção, sempre o último parâmetro é uma String que representa o nome do ator que sofrerá a translação.

O primeiro parâmetro é sempre 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 exemplos dados em cada função tornarão isto mais claro.

Na imagem gerada, os três eixos de coordenadas têm os sentidos indicados pela figura abaixo (convenção da mão-direita - Right Hand) -- o eixo x tem sentido positivo saindo da página.



translate_actor (A_R, double, double, double, String) - os parâmetros "double" desta função são as coordenadas do ponto para onde o ator se deslocará (se a translação for absoluta) ou a direção para a qual o ator se deslocará, a partir de seu centro de gravidade (se a translação for relativa). Veja o exemplo:

```
(...)   At ( 2.5 )
        {       translate_actor ( ABSOLUTE,  0.0,  3.0,  2.0, "cubo1" );
                translate_actor ( RELATIVE,  0., 3.,  2., "cubo2" );   }
```

Neste exemplo, no instante 2.5 segundos, o cubo1 será deslocado para o ponto (0, 3, 2). O cubo2 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, double, String) - é um caso particular da função anterior, em que o deslocamento se faz de n unidades (n é o segundo parâmetro) na direção x. É equivalente a se fazer o ponto da função anterior como sendo (n, 0., 0.). Assim: translate_actor_x (ABSOLUTE, 3., "esfera") = translate_actor (ABSOLUTE, 3.0, 0., 0., "esfera"). Veja o exemplo:

```
(...)   At ( 2.5 )
        {       translate_actor_x ( ABSOLUTE, 1.5, "cubo1" );
                translate_actor_x ( RELATIVE, 1.0, "cubo2" );       }
```

No exemplo, o ator "cubo1" será deslocado para o ponto (0, 0, 1.5), no instante 2.5 segundos. O cubo2, que sofreu translação relativa, vai se deslocar 1 unidade na direção x, a partir da posição onde estava.

translate_actor_y (A_R, double, String) - é o equivalente da função anterior, fazendo-se o deslocamento na direção y. Assim: translate_actor_y (RELATIVE, 1.4, "esfera") = translate_actor (RELATIVE, 0., 1.4, 0., "esfera") . Como exemplo:

```
(...)   At ( 2.0 )
        {       translate_actor_y ( ABSOLUTE, 1.8, "cubo1" );
                translate_actor_y ( RELATIVE, -2.0, "cubo2" );      }
```

No exemplo, o ator "cubo1" será deslocado para o ponto (0, 1.8, 0), no instante 2.0 segundos. O cubo2, que sofreu translação relativa, vai se deslocar 2 unidades na direção -y, a partir da posição onde estava.

translate_actor_z (A_R, double, String) - exatamente igual às duas anteriores, fazendo o deslocamento na direção z.

translate_actor_actor(A_R, String, String, double, String) - os parâmetros do tipo String são atores previamente definidos. Se o primeiro parâmetro for ABSOLUTE, o ator dado pelo último parâmetro se moverá para junto do ator dado pelo 3º. parâmetro. Se for RELATIVE, o ator se moverá n unidades (n é o parâmetro do tipo double) na direção dada por cg_first - cg_second (onde "cg_" significa centro de gravidade do ator e "first" indica o ator definido pelo segundo parâmetro e "second", o ator definido pelo terceiro parâmetro). Exemplo:

```
translate_actor_actor(RELATIVE, "A1", "A2", 10. , "A3");
```

No exemplo, o ator A3 se moverá 10 unidades na direção dada pelo centro de gravidade de A1 menos o centro de gravidade de A2.

OBS.:

1. Ao final destas funções é usada internamente a função "broad_actor", para propagar a translação para todos os escravos.

2. As funções de translação devem ser usadas dentro da cena.

3.3.4: Arquivo a_rotate.c:

Neste arquivo, estão funções relacionadas à rotação (outra transformação geométrica linear rígida). Por convenção, sempre o último parâmetro é uma String que representa o nome do ator que sofrerá a rotação.

O primeiro parâmetro é sempre a condição ABSOLUTE ou RELATIVE, se esta for necessária. *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 no sentido horário e a negativa, no anti_horário; desde que o observador esteja olhando do lado positivo de cada eixo. (Por exemplo: com o observador em um ponto positivo do eixo x, observando a origem, olha-se o eixo x do lado negativo, fazendo com que a rotação positiva seja no sentido anti_horário.) A convenção que determina o sinal positivo para a rotação é semelhante à regra da direção das linhas de força num fio de corrente (eletromagnetismo - ver /HAYT-83/).

rotate_actor (A_R, double, double, double, double, String) - os três primeiros "double" são as coordenadas de um vetor, que será o eixo da rotação. O penúltimo parâmetro é a quantidade (em graus) que o ator girará. Por exemplo:

```
(...)   After ( 3.0 )
        {       rotate_actor ( ABSOLUTE, 1., 0., 2., 60., "cubo1" );
                rotate_actor ( RELATIVE, 1., 0., 2., -20., "cubo2" );  }
```

No exemplo, a partir do instante 3 segundos, o ator "cubo1" vai girar, a cada quadro, 60º em torno da origem (rotação absoluta), tendo como eixo o vetor (1, 0, 2). O ator "cubo2" vai girar, a cada quadro, -20º em torno de seu centro de gravidade, tendo o mesmo vetor como eixo.

rotate_actor_x (A_R, double, String) - caso particular da função anterior, em que o vetor (eixo da rotação) é o eixo x. O segundo parâmetro é o valor em graus da rotação. Assim: rotate_actor_x (RELATIVE, 30., "esfera") = rotate_actor (RELATIVE, 1., 0., 0., 30., "esfera") . Exemplo:

```
(...)   After ( 3.5 )
        {       rotate_actor_x ( ABSOLUTE,  -50., "cubo1" );
                rotate_actor_x ( RELATIVE, 25., "cubo2" );      }
```

No exemplo, a partir do instante 3.5 segundos, o ator "cubo1" vai girar, a cada quadro, -50º em torno da origem (rotação absoluta), tendo como eixo de rotação o eixo x cartesiano. O ator "cubo2" vai girar, a cada quadro, 25º em torno de seu centro de gravidade, com o mesmo eixo de rotação (eixo x).

rotate_actor_y (A_R, double, String) - semelhante à anterior, considerando o eixo y como o eixo de rotação.

rotate_actor_z (A_R, double, String) - semelhante às duas anteriores, considerando o eixo z como o eixo de rotação.

free_rotate_actor (double, double, double, double, double, double, double, String) - 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 penúltimo parâmetro é o valor em graus que o ator girará. O último parâmetro continua sendo o nome do ator. Exemplo:

```
(...)   At ( FRAME (10) )
        {       free_rotate_actor ( 2., 5., 6., 0., 1., 2., 45., "esfera" );  }
```

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.: 1. Se o ponto de referência for a origem, a função "free_rotate_actor" será igual à função "rotate_actor (ABSOLUTE, ...)". Se o ponto de referência for "gc_actor" (centro de gravidade), a função "free_rotate_actor" será igual a "rotate_actor (RELATIVE, ...)".

rotate_actor_actor(A_R, String, String, double, String) - Os parâmetros do tipo String são atores previamente definidos, sendo que o último indica o ator que gira. O eixo de rotação é definido por cg_first - cg_second (onde “cg_” significa centro de gravidade do ator, “first” é o ator definido pelo segundo parâmetro e “second” é o ator definido pelo terceiro parâmetro). Se o primeiro parâmetro for ABSOLUTE, o ator girará em torno do ator definido pelo terceiro parâmetro. Se for RELATIVE, o ator girará em torno do seu centro de gravidade. O parâmetro do tipo double é o número de graus girados. Exemplo:

```
rotate_actor_actor(RELATIVE, “first”, “second”, 33., “movel”);
```

No exemplo acima o ator “movel” girará 33 graus em torno do eixo dado pela diferença dos centros de gravidade de “first” e de “second”.

OBS.:

1. Ao final destas funções é usada internamente a função "broad_actor", para propagar a rotação para todos os escravos.

2. As funções de rotação devem ser usadas dentro da cena.

3.3.5: Arquivo a_scale.c:

Neste arquivo, estão funções relacionadas ao escalamento (outra transformação geométrica linear rígida). Escalamento significa o aumento ou diminuição do tamanho do objeto em uma ou mais direções. Por convenção, sempre o último parâmetro é uma String que representa o nome do ator que sofrerá o escalamento.

O primeiro parâmetro é sempre a condição ABSOLUTE ou RELATIVE, se esta for necessária. *"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/.

scale_actor (A_R, double, double, double, String) - os parâmetros "double" desta função são coordenadas que representam o quanto o ator vai crescer ou diminuir em cada direção (x, y e z, respectivamente). Estas coordenadas devem ser dadas em valores percentuais. Valores negativos nas coordenadas significa diminuição de tamanho. Exemplo:

```
(...)   scale_actor ( ABSOLUTE, 30., 20., 0., "cubo1" );
        scale_actor ( RELATIVE, -50., 0., -60., "cubo2" );
```

No exemplo, o ator "cubo1" vai aumentar, com relação à origem (de maneira absoluta): 30% na direção x, 20% na direção y e nada na direção z (coordenadas de p1). O ator "cubo2" diminui, com relação ao seu centro de gravidade (de maneira relativa): 50% na direção x e 60% na direção z (coordenadas de p2).

scale_actor_x (A_R, double, String) - 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., "esfera") = scale_actor (RELATIVE, 35., 0., 0., "esfera"). Veja o exemplo:

```
{       scale_actor_x ( ABSOLUTE, -50., "cubo1" );
        scale_actor_x ( RELATIVE, 10., "cubo2" ); }
```

No exemplo, o ator "cubo1" vai diminuir, com relação à origem (de maneira absoluta), 50% na direção x. O ator "cubo2" aumenta, com relação ao seu centro de gravidade (de maneira relativa), 10% na direção x.

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

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

growth (A_R, double, String) - função que permite um crescimento igual nas três dimensões. O segundo parâmetro é o valor, em porcentagem, deste crescimento. Assim:

growth(ABSOLUTE, 65., "esfera") = scale_actor (ABSOLUTE, 65., 65., 65., "esfera").

shrink (A_R, double, String) - 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., "cubo") = growth (RELATIVE, -40., "cubo");

free_scale_actor (double, double, double, double, double, double, String) - esta função 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). O três primeiros parâmetros desta função são as coordenadas do ponto de referência desejado. Depois, vêm as coordenadas que determinam o valor percentual do escalamento em cada direção. O último parâmetro continua sendo o nome do ator. Exemplo:

```
(...)    At ( FRAME (14) )
         {      free_scale_actor ( 0., 2., 6., 0., 10., -36., "esfera" );    }
```

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, a função "free_scale_actor" será igual à função "scale_actor (ABSOLUTE, ...)". Se o ponto de referência for "gc_actor" (centro de gravidade), a função será igual a "scale_actor (RELATIVE, ...)".

scale_actor_actor(A_R, String, String, double, String) - Os parâmetro do tipo String são atores previamente definidos, sendo que o último é o ator que sofrerá o escalamento. O escalamento se dará na direção definida por cg_first - cg_second (onde “cg_” significa centro de gravidade, e “first” e “second” representam, respectivamente, os atores dados pelo segundo e terceiro parâmetros da função). 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 do tipo double é a percentagem do escalamento.

OBS.:

1. Ao final destas funções é usada internamente a função "broad_actor", para propagar o escalamento para todos os escravos.

2. As funções de escalamento devem ser usadas dentro da cena.

A seguir, será dada uma animação completa, para que se possa melhor visualizar o uso das funções e macros até aqui vistas:

```
/********************** Início da Animação ************************/
#include <stdio.h>
#include "p_transf.h"
#include "p_vector.h"
```

```
#include "p_alloc.h"
#include "p_error.h"
#include "p_graph.h"                /* No início de cada animação, deve-se incluir todos */
#include "p_poly3d.h"               /* os arquivos.h necessários. */
#include "camera.h"
#include "actor.h"
#include "time.h"
#include "scanline.h"
#include "sl_wire.h"
#include "light.h"
#include "script.h"
#include "motion.h"

/* extern Actor  *actors[ N_MAX_ACTORS ]; */
                                    /* Quando se usa macros do tipo Ppoly (id) ou qualquer
                                     outra macro ou função que use o vetor de atores, ele de-
                                     ve ser declarado no início. Mas esse não é o caso desta
                                     animação. */
void cena (cue )
Cue *cue;                           /* O procedimento cena se realizará no intervalo deter-
                                     minado por uma cue, que é o seu parâmetro. */
{                                   /* Início da cena. */
BEGIN_SCENE
NO_CLOCK;                           /* Definição do relógio local. */

part ( "cubo1" );                   /* Ator "cubo1" participa da cena. */
part ( "cubo2" );                   /* Ator "cubo2" participa da cena. */
part ( "cubos" );                   /* Ator "cubos" participa da cena. */
decor ( "esfera" );                 /* Decoração "esfera" participa da cena. */

At ( 0.0 )
{
        translate_actor_y ( ABSOLUTE, 2., "cubo1" );
        translate_actor_y ( ABSOLUTE, -2, "cubo2");
                                    /* cubo1 é inicializado no ponto ( 0, 2, 0 ) e cubo2
                                    no ponto ( 0, -2, 0 ). Toda decoração é inicializada
                                    na origem. */
        grasp ( "cubo1", "cubos" );
        grasp ( "cubo2", "cubos" );
                                    /* cubo1 e cubo2 são escravos de cubos. */
}

Between ( 0.0, 2.0 )
{
        rotate_actor_x ( ABSOLUTE, 30., "cubos" );
                                    /* rotação de "cubos" e seus escravos em torno da
                                     origem, com eixo de rotação x e girando 30º por
                                     quadro. */
}

At ( 2.0 )
{
        scale_actor_y ( RELATIVE, 20., "cubo1" );
        scale_actor_y ( RELATIVE, 20., "cubo2" );
                                    /* os dois cubos são "espichados" na direção y,
                                     transformando-se em paralelepípedos. */
```

```
}

After ( 2.0 )
{
        rotate_actor ( RELATIVE, 1., 0., 1., 30., "cubos" );
                                    /* rotação de "cubos" e seus escravos em torno de
                                     seu centro de gravidade, com eixo de rotação sendo
                                     o vetor (1,0,1). Gira-se 30 graus por quadro. */
}

END_SCENE
}                                   /* Fim da cena. */

BEGIN_SCRIPT ( 1 )                  /* Início do script com 1 cena. */

set_studio ( NULL, 2, "../images/scan/actor" );
                                    /* Usa-se configuração default ( NULL ), o "shoot"
                                     é do tipo 2 ( ver s_define.c, no script.h ) e as ima-
                                     gens geradas se chamarão "actor" e terão o "path"
                                     especificado. */

TotalTime = 4.0;
Rate = 20;                          /* Animação com 4 seg. de duração a uma taxa de
                                     20 quadros/s. */
cast_poly ( "cubo1", “cube.brp” );
cast_poly ( "cubo2", “cube.brp” );  /* cubo1 e cubo2 são poliedros do tipo cube.brp*/
cast_nicky ( "cubos", 0., 0., 0. ); /* cubos é um NICKY, com referência na origem. */
furniture ( "esfera", “esf.brp” );  /*esfera é poliedro do tipo esf.brp. */

sipp_render ( FLAT_SIPP, 1, FALSE );
sipp_paint_actor ( "cubo1", BASIC, “/usr/cores/amarelo.cor” );
sipp_paint_actor ( "cubo2", BASIC, “/usr/cores/amarelo.cor”);
sipp_paint_decor ("esfera", BASIC, “/usr/cores/amarelo.cor”);
                                    /* definição de cores e modelos de iluminação para
                                     os atores tipo POLY e para a decoração. */
SET_CUE ( cue[ 0], 0.0, 4.0 );      /* cue[ 0] começa no início e termina no final da
                                     animação. */
LIST_SCENES
        cena ( cue[ 0] );           /* A cena ocorrerá no intervalo determinado por
                                     cue[ 0]. */
END_LIST

END_SCRIPT
/**********************  Fim da animação ***************************/
```

Nesta animação, existe uma esfera imóvel na origem (decoração). Na inicialização, os atores "cubo1" e "cubo2" são colocados acima e abaixo da esfera, respectivamente.

No instante seguinte ao zero, e até o instante 2.0, o ator "cubos" gira em torno da esfera (que está na origem - rotação absoluta). Aliás, como "cubos" tem ponto de referência na origem, o efeito de uma rotação absoluta é o mesmo da relativa. Na prática, são os atores "cubo1"e "cubo2" que giram 30 graus a cada quadro, já que "cubos" é um apelido para a aglutinação destes. A rotação usa como eixo a direção x.

No instante 2.0, os dois cubos são "espichados" na direção y e têm seus tamanhos aumentados em 20% nesta direção (escalamento).

Depois do instante 2.0, a rotação de "cubos" recomeça. Porém, o novo eixo de rotação é o vetor (1, 0, 1).

4 - MÓDULO MOTION.H:

Este módulo é responsável pela definição de trajetórias ("caminho no espaço bidimensional ou tridimensional a ser percorrido por um determinado elemento da cena" /SILV-92/) e geração de trilhas (tracks). A trajetória é definida através da interpolação de curvas, a partir de pontos de controle. Seguidamente, são geradas as trilhas, sobre as quais o ator poderá se movimentar. O objetivo das trilhas é tornar os movimentos mais suaves e naturais.

Também no módulo motion.h está definido todo o movimento do sistema. No TOOKIMA, movimento se refere basicamente às acelereções sofridas por um ator ao seguir uma trajetória (ele pode ter um movimento acelerado, linear ou desacelerado).

4.1: Estruturas do motion.h:

Track - é definida como se segue:

```
typedef struct {   Boolean      done;
                   int          n_ele;
                   double       last_d_value;
                   ValueType    type;
                   Element      *eles;          } Track;
```

O campo booleano (variável "done") identifica se a trilha já foi calculada ou não. Dessa maneira, qualquer modificação na trilha ou no relógio local (por exemplo, uma cue de duração variável - ver módulo time.h) deve fazer Track.done = FALSE, de modo que a trajetória seja recalculada.

O campo inteiro (variável "n_ele") especifica o número de elementos de uma trilha (esses elementos são do tipo definido pela variável "type").

O campo real (variável "last_d_type") é usado para o caminhamento sobre a trilha e está relacionado com a "dinâmica" do movimento.

O campo ValueType (variável "type") - ver definição do tipo ValueType em actor.h - define o tipo de elemento que comporá a trilha. Poderão existir trilhas de POINT (para luzes, observadores, etc), DOUBLE (para ângulos, por exemplo), VECTOR (para eixos, por exemplo) e COLOR.

A variável "*eles" consistirá de um vetor de "n_ele" elementos do tipo Element - ver definição do tipo Element em actor.h. De cada elemento do vetor, apenas o campo correspondente ao tipo type será definido, construindo uma trilha de "n_ele" elementos do tipo "type". Por exemplo: se type = COLOR, serão definidos apenas track->eles[i].c .

Trajtype - não é uma estrutura, mas uma enumeração, definida como se segue:

```
typedef enum { LINEAR, QUADR, CUBIC, EXP, N_TYPES } TrajType;
```

4.2: Macros do motion.h:

Round (x) - definida como ((int)((x) + 0.5)). Usada para arredondar um valor x.

TRACK (name) - usada como uma forma alternativa de definição da track. É equivalente a: static Track *(name).

4.3: Funções do motion.h:

As funções definidas neste módulo estão em 3 arquivos: m_track.c (funções relacionadas à geração das trilhas e caminhamento sobre as mesmas), m_dynam.c (funções relacionadas à "dinâmica" do sistema) e m_math.c (funções matemáticas auxiliares para geração das trilhas).

4.3.1: Arquivo m_track.c:

known_track (TrajType, ValueType, Element, Element, char[3], Track *, Watch) - esta função cria uma "trilha_conhecida", ou seja, a trajetória é uma função matemática conhecida e determinada pelo seu primeiro parâmetro (TrajType). Este parâmetro pode ser: LINEAR (x), QUADR (x^2), CUBIC (x^3) ou EXP ($x*e^{(x-1)}$), definindo o tipo de trajetória desejada. O segundo parâmetro (ValueType) define o tipo de valor que está sendo tratado pela trajetória (pode ser DOUBLE, POINT, VECTOR ou COLOR). Os dois parâmetros seguintes definem, respectivamente, o valor inicial (start) e o final (stop) do tipo tratado (devem ser variáveis cujos valores foram previamente determinados através de funções do ProSIm: SET_POINT, SET_VECTOR, SET_RGB, etc, dependendo do tipo de valor tratado - segundo parâmetro da função). Estas funções do ProSIm estão descritas no apêndice A. O parâmetro seguinte é uma "string" com o nome da trilha (deve vir entre aspas); esta "string" deve ser T0, T1, etc, conforme seja a primeira, segunda ou n-ésima trilha declarada. O penúltimo parâmetro é a variável trilha e *deve ser precedida de &*. Esta variável deve ser declarada como Track no início do programa (através da macro TRACK ( ), já vista neste módulo). O último parâmetro é simplesmente o relógio local da cena (deve ser colocada aí a variável "clock").

O funcionamento de known_track é o seguinte: escolhido o tipo de trajetória a ser seguida, existe uma função associada a ela, como visto no parágrafo anterior. Todas as funções variam de 0 a 1 (embora de maneira diferente), para x variando de 0 a 1. Escolhido o tipo de valor tratado (escolhemos POINT, por exemplo), existirá um loop ("for") que a cada quadro definirá um ponto de uma lista de pontos de controle da seguinte maneira:

```
SET_POINT ( p_list[ i],   aux*(stop.p->x - start.p->x) + start.p->x,
                          aux*(stop.p->y - start.p->y) + start.p->y,
                          aux*(stop.p->z - start.p->z) + start.p->z );
```

onde p_list[i] é o i-ésimo elemento da lista; aux é a variável dada pela função escolhida (x, x^2, etc), que assumirá valores de 0 a 1; start e stop são, respectivamente, os pontos inicial e final da trajetória escolhida (anteriormente devem ter sido dados valores para estes pontos, por exemplo: SET_POINT (start, 0.0, 1.0, 1.6)). Percebe-se que, como aux varia de 0 a 1, os valores dos pontos p_list[i] variarão de start a stop.

A partir da lista de pontos (ou de cores, ou de vetores, ou de reais), cujo primeiro valor é start e o último é stop, chama-se a função apequs, que, a partir dos pontos de controle definidos anteriormente, retorna uma lista de pontos calculados por uma "Spline

cúbica"[2] periódica e quase uniformemente espaçada (ver função apequs, neste arquivo), aumentando significativamente o número de pontos que define a trilha.

Finalmente, os valores retornados pela função apequs serão colocados num arquivo que tem o nome dado pelo quinto parâmetro da função (no diretório /tmp).

Dois exemplos do funcionamento de known_track poderão deixar as coisas mais claras:

Ex.1:

```
(...)
TRACK ( T0 );                   /* declaração da variável T0 como do tipo
        .                       track . */
        .
        .
void cena1 (cue[ 0] )
{
        .                       /* outras variáveis da cena */
        .
        .
Element start, stop;            /* start e stop devem ser declarados como
        .                        elementos */
        .
        .


BEGIN_SCENE                     /* declaração TRACK ( ) deve ser externa à
        .                       cena */
        .
        .
start.p = ALLOC (1, Point);     /* É obrigatório o uso de ALLOC */
stop.p = ALLOC (1, Point);      /* ALLOC é uma função do ProSIm que
                                aloca memória em tempo de execução. O
                                 primeiro parâmetro indica a quantidade e o
                                 segundo, o tamanho da memória alocada.
                                 Neste caso, foi alocada memória para um
                                 ponto para start.p e para stop.p */
SET_POINT ( *start.p, 0.0, 3.0, 4.0);
SET_POINT ( *stop.p, 5.0, 4.0, 5.0);
                                /* É necessário o uso de *start.p e *stop.p */
known_track ( LINEAR, POINT, start, stop, "T0", &T0, clock);
(...)
```

/* Execução de known_track, para o caso linear de pontos:

Para i variando de 0 a n_ele = (Rate * (cue.stop - cue.start)) e x variando de 0 a 1, com passo 1/ n_ele faz-se:

```
{  aux = x;
   SET_POINT ( p_list[ i],   aux*(stop.p->x - start.p->x) + start.p->x,
                             aux*(stop.p->y - start.p->y) + start.p->y,
                             aux*(stop.p->z - start.p->z) + start.p->z );
}
```

[2] "Tecnicamente uma Spline é uma régua elástica, usada em desenhos de engenharia, que pode ser curvada de forma a passar por um dado conjunto de pontos (...). Sob certas hipóteses (...) pode ser descrita, aproximadamente, como sendo construída por partes, cada qual um polinômio cúbico (...). Tal função é chamada uma *função Spline cúbica*." /RUGG-88/

Chama-se a função apequs, que retorna uma lista de pontos (armazenada num arquivo) a partir dos pontos de controle em p_list :

```
apq = apequs ( p_list, n_ele, &n_ele );
```

Define-se também para a trilha:

```
T0->n_ele = n_ele;
T0->done = TRUE;
T0->type = POINT;
*/
```

Ex.2:

```
(...)
TRACK ( T0 );                   /* declaração da variável T0 como do tipo
    .                           track . */
    .
    .
void cena1 (cue[ 0])
    .
    .
    .
Element start, stop;            /* start e stop devem ser declarados como
    .                           elementos */
    .
    .
BEGIN_SCENE                     /* a definição TRACK ( ) deve ser externa à
    .                           cena */
    .
    .
start.d = ALLOC (1, double);    /* É obrigatório o uso do ALLOC */
stop.d = ALLOC (1,double);
*start.d = 0.;
*stop.d = 90.;                  /*É necessário o uso de *start.d e *stop.d */
known_track (CUBIC, DOUBLE, start, stop, "T0", &T0, clock);
(...)
```

/* Execução de known-track para o caso cúbico de reais:

Para i variando de 0 a n_ele = (Rate * (cue.stop - cue.start)) e x variando de 0 a 1, com passo 1/ n_ele faz-se:

```
{       aux = x3;
        SET_POINT (p_list[ i], 0., aux*(*stop.d - *start.d) + *start.d, 0. );
}
```

Chama-se a função apequs, que retorna uma lista de pontos (armazenada em arquivo) a partir dos pontos de controle em p_list (só a coordenada y dos pontos interessa, já que agora trabalha-se com reais):

```
apq = apequs ( p_list, n_ele, &n_ele );
```

Define-se também pára a trilha:

```
T0->n_ele = n_ele;
T0->done = TRUE;
T0->type = DOUBLE;
*/
```

OBS.: Quando usar o tipo COLOR, as variáveis *start.c e *stop.c devem ser igualadas a variáveis externas à cena (declaradas junto com TRACK ( )), já que a função SET_RGB, que dá valor às mesmas, é uma função que aparece no script.

free_track (String, ValueType, char[3], Track*, Watch) - esta função cria uma trajetória a partir de pontos de controle dados num arquivo. O nome desse arquivo (e o "path", se necessário) é o primeiro parâmetro desta função (o nome e o "path" do arquivo devem estar entre aspas). O segundo parâmetro novamente pode ser DOUBLE, POINT, COLOR ou VECTOR. O terceiro parâmetro é uma string com o nome da trilha ("T0", "T1", etc, na ordem em que elas são declaradas no programa). O quarto parâmetro é a variável trilha e *deve ser precedida de &*. Esta variável deve ser declarada como Track no início do programa. O último parâmetro é novamente o relógio local (usa-se a variável "clock" como este último parâmetro).

O funcionamento de free_track é semelhante ao de known_track, com a diferença de que agora os pontos de controle para a função apequs são dados em um arquivo, e não mais determinados por funções matemáticas conhecidas: para ler os valores deste arquivo, usa-se a função read_pc (que será vista ainda neste arquivo). Lida a lista de pontos de controle, chama-se a função apequs, que retornará valores que serão então armazenados no arquivo com o nome da trilha.

O arquivo de pontos de controle deve ter na primeira linha o número de pontos de controle e em cada uma das linhas seguintes os valores das coordenadas x, y e z de cada ponto ou vetor (ou valores r, g e b de cada cor). Se for uma trilha de DOUBLE, o valor usado será a coordenada y dos pontos gerados pela apequs.

Exemplo:

```
(...)
TRACK (T0);                    /* declaração da variável T0 como do
                .              tipo track. */
                .
                .
BEGIN_SCENE                    /* embora nesta função a declaração de
                .              TRACK também possa ser interna à cena */
                .
                .
free_track ( "in/pontos_ctr.in", COLOR, "T0", &T0, clock );
                                   /* os pontos de controle estão no arquivo
                                    pontos_ctr.in, no subdiretório in */
(...)
```

/* Execução desta função:

Inicialmente são dados valores iniciais para algumas variáveis e é lida a lista de pontos de controle através da função read_pc:

```
grain = 2/(clock.inv_dura * clock.d_time);
n_ele = 0;
read_pc ( "in/pontos_ctr.in", &n_ele);
```

Depois grain é definido como sendo o máximo entre grain e n_ele e é chamada a função apequs:

```
apq = apequs ( p_list, n_ele, &grain );
```

Finalmente, define-se a trilha com grain pontos: para i variando de 0 a grain. Define-se também para a trilha:

```
T0->n_ele = grain;
T0->done = TRUE;
```

```
T0->type = COLOR; */
```

apequs (Point *, int, *int) - esta função tem 3 parâmetros: o primeiro é uma lista de pontos de controle, o segundo é o número de pontos da lista e o terceiro é o número de pontos da b-spline calculada a partir da lista dada (este terceiro parâmetro deve ser precedido de &). Esta função não é usada diretamente pelo animador, mas é usada dentro das funções "known_track" e "free_track", para se obter uma nova lista de pontos calculada por uma b-spline cúbica aproximada, chamada apequs (cujo nome deriva de *a-pe*riódica *q*uase *u*niformemente espaçada, mostrando a diferença entre esta formulação e a b-spline tradicional.

bspline (Point *, int, *int) - outra aproximação matemática (bspline) para a lista de pontos de controle do trilho. Os parâmetros são idênticos ao da função "apequs". Esta aproximação representa a formulação clássica, cujos pontos são espaçados segundo o espaçamento dos pontos de controle.

OBS.: A curva obtida tanto pela função "apequs" como pela "bspline" é "uma sequência de *grain* pontos ordenados em um total de (*npc* - 3) segmentos consecutivos, onde *npc* é o número de pontos de controle definidos pelo animador - segundo parâmetro das funções - e *grain* é o número de pontos desejados para serem calculados - terceiro parâmetros das funções". /SILV-92/

read_pc (String *, int *) - esta função também não é usada diretamente pelo animador, mas é chamada durante a execução da função "free_track". Serve para ler o arquivo com os pontos de controle para uma trajetória. O primeiro parâmetro é o nome do arquivo e o segundo é uma variável que receberá o valor da primeira linha do arquivo (significando o número de pontos do arquivo). O segundo parâmetro deve vir precedido de &.O arquivo onde estão os pontos de controle deve ter a seguinte forma:

$$
\begin{array}{c}
n \\
x_1\ y_1\ z_1 \\
x_2\ y_2\ z_2 \\
(...) \\
x_n\ y_n\ z_n
\end{array}
$$

onde n é inteiro; x_i, y_i e z_i são reais.

following_track (A_R, char[3], Track *, double, Watch) - depois de gerada a trilha, é necessária uma função para o caminhamento sobre a mesma. É "following_track" que faz isto, independentemente do modo de geração da trilha.

O primeiro parâmetro indica se é um caminhamento absoluto (usa-se ABSOLUTE) ou relativo (usa-se RELATIVE):

"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 o ator andou neste último TICK do relógio, ou seja, 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 é o nome da trilha (na verdade, nome do arquivo no diretório /tmp onde está definida a trilha. O terceiro parâmetro é a variável trilha (antecedida por &). O quarto parâmetro é uma variável obtida a partir de uma função dinâmica (ver funções do arquivo m_dynam.c, neste módulo), que determina se o caminhamento será acelerado, desacelerado ou uniforme. O último parâmetro é um relógio, que determina o intervalo em que a trilha será percorrida (é aconselhável que seja o mesmo em que a ela foi criada, i.e., o último parâmetro de "known_track" ou "free-track").

see_track (char[3], Track *) - serve para visualizar a trilha descrita num arquivo dado pelo primeiro parâmetro (diretório /tmp), definida pela varíavel dada pelo segundo parâmetro. Assim, se no décimo quadro da animação se quiser ver a trilha T1, deve-se fazer:

```
At ( FRAME(10) ) {  see_track  ("T2", &T2); }
```

OBS.: Todas as funções deste arquivo existem no nível da cena, e não do script.

A seguir, são dadas duas animações completas, para facilitar a compreensão das funções deste módulo:

Animação 1:

/* A animação consiste no caminhamento ABSOLUTO de um cubo sobre uma trilha criada a partir de pontos de controle em um arquivo chamado pontos.in */

```
#include <stdio.h>
#include "p_transf.h"
#include "p_vector.h"
#include "p_alloc.h"
#include "p_error.h"
#include "p_graph.h"          /* No início de cada animação, deve-se incluir todos os */
#include "p_poly3d.h"         /*arquivos.h necessários. */
#include "camera.h"
#include "actor.h"
#include "time.h"
#include "scanline.h"
#include "sl_wire.h"
#include "light.h"
#include "script.h"
#include "motion.h"

TRACK ( T0 );                 /* Declaração da variável  T0 como sendo do tipo
                              Track */

void cena1 (cue )             /* Procedimento "cena1", que vai ser chamado no script */
Cue *cue;                     /* e está associado a uma cue. */
{
double ddin;                  /* Variável dinâmica, usada como terceiro parâmetro de
                               "following_track". */
Point *p;                     /* É uma trilha de pontos. A variável p (deve ser declarada
                               como *p) vai ser o campo p da trilha gerada e servirá como
                               parâmetro para a função de movimento (translate).*/
```

```
Element *e;                          /* Variável que receberá o valor dado pela função
                                      "following_track" ( esta função retorna um Element ) */
BEGIN_SCENE
NO_CLOCK;

part ( "cubo" );                     /* O ator cubo participa da cena. */

At ( 0 )
{
 translate_actor ( ABSOLUTE, 0., 0., 0., "cubo" );
}                                    /* O cubo é inicializado no ponto ( 0, 0, 0 ). */

After ( 0 )
{
 free_track ( "pontos.in", POINT, "T0",  &T0, clock );
                                     /* É definida a trilha de pontos T0, a partir dos pontos de
                                      controle em "pontos.in". */
 ddin = linear ( ABSOLUTE, clock );
                                     /* A função linear será vista no arquivo m_dynamic, neste
                                      módulo. Ela faz com que o ator caminhe sobre a trilha com
                                      movimento uniforme. */
 e = following_track ( ABSOLUTE,  "T0", &T0, ddin, clock);
                                     /* Segue-se a trilha de maneira absoluta e com movimento
                                      uniforme ( ddin, que é definida com a função "linear" ). */
  p = e->p;                          /* Só o campo p do elemento "e" anteriormente definido
                                      interessa, já que se trata de uma trilha de pontos. A var iável
                                      p vai ser usada como parâmetro na função "translate" a
                                      seguir. */
 translate_actor ( ABSOLUTE, p.x, p.y, p.z, "cubo" );
                                     /* O cubo será transladado, absolutamente, para os pontos
                                      definidos pela trilha. */
}

END_SCENE
}                                    /* Fim da cena. */

BEGIN_SCRIPT (1)

set_studio ( NULL, 2, "../images/scan/animacao" );
                                     /*As imagens geradas se chamarão animacao e estarão no
                                      subdiretório ../images/scan. */
TotalTime = 6;
Rate = 10;                           /* A animação terá 60 quadros (  6 segundos numa taxa de
                                     10 quadros/s ). */
cast_poly ( "cubo", “cube.brp” );    /* Definição do ator cubo como um poliedro. */
sipp_render (PHONG_SIPP, 2, FALSE);
sipp_paint_actor ( "cubo", PHONGS, "home/leblon/leopini/user/tarcisio/cor/roxo.cor" );
                                     /* O cubo é pintado com a cor definida no arquivo roxo.cor
                                      do subdiretório especificado. Essa textura é do tipo PHONGS */
SET_CUE ( cue[ 0], FRAME ( 0 ), FRAME ( 60 ) );
                                     /* cue[ 0] é definida como começando em 0 e terminando no
                                      final da animação - FRAME ( 60 ). */

LIST_SCENES
        cena1 ( cue [ 0] );                  /* cena1 vai ocorrer no intervalo definido por cue [ 0]. */
END_LIST
```

```
END_SCRIPT                  /***** Fim da animação 1. *****/
```

Animação 2:

/* A animação consiste na rotação RELATIVA de um cubo roxo na direção x. O ângulo da rotação é dado por uma trilha_conhecida QUADRÁTICA de DOUBLE, variando de 0 a 90º. Existe na animação ainda um cubo amarelo que fica parado, servindo como referência. */

```
#include <stdio.h>
#include "p_transf.h"
#include "p_vector.h"
#include "p_alloc.h"
#include "p_error.h"
#include "p_graph.h"            /* No início de cada animação, deve-se incluir todos os */
#include "p_poly3d.h"           /*arquivos.h necessários. */
#include "camera.h"
#include "actor.h"
#include "time.h"
#include "scanline.h"
#include "sl_wire.h"
#include "light.h"
#include "script.h"
#include "motion.h"

static Track T0;                /* Declaração equivalente a "TRACK ( T0 )". */

void cena1 (cue )               /* Procedimento "cena1", que vai ser chamado no script */
Cue *cue;                       /* e está associado a uma cue. */
{
double ddin;                    /* Variável dinâmica, usada como terceiro parâmetro de
                                   "following_track". */
double *d;                      /* É uma trilha de reais. A variável d (deve ser declarada
                                   como *d) vai ser o campo d da trilha gerada e servirá como
                                   parâmetro para a função de movimento (rotate).*/
Element *e;                     /* Variável que receberá o valor dado pela função
                                    "following_track" ( esta função retorna um Element ) */
Element start, stop;            /* Elementos cujos campos "d" servirão como valores inicial
                                    e final para a trilha gerada. */

BEGIN_SCENE
NO_CLOCK;

part ( "cubo1" );
part ( "cubo2" );               /* Os atores cubo1 e cubo2 participam da cena. */

At ( 0 )                        /* Inicializações. */
{
 translate_actor ( ABSOLUTE, 0.5, 0., 0., "cubo1" );
 translate_actor ( ABSOLUTE, -0.5, 0., 0., "cubo2" );
}

start.d = ALLOC ( 1, double );
stop.d = ALLOC (1, double );    /* Alocação de memória para start.d e stop.d. */
```

```
*start.d = 0.0;
*stop.d = 90.0;                    /*A trilha de reais gerada irá de 0 a 90. */

known_track ( QUADR, DOUBLE, start, stop, "T0",  &T0, clock );
                                   /* É definida a trilha quadrática de reais (só a coordenada
                                    y dos pontos da trilha) anim2, tendo início em start.d e
                                    terminando em stop.d. */
ddin = slow_in ( RELATIVE, clock );
                                   /* A função slow_in será vista no arquivo m_dynamic, neste
                                    módulo. Ela faz com que o ator caminhe sobre a trilha com
                                    movimento acelerado. O fato de ser RELATIVE indica
                                    que, ao final da animação, o ator terá girado 90 o em relação
                                    ao  primeiro quadro ( ver arquivo m_dynamic ). */
e = following_track ( ABSOLUTE, "T0",  &T0, ddin, clock);
                                   /* Segue-se a trilha de maneira absoluta e com movimento
                                    acelerado ( ddin é definida com a função "slow_in" ). */
d = e->d;                          /* Só o campo d do elemento "e" anteriormente definido
                                    interessa, já que se trata de uma trilha de reais. A variável d
                                    vai ser usada como parâmetro na função "rotate". */
rotate_actor_x ( RELATIVE, *d, "cubo1" );
                                   /* O cubo1 será girado, relativamente, de ângulos definidos
                                    pela trilha. É importante saber, que se usa *d quando se
                                    tratar de uma trilha de reais. */
p_free(start.d);
p_free(stop.d);                    /* Libera a memória dinâmica. */

END_SCENE
}                                  /* Fim da cena. */

BEGIN_SCRIPT (1)

set_studio ( NULL, 2, "../images/scan/outra_anim" );
                                   /*As imagens geradas se chamarão outra_anim e estarão no
                                    subdiretório ../images/scan. */
TotalTime = 5;
Rate = 16;                         /*A animação terá 80 quadros ( 5 segundos numa taxa de
                                   16 quadros/s ). */

cast_poly ( "cubo1", /home/brp/cube.brp” );
cast_poly ( "cubo2", “/home/brp/cube.brp” );        /*Definição dos atores cubo1 e cubo2 como
                                                    poliedros. */

sipp_render ( FLAT_SIPP, 2, TRUE );
sipp_paint_actor ( "cubo1", PHONGS, “roxo.cor” );
sipp_paint_actor ( "cubo2", WOOD, “amarelo.cor” );
SET_CUE ( cue[ 0], FRAME ( 0 ), FRAME ( 80 ) );
                                   /* cue[ 0] é definida como começando em 0 e terminando no
                                    final da animação - FRAME ( 80 ). */

LIST_SCENES
        cena1 ( cue [ 0] );        /* cena1 vai ocorrer no intervalo definido por cue [ 0]. */
END_LIST

END_SCRIPT                         /***** Fim da animação 2. *****/
```

4.3.2: Arquivo m_dynamic.c:

As funções deste arquivo retornam valores reais que serão usados como parâmetros para o "caminhamento" sobre uma trilha (último parâmetro da função "following_track", como visto anteriormente). Entretanto, estas funções também podem ser usadas de maneira mais simples, sem a presença das trilhas (podem ser, por exemplo, o valor do ângulo numa rotação). Dependendo da função, o movimento a ela associado será uniforme, acelerado ou desacelerado.

linear (A_R, Watch) - função para movimento uniforme. Associada à função linear: f (t) = t. O primeiro parâmetro indica se a avaliação do tempo é absoluta ou relativa (ver OBS.). O segundo parâmetro é o relógio local da cena ("clock"). Um exemplo da utilização desta função é visto na animação 1 no arquivo anterior.

slow_in (A_R, Watch) - função que determina movimento acelerado. Está associada à função: f (t) = 1 - cos (90t). Os parâmetros são idênticos aos da função "linear". Esta função é utilizada na animação 2 no arquivo anterior.

slow_out (A_R, Watch) - determina movimento desacelerado. Associada à função: f (t) = sen (90t). Tem os mesmos parâmetros das funções anteriores.

acc_dec (A_R, Watch) - determina movimento acelerado até a metade do percurso (também metade do tempo) e depois desacelerado. Associada às funções:

f (t) = [1 - cos (180t)] / 2 , na primeira metade (t <= 0.5);
f (t) = sen[180*(t - 0.5)] / 2 + 0.5 , na segunda metade (t > 0.5).

dec_acc (A_R, Watch) - determina movimento desacelerado até a metade do percurso (também metade do tempo) e depois acelerado. Associada às funções:

f (t) = sen (180t) / 2 , na 1a. metade (t <= 0.5);
f (t) = {1 - cos[180*(t - 0.5)]} / 2 + 0.5 , na 2a. metade (t > 0.5);

Até aqui, as funções estudadas retornam valores normalizados (entre 0.0 e 1.0) para tempo normalizado. As funções que serão vistas a seguir, já não apresentam esta característica. Mas, antes de mais nada, deve ser dada uma explicação sobre o uso do ABSOLUTE e RELATIVE nas funções deste arquivo:

OBS.: Se for ABSOLUTE: o valor t das funções apresentadas vai ser o tempo local normalizado (variando de 0.0 a 1.0), dado por: Tlocal / dura.

Se for RELATIVE: o valor t é o tempo local normalizado menos um valor delta_time (delta_time = clock.d_time).

"A condição A_R para aceleração ou cinemática identifica 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/

Veja os exemplos:

Ex. 1:
```
din = linear ( ABSOLUTE, clock );
rotate_actor_x ( RELATIVE, 90*din, "cubo1");
```

Neste exemplo, o ator "cubo1" girará, a cada quadro, de um ângulo maior; no último quadro ele girará de 90º *em relação ao penúltimo quadro* (condição ABSOLUTE para a dinâmica). Se a condição da função linear fosse RELATIVE, teríamos uma rotação constante do cubo a cada quadro, sendo que no último quadro ele teria girado de 90º *em relação ao primeiro quadro.*

Ex. 2:
```
din = slow_out ( ABSOLUTE, clock );
e = following_track ( ABSOLUTE,  "T2", &T2, din, clock);
p = e->p;
translate_actor ( ABSOLUTE, p.x, p.y, p.z, "cubo1" );
```

Neste exemplo, T2 é uma trilha de POINT. Igualmente à animação 1 do arquivo anterior, a função dinâmica é usada com a condição ABSOLUTE. Desta maneira o ator será deslocado a cada quadro para o ponto definido pela trilha. Se a condição fosse RELATIVE, o resultado ficaria difícil de se prever.

Existem também condições A_R nas funções "following_track", e nas transformações geométricas lineares rígidas (translate, rotate, etc. - ver módulo actor.h). Por isso, "a mesclagem de referencial no caminhamento e na transformação geométrica, assim como o uso de dinâmica de referencial relativo, podem levar a efeitos interessantes, mas raramente previsíveis". /SILV-92/

acc_motion (A_R, double, double, Watch) - determina movimento acelerado através da função: $f(t) = Vo*t + 0.5*a*t^2$, onde "Vo" (velocidade inicial) é o segundo parâmetro da função e "a" (aceleração) é o terceiro parâmetro. O primeiro parâmetro e o último, como nas funções anteriores, são: a condição ABSOLUTE / RELATIVE e o relógio interno da cena, respectivamente. Como já foi dito, mesmo com o tempo "t" normalizado, a função poderá retornar valores não normalizados. Por exemplo:

```
din = acc_motion ( RELATIVE, 1.5, 1.0, clock );
rotate_actor_x ( RELATIVE, 90*din, "cubo1");
```

Neste exemplo, "din" será dada por: $1.5t + 0.5t^2$. Usando o tempo normalizado, ao final da cena (t = 1), a variável din será igual a 2.0. Assim, o cubo terá girado 180º em relação à sua posição no início da cena. O quanto o cubo girará também depende, neste caso, dos valores de "Vo" e "a".

dec_motion (A_R, double, double, Watch) - exatamente igual à anterior, determinando o movimento desacelerado ("a" negativo). Deste modo:

```
dec_motion ( ar, Vo, a, clock ) = acc_motion ( ar, Vo, -a, clock )
```

free_dynamic (A_R, String *, Track *, Watch) - esta função permite que o animador crie sua própria curva de dinâmica a partir de pontos de controle dados em um arquivo (o nome do arquivo é o segundo parâmetro da função). A forma do arquivo de entrada é a mesma do arquivo para a função "free_track", no m_track.c.

O terceiro parâmetro é o nome da trilha gerada pelos pontos de controle e, assim como em "free_track", *deve ser precedido de &.*

O primeiro parâmetro e o último, como nas funções anteriores, são: a condição ABSOLUTE / RELATIVE e o relógio interno da cena, respectivamente.

O funcionamento é parecido com o da função "free_track": o arquivo é lido; depois é chamada a função apequs, que retornará uma lista de "grain" pontos. Após isso, diferentemente da função "free_track", os parâmetros são normalizados (a função retornará

valores entre 0 e 1). Finalmente então, são passados para a trilha especificada. Se a função for usada de maneira absoluta, ela retorna, a cada instante, a coordenada y de um ponto da trilha, já que funções dinâmicas retornam valores reais. Se for usada de maneira relativa, ela retorna a diferença entre as coordenadas y de dois pontos consecutivos da trilha.

A seguir será dada mais uma animação completa, usando esta função:

Animação 3:

```
/* A animação consiste na  rotação RELATIVA de uma pirâmide lilás, com
eixo de rotação z. O arquivo de entrada para a função dinâmica  ("dados.in") é
dado a seguir:
                    4
                    0.0 3.0 0.0
                    0.0 4.0 0.0
                    0.0 6.0 0.0
                    0.0 9.0 0.0                    */

#include <stdio.h>
#include "p_transf.h"
#include "p_vector.h"
#include "p_alloc.h"
#include "p_error.h"
#include "p_graph.h"
#include "p_poly3d.h"
#include "camera.h"
#include "actor.h"
#include "time.h"
#include "scanline.h"
#include "sl_wire.h"
#include "light.h"
#include "script.h"
#include "motion.h"

TRACK ( T0 );                   /* Declaração da trilha T0. */

void cena1 (cue)                /* Procedimento "cena1", que vai ser chamado no script */
Cue *cue;                       /* e está associado a uma cue. */
{

BEGIN_SCENE
NO_CLOCK;

part ( "pir" );                 /* O ator "pir" participa da cena. */

translate_actor ( ABSOLUTE, 1., 1., 0., "pir" );
                                /* A pirâmide é inicializada no ponto ( 1, 1, 0 ). */
END_SCENE
}

void cena2 (cue )               /* Procedimento "cena2", que vai ser chamado no script */
Cue *cue;                       /* e está associado a uma cue. */
{
double ddin;                    /* Variável dinâmica, usada como parâmetro na rotação. */
```

```
BEGIN_SCENE
NO_CLOCK;

part ( "pir" );

ddin = free_dynamic ( RELATIVE, "dados.in", &T0, clock );
rotate_actor_z ( RELATIVE, 90*ddin, "pir");
                                /* Ao final da cena, a pirâmide terá girado 90 º em relação
                                ao primeiro quadro da mesma. Pelo arquivo de entrada,
                                 percebe-se que o movimento foi acelerado, já que
                                a diferença entre as coordenadas y dos pontos de
                                controle aumentava os pontos de controle criaram
                                uma curva de dinâmica acelerada). */
END_SCENE
}

BEGIN_SCRIPT (2)                /* O script tem 2 cenas */

set_studio ( NULL, 3, "../images/scan/girapir" );
                                /*As imagens geradas se chamarão girapir e estarão no
                                 subdiretório ../images/scan. */
TotalTime = 4;
Rate = 16;                      /* A animação terá 64 quadros ( 4 segundos numa taxa de
                                16 quadros/s ). */

cast_poly ( "pir", “pyr.brp” );         /* Definição do ator pir como um poliedro. */

sipp_render ( GOUR_SIPP, 1, TRUE);
sipp_paint_actor ( "pir", BASIC, “lilas.cor” );

SET_CUE ( cue[ 0], 0.0, 0.0 );  /* cue[ 0] é definida como o instante 0 ( começa e termina
                                  em 0 ). */
SET_CUE ( cue[ 1], FRAME ( 0 ), FRAME ( 64 ) );
                                /* cue[ 1] é definida como começando em 0 e terminando no
                                 final da animação - FRAME ( 64 ). */

LIST_SCENES
        cena1 ( cue [ 0] );             /* cena1 vai ocorrer no intervalo definido por cue [ 0]. */
        cena2 ( cue [ 1] );             /* cena2 vai ocorrer no intervalo definido por cue [ 1]. */
END_LIST

END_SCRIPT                      /***** Fim da animação 3. *****/
```

4.3.3: Arquivo m_math.c:

As funções deste arquivo são puramente matemáticas e auxiliam no cálculo das curvas matemáticas para a geração das trilhas. Elas não vão ser usadas diretamente pelo animador, pois apenas vão ser chamadas dentro de outras funções (por exemplo "apequs" e "bspline"). Por esta razão, elas não vão ser muito detalhadas.

mult_1x4_4x4 (double*, double, Matrix) - multiplica um vetor 1x4 (segundo parâmetro) por uma matriz 4x4 (último parâmetro). O primeiro parâmetro é o vetor 1x4 que receberá o resultado. O tipo Matrix é definido no ProSIm - ver apêndice A.

mult_1x4_4x3 (double*, double*, double*) - multiplica um vetor 1x4 (segundo parâmetro) por uma matriz 4x3 (último parâmetro). O primeiro parâmetro é o vetor 1x3 que receberá o resultado.

mult_1x4_k (double*, double*, double) - multiplica um vetor 1x4 (segundo parâmetro) por um número real (último parâmetro). O primeiro parâmetro é o vetor 1x4 que receberá o resultado.

Gs (double*, int, Point*) - o primeiro parâmetro é uma matriz 4x3 que, em cada linha, vai receber às coordenadas dos pontos pc[i-1], pc[i], pc[i+1] e pc[i+2]; onde "i" é o segundo parâmetro e "pc" é o terceiro (uma lista de pontos). Determina os quatro pontos consecutivos que vão ser interpolados para a obtenção de uma curva cúbica, que é um segmento da spline.

p_i_u (double, Matrix, double*) - retorna o seguinte resultado:

([u^3 u^2 u 1] / 6) * A * B, onde u é o primeiro parâmetro; A é uma matriz 4x4 (segundo parâmetro); e B é uma matriz 4x3 (terceiro parâmetro).

Seguindo a formulação matricial para B_Spline cúbica detalhada em /MORT-85/, teremos a matriz A especificada e, se a matriz B for o resultado de uma função Gs, acharemos os pontos desejados na B_Spline. Ver também /FOLEY-90/.

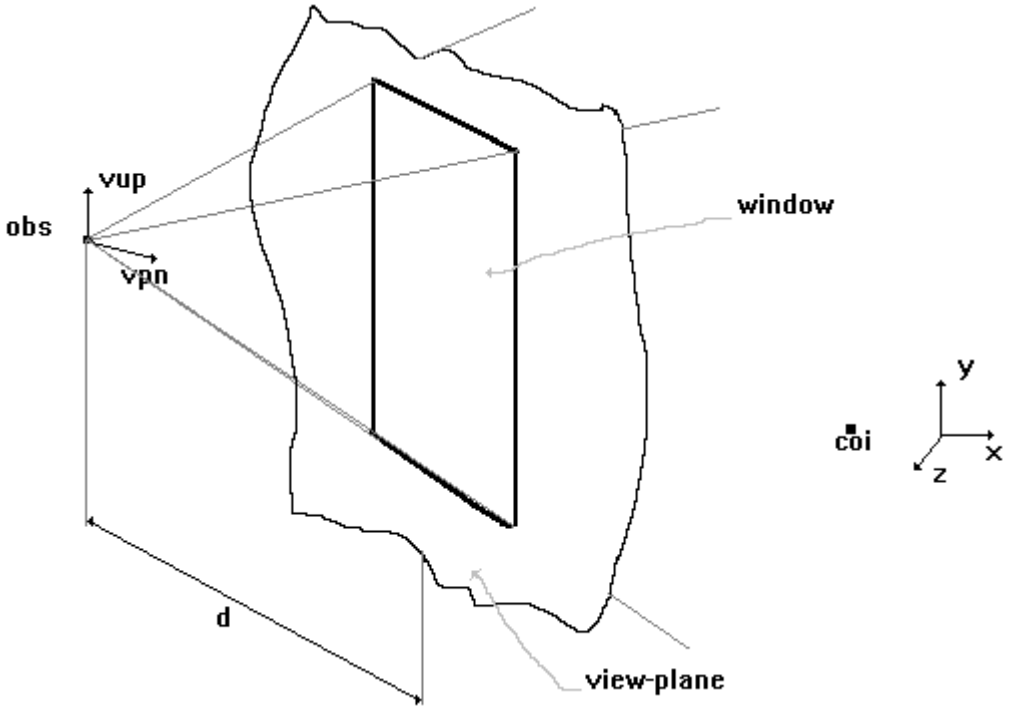
5 - MÓDULO CAMERA.H:

A câmera sintética é um elemento que compõe a cena, mas não é visível na imagem, diferentemente dos atores, das decorações e dos movimentos. A câmera apenas influencia o modo de visualizar; entretanto, pode-se produzir animações inteiras usando somente recursos da câmera (sem a necessidade de movimentação dos atores).

É neste módulo que se encontra definida a estrutura de dados que caracteriza a câmera sintética.

As funções que transformam a estrutura "camera" estão em 7 arquivos, separados de acordo com o elemento da câmera (observador, centro de interesse, etc.) que elas manipulam. Em cada um dos sub-ítens 5.3. deste capítulo, os arquivos com as funções serão melhor especificados.

Modelo da câmera:



Da figura anterior:

* **obs** é a posição do **obs**ervador (quando em projeção perspectiva é o centro de projeção).

* **coi** - **c**enter **o**f **i**nterest - é o centro da atenção do observador (um ponto da cena para o qual ele está olhando).

* **vup** - **v**iew **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 **d**istance - é a distância entre o observador e o plano de visualização.

* **vpn** - identifica a direção de visualização, dado por (vpn = coi - obs).

Além desses, ainda existemparâmetros que definirão 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/.

5.1 - Estruturas do camera.h:

ProjectionType - enumeração que especifica o tipo de projeção desejada (pode ser perspectiva ou ortogonal). Da listagem do programa:

```
typedef enum { ORTHO, PERSP } ProjectionType;
```

Camera - estrutura capaz de modelar as seguintes funcionalidades da câmera sintética: permitir a visualização arbitrária; executar a projeção da imagem; estabelecer qual parte da imagem será mostrada e em que parte da tela. Para isto, esta estrutura tem como parâmetros: os do modelo da câmera e do enquadramento (já citados neste capítulo), além do tipo de projeção a ser realizada. Da listagem do programa:

```
typedef struct {
        /******** Observer *********/
        Point     obs,
                  coi,
                  vup;
        double    d;
        /******** Window in the ViewPlane *******/
        double    u_min, u_max,
                  v_min, v_max;
        /******* Viewport on the screen **********/
        int       x_corner, y_corner,
                  x_length, y_length;       /* a origem é o canto superior
                                               esquerdo */
        /******** Projection parameter **********/
        ProjectionType prj;
        } Camera;
```

Os parâmetros obs, coi e vup definem, respectivamente, o observador, o centro de interesse e a direção vup (parâmetros do modelo da câmera).

O real d define a distância focal.

Os reais u_min, u_max, v_min e v_max definem a janela ("window", necessária para o enquadramento).

Os inteiros x_corner, y_corner, x_length e y_length definem a porta de visualização ("viewport"), que é outro parâmetro de enquadramento.

Finalmente, o último parâmetro é do tipo ProjectionType e define o tipo de projeção a ser realizada (ortogonal - ORTHO - ou perspectiva - PERSP).

5.2 - Macros do camera.h:

EPSILON - valor real definido como sendo 10^{-20}.

Quase0 (x) - macro usada quando se quer saber se um valor está muito próximo de zero (com diferença menor que EPSILON).

Y_ASPECT_RATIO - definido como sendo 0.75 .
X_ASPECT_RATIO - definido como sendo 1.0 .

X_SC_MAX - define o número de *pixels* por scan-line. Definido como 1024.
Y_SC_MAX - define o número de scan-lines. Definido como sendo 1024.

Obs.: Estas quatro últimas macros serão utilizadas apenas no scanline, que está fora do interesse deste texto. Para maiores informações, ver /PRET-93/.

5.3 - Funções do camera.h:

5.3.1: Arquivo c_define.c:

Neste arquivo estão as funções que definem e inicializam a câmera, atribuindo valores aos parâmetros da mesma e adequando-a às necessidades de visualização. Existe ainda uma função de inicialização default e uma função que fornece a listagem dos valores atuais dos parâmetros da câmera.

set_obs (double, double, double) - função que inicializa a posição do observador (campo .obs da estrutura Camera), em coordenadas cartesianas. Os três parâmetros são, respectivamente, as coordenadas x, y e z 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. Por exemplo:

```
set_obs ( 1., 2., 0. );
```

indica que o observador está no ponto (1, 2, 0).

set_vup (double, double, double) - função que atualiza a posição de vup (indica a direção do que é para cima) na posição desejada. Os três parâmetros são, respectivamente, as coordenadas do vetor vup nas direções x, y e z. A função retornará mensagens de erro se:

1. Os três parâmetros forem nulos (vup não pode ser um vetor nulo);
2. vup for colinear a vpn (onde vpn = coi - obs);

Caso não haja erro, calcula-se o vetor normal a vup e vpn e acha-se vup ortogonal por: vup = vetor normal X vpn. Este valor é colocado no campo .vup da estrutura Camera.

set_coi (double, double, double) - atualiza a posição do centro de interesse do objeto, em coordenadas cartersianas. Os parâmetros são, respectivamente, as coordenadas x, y e z do coi. Estes valores serão colocados no campo .coi da estrutura Camera. A função também verifica se o centro de interesse coincide com o observador; se isto acontecer, aparece uma mensagem de erro. Se esta inicialização preceder à set_obs, a verificação será feita com relação à origem, visto que o observador é inicializado na origem pela linguagem. Por exemplo:

`set_coi ( 0., 1., 0. );` estabelece que o coi é o ponto (0, 1, 0).

set_d (double) - atualiza a distância focal (distância entre o centro de observação e o plano de visualização). O parâmetro da função é o novo valor da distância focal. O valor do parâmetro não pode ser zero.

set_viewport (int, int, int, int) - função que instancia a viewport - parte da tela que será efetivamente usada para mostrar a imagem. Os parâmetros são dados em coordenadas de tela do dispositivo (DC - device coordinates) e são respectivamente: x_corner, x_length, y_corner e y_length. Assim:

`set_viewport ( 5, 300, 5, 300 );` permite a visão da animação em janelas de 300x300 (DC) na tela.

set_window (double, double, double, double) - instancia a window (parte do plano de projeção que contém a imagem que nos interessa). Os parâmetros são, respectivamente: u_min, u_max, v_min e v_max. A dimensão destes parâmetros deve ser compatível com o valor do parâmetro "d", para garantir um cone de projeção perspectiva coerente.

set_aperture (double, double) - esta função permite que, através do ângulo de abertura da pirâmide de visualização, se possa definir a janela (window), sem a preocupação com o valor de "d". Os dois parâmetros são os valores em graus dos ângulos nas direções u e v, respectivamente. A função primeiramente verifica o tipo de projeção, pois só é permitido em projeção perspectiva. A seguir, converte os valores de graus para radianos e calcula os parâmetros para definir a janela:

```
u_max = d * tan ( u_radianos );      u_min = - u_max;
v_max = d * tan ( v_radianos );      v_min = - v_max;
```

set_persp ( ) - seta o campo .prj (tipo de projeção) da estrutura Camera em "PERSP" (projeção perspectiva). Não possui parâmetros.

set_ortho ( ) - seta o campo .prj da estrutura Camera em "ORTHO" (projeção ortogonal). Não possui parâmetros.

set_cam_defaults ( ) - seta os parâmetros da câmera em valores default, usando as funções set_..., vistas anteriormente. Não possui parâmetros. Os valores default são:

```
set_obs ( 1.5, 1.5, 1.5 );
set_coi (0., 0., 0. );
set_vup (0., 1., 0. );
set_d ( 1.0 );
set_persp ( );                          /* proj. perspectiva */
set_viewport ( 5, 100, 5, 100 );
set_aperture ( 30., 30. );
```

camera_list ( ) - mostra na tela os valores atuais dos parâmetros da câmera. Não possui parâmetros. Esta função vai ser chamada internamente à função "shoot" (ver módulo script.h).

OBS.:

1. As funções set_... são usadas a nível de script, logo após a função "set_studio" (ver módulo script.h), para modificar parâmetros da câmera que tenham sido estabelecidos através de "set_studio".

2. A ordem de inicialização não é fixa, mas trocá-la pode levar a resultados diferentes. *É bom garantir que set_vup venha depois de set_obs e set_coi e também que set_aperture venha depois de set_d e set_ortho (ou set_persp).*

5.3.2: Arquivo c_observ.c:

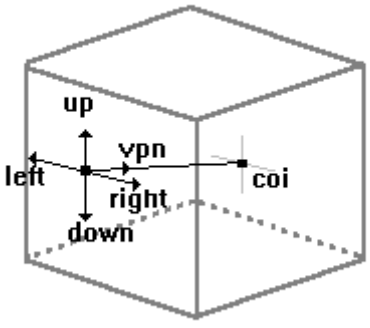
Neste arquivo estão as funções que manipulam o parâmetro "observador" da câmera. Todas as funções deste arquivo começam com "go_" .

go_forward (double) - move o observador para frente, na direção do centro de interesse. O parâmetro fornece o valor, em unidades, deste movimento. Seu efeito é o de um zoom, aumentando os detalhes da cena próximo ao coi. Para isso, é calculado o vetor vpn (vpn = coi - obs), que é então normalizado e multiplicado pelo valor do parâmetro. O resultado é então somado com o ponto onde o observador estava, resultando na nova posição do observador.

Se esta função for usada de tal modo que o observador ultrapasse o coi, o observador verá as "costas" do objeto.

go_backward (double) - reverso de "go_forward", ou seja afasta o observador do centro de interesse. O parâmetro é o valor de quanto o observador se afastará do coi. A função calcula o valor de vpn, normaliza-o e multiplica pelo valor do parâmetro. O resultado é subtraído do ponto onde o observador estava, resultando na nova posição do observador.

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 a seguir:



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 Coordinate).

go_up (double) - desloca o observador na direção de vup. O valor deste deslocamento é dado pelo parâmetro da função. Para isso, vup é multiplicado pelo valor do parâmetro e a este valor se soma a posição anterior do observador, obtendo-se a nova posição do mesmo. Finalmente, atualiza-se vup, evitando que ele perca sua direção realmente desejada.

Seu uso repetitivo faz com que o observador passe a olhar o observador em coi, cada vez mais de cima.

go_down (double) - desloca o observador na direção oposta de vup. O parâmetro determina o valor deste deslocamento. Seu funcionamento é semelhante ao da função anterior, sendo que, ao invés de somar, subtrai-se o valor obtido da posição anterior do observador.

Seu uso repetitivo faz com que o observador passe a olhar o objeto em coi, cada vez mais de baixo.

go_right (double) - 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. A função calcula vpn, normaliza-o e o multiplica por vup; o resultado é normalizado e somado com o ponto em que o observador estava, obtendo-se a nova posição do mesmo. Finalmente, o valor de vup é atualizado.

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 (double) - semelhante à função "go_right", só que deslocando o observador para a esquerda. O funcionamento também é semelhante ao da anterior, sendo que o valor obtido é subtraído da posição anterior do observador.

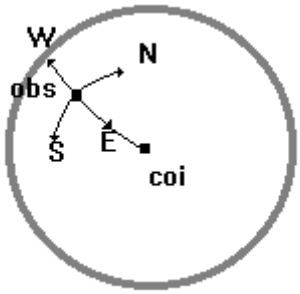
Seu uso repetitivo provoca um deslocamento espiral crescente para a esquerda.

Exemplos:

go_up (3.); provoca deslocamento do observador em 3 unidades na direção vup.

go_left (4.5); provoca o deslocamento do observador em 4.5 unidades 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, go_west, como indica a figura a seguir:



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 (double) - Desloca o observador na direção Norte. O valor em graus deste deslocamento é dado pelo parâmetro da função. Inicialmente, a função calcula vpn normalizado e o raio. Depois, calcula a nova posição do observador para garantir o deslocamento desejado. Determina então o novo vpn e recalcula a posição do observador, garantindo que esta tenha raio igual à anterior. Finalmente, recalcula um novo valor para vup, a partir do anterior.

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 -90º e +90º).

go_south (double) - semelhante a "go_north", sendo que o deslocamento se dará de θ graus (onde θ é o valor do parâmetro) na direção oposta (Sul). Funciona usando "go_north" com o parâmetro negativo: go_south (teta) = go_north (-teta). Por exemplo:

```
go_south ( 18. ) = go_north ( -18. );
```

go_east (double) - Desloca o observador na direção Leste (direita). O valor em graus deste deslocamento é dado pelo parâmetro da função. Inicialmente, a função calcula vpn normalizado e o raio, achando então o vetor normal a vup e vpn (vetor que indica a direção Leste). Depois, calcula a nova posição do observador para garantir o deslocamento desejado. Determina então o novo vpn e recalcula a posição do observador, garantindo que esta tenha raio igual à anterior. Finalmente, recalcula um novo valor para vup, a partir do anterior.

Seu uso repetitivo provoca uma volta sobre o objeto (coi) sem modificar a distância entre o coi e o observador.

go_west (double) - semelhante a "go_east", sendo que o deslocamento se dará de θ graus (onde θ é o valor do parâmetro) na direção oposta (Oeste). Funciona usando "go_east" com o parâmetro negativo: go_west (teta) = go_east (-teta). Dessa maneira:

```
go_west (50.) = go_east (-50.);
```

go_flying (double, double, double) - permite o deslocamento do observador para uma posição arbitrária (semelhante a "set_obs"), mas garantindo que a direção de visualização seja recalculada, para se ajustar à direção deste movimento. Os parâmetros da função são as três coordenadas (x, y e z, respectivamente) 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.

A função testa se existe movimento e, existindo, calcula a distância entre o coi e o observador. Depois, seta o observador no valor desejado. Recalcula então na direção do movimento, à mesma distância que antes. Finalmente, reseta vup com o valor antes calculado.

go_crow (double, double, double) - "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 nas direções x, y e z são dados, nesta ordem pelos parâmetros da função.

Produz o efeito do carrinho, ou "traveling", para os animadores convencionais. Por exemplo:

```
go_crow ( 0., 3., -1. );
```

faz com que o observador e o coi se desloquem de 3 unidades na direção y e 1 unidade na direção -z.

OBS.:

1. As funções "go_..." são usadas a nível de cena, não de script.

2. Tanto sobre o cubo (rigth, left, up, down) quanto sobre a esfera (north, south, east, west), as transformações "go_forward" e "go_backward" funcionam coerentemente.

3. Se as funções "go_..." forem usadas dentro de um loop de tempo (After, Between, Before), elas serão executadas uma vez a cada quadro. Por exemplo, o fragmento de uma animação:

```
After ( FRAME ( 15 ) )
{       go_west ( 60. );        }
```

Neste exemplo, a partir do quadro 16, o observador se deslocará, a cada quadro, 60 graus na direção Oeste.

A função "go_traveling" é uma exceção, pois só será executada uma vez dentro de um loop de tempo.

5.3.3: Arquivo c_aimpnt.c:

Neste arquivo estão as funções que manipulam o parâmetro "coi" ("center of interest" ou "aimpoint") da câmera. Todas as funções deste arquivo começam com "aim_".

aim_up (double) - função que desloca o centro de interesse na direção vup. O valor do deslocamento é dado pelo parâmetro da função. Para isso, a função multiplica coi pelo valor do deslocamento e adiciona ao antigo coi; depois, reinicializa vup. Por exemplo:

```
(...)
set_vup ( 0., 1., 0.);         /* vup é direção y. */
set_coi ( 0., -2, 3. );        /* coi é ( 0, -2, 3 ). */
aim_up ( 3. );                 /* novo coi será (0, -2, 3) + 3*(0,1,0) = (0,1,3). */
```

aim_down (double) - desloca o centro de interesse na direção oposta a vup. O valor do deslocamento é dado pelo parâmetro da função. Para isso, a função multiplica coi pelo valor do deslocamento e subtrai do antigo coi; depois, reinicializa vup. Por exemplo:

```
(...)
```

```
set_vup ( 0., 1., 0.);        /* vup é direção y. */
set_coi ( 0., -2, 3. );       /* coi é ( 0, -2, 3 ). */
aim_up ( 3. );                /* novo coi será (0, -2, 3) - 3*(0,1,0) = (0,-5,3). */
```

aim_in (double) - função que aproxima o coi do observador de um número de unidades especificado pelo parâmetro. Para isso, após a determinação de vpn (onde vpn = coi - obs) e sua normalização, este é multiplicado pelo valor do parâmetro e o resultado é subtraído de coi.

OBS.: 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 (double) - afasta o coi do observador de um número de unidades especificado pelo parâmetro. Para isso, após a determinação de vpn (vpn = coi - obs) e sua normalização, este é multiplicado pelo valor do parâmetro e o resultado é somado ao coi.

OBS.: A primeira vista, essa função parece idêntica a "go_backward", sendo que quem se move é o coi. Entretanto, afastar o observador significa ver a animação mais de longe e afastar o coi pode apresentar efeito visual diferente (por exemplo, se o coi se afastar na reta que liga o observador a ele, nenhuma alteração visual ocorre).

aim_right (double) - desloca o centro de interesse para a direita do observador, paralelamente ao plano de visualização. O valor do deslocamento é dado pelo parâmetro da função. Para isso, é determinado vpn normalizado, que é multiplicado pelo vetor vup, sendo o resultado normalizado. Este valor é multiplicado pelo valor do deslocamento e somado ao coi.

aim_left (double) - desloca o centro de interesse para a esquerda do observador, paralelamente ao plano de visualização. O valor do deslocamento é dado pelo parâmetro da função. Para isso, é determinado vpn normalizado, que é multiplicado pelo vetor vup, sendo o resultado normalizado. Este valor é multiplicado pelo valor do deslocamento e subtraído de coi.

aim_north (double) - desloca o centro de interesse de teta graus (onde teta é o valor do parâmetro) na direção *norte*, sobre uma esfera com centro no observador. A função calcula vpn normalizado e o raio, calcula o novo coi, considerando o ângulo teta. Depois, recalcula coi, garantindo que obs-coi = raio e reseta vup.

aim_south (double) - semelhante a "aim_north", mas na direção*sul*. Assim:

```
aim_south ( teta ) = aim_north ( -teta )
```

aim_east (double) - desloca o centro de interesse de teta graus (onde teta é o valor do parâmetro) na direção *leste*, sobre uma esfera com centro no observador. A função calcula vpn normalizado e o raio, calcula a normal de vpn e vup e calcula o novo coi, considerando o ângulo teta. Depois, recalcula coi, garantindo que obs-coi = raio e reseta vup.

aim_west (double) - semelhante a "aim_east", mas deslocando na direção *oeste* sobre a esfera centrada no observador. Assim:

```
aim_west ( teta ) = aim_east ( -teta )
```

OBS.:

1. As funções "aim_..." deste arquivo são usadas a nível de cena, não de script.

2. As funções "aim_north/south/east/west" são parecidas com as funções "go_north/south/east/west". Entretanto, existem diferenças sutis, no que diz respeito ao movimento do coi e do observador.

5.3.4: Arquivo c_mix.c:

Neste arquivo estão as funções que aplicam os cálculos da visualização arbitrária sobre o elemento a ser visualizado, no caso atores. É a parte do modelo de câmera que se integra (mistura - mix) à estrutura de representação do objeto. É o elo de ligação e aplicação do modelo de câmera aos objetos de um determinado sistema. Cada sistema pode e deve alterar este arquivo parcialmente para adequá-lo às suas necessidades.

obs_actor(String) - 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.

aim_actor (String) - função que especifica um ator que será o centro de interesse. O parâmetro é o nome do ator cujo centro de gravidade vai ser o centro de interesse. Funciona simplesmente calculando o centro de gravidade do ator em questão através da função gc_actor (ver módulo actor.h, arquivo a_miscel.c) e colocando este ponto como coi.

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 gravidades dos atores: gc_geral = (Σ gc_actors) / n_actors. Não possui parâmetros. É uma maneira de "centralizar" a visualização de uma cena com muitos atores.

OBS.:

1. As funções deste arquivo são usadas a nível de cena.

5.3.5: Arquivo c_standr.c:

Arquivo que contém funções com nomes mneumônicos em relação aos termos usados em cinema. Estas funções são implementadas, na maioria, por funções dos arquivos anteriores. É interessante perceber a relação entre a função e seu mneumônico, afim de melhor compreender ambos.

zoom_in (double) - 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 da função). 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: aproximação do observador (go_forward) ou modificação na forma de visualização (zoom_in).

zoom_out (double) - efetua um encolhimento nos detalhes da imagem, através da alteração do tamanho da window (mantendo viewport constante). O tamanho da janela é modificado considerando a porcentagem desejada (parâmetro da função). É o oposto da função "zoom_in", assemelhando-se a "go_backward".

spin_clock (double) - modifica (roda) a direção de vup de θ graus (onde θ é o parâmetro) 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 de θ graus (é semelhante ao efeito de torcer a cabeça para visualizar a cena). Para isso, a função determina vpn e o multiplica por vup, com o resultado determina o novo vup, considerando o ângulo θ.

spin_Cclock (double) - idêntica a "spin_clock", sendo que a rotação é feita no sentido anti_horário. Funciona usando "spin_clock" com o parâmetro negativo. Assim:

spin_Cclok (45.) = spin_clock (-45.)

tilt_up (double) - idêntica a "aim_north". Definida pelo movimento de elevar a cabeça para alcançar a visualização de alguma coisa acima (na vertical). Assim:

tilt_up (teta) = aim_north (teta)

tilt_down (double) - idêntica a "aim_south". Definida pelo movimento de abaixar a cabeça para alcançar a visualização de alguma coisa abaixo (na vertical). Assim:

tilt_up (teta) = aim_south (teta) = aim_north (-teta)

pan_right (double) - idêntica a "aim_east". Executa um movimento para visualização na linha do horizonte (linha panorâmica) para a direita. Desse modo:

pan_right (teta) = aim_east (teta)

pan_left (double) - idêntica a "aim_west". Executa um movimento para visualização na linha do horizonte (linha panorâmica) para a esquerda. Desse modo:

pan_left (teta) = aim_west (teta) = aim_east (-teta)

do_ortho ( ) - função que modifica, se necessário, os parâmetros da câmera para adequá-los à projeção ortogonal. Primeiramente, ela verifica se já está em projeção ortogonal; caso não esteja, os parâmetros são modificados, calculando o centro de gravidade geral da cena (função gc_general, do módulo miscel.h). Em seguida, são calculados os novos parâmetros para a window, para adequá-la à posição do coi (e não a uma distância "d" do observador). Não possui parâmetros.

do_persp ( ) - função que modifica, se necessário, os parâmetros da câmera para adequá-los à projeção perspectiva. Primeiramente, ela verifica se já está em projeção perspectiva; caso não esteja, os parâmetros são modificados, calculando o centro de gravidade geral da cena (função gc_general, do módulo miscel.h). Em seguida, é calculada a nova distância focal "d" e, finalmente, são calculados os novos parâmetros para a window, afim de trazê-la para a posição "d", a partir do observador.

Obs.: As funções "do_ortho" e "do_persp" não precisam ser usadas diretamente pelo animador, pois são usadas internamente à função "shoot", do arquivo s_define.c, do módulo script.h.

traveling (double, double, double) - o mesmo que "go_crow": modifica a posição do observador de dx, dy e dz (respectivamente, os parâmetros da função), sem modificar a direção de visualização vpn; ou seja, a mesma translação é aplicada ao observador e ao coi. Assim:

traveling (dx, dy, dz) = go_crow (dx, dy, dz)

5.3.6: Arquivo c_arbviw.c:

Arquivo com funções responsáveis pelo cálculo da matriz de visualização (NormalizedView) que, aplicada aos objetos de uma cena, permite uma visualização dos mesmos em uma posição arbitrária e com a linha central do eixo z de visualização. Também contém funções do "pipeline de visualização".

camera ( ) - função que gera uma matriz final (NormalizedView) que será construída pela acumulação das matrizes (ver /FOLE-90/):

NormalizedView = T(-obs).Ry.Rx.Rz.SH.Tz(-d).MPER

Etapas da execução da função:

1. Translada-se o sistema para a origem, tomando como base a posição do observador (obs), que findara sendo (0,0,0). T é responsável pelo pseudo ZOOM;

2. Roda-se em torno de y com teta negativo até vpn (ou coi) ficar no plano yz. Ry é responsável pelo PAN.

3. Roda-se em torno de x com fi positivo para que vrp (ou vpn) coincida com -z do sistema de coordenadas da mão direita. Rx é responsável pelo TILT.

4. Roda-se vup em torno de z com alfa negativo para que vup fique coincidente com o eixo y. Rz é responsável pelo SPIN.

5. Executa-se o SHEAR, para que a linha central do volume de visualização seja o próprio eixo z.

Nesta implementação:

vpn = (coi - obs) / norma e

vrp = vpn * d + obs.

Como vpn é o eixo z e obs é a origem: vrp = (0, 0, d).

Depois da quinta etapa, a window estará centrada em z:

-sx <= xw <= sx e -sy <= yw <= sy ,

onde sx = 0.5 (u_min + u_max) e sy = 0.5 (v_min + v_max).

Esta função não possui parâmetros e não será usada diretamente pelo animador, pois é chamada internamente em outras funções (tais como "set_studio", do módulo script.h e "set_camera", do arquivo c_scan.c, deste módulo).

border_viewport ( ) - função que, se usada no script, desenha um retângulo (borda) em torno da viewport, se em ambiente DOS, PC. Para isso, ela utiliza funções do ProSIm (por esta razão, seu funcionamento não será detalhado).

window_viewport (Point, Camera) - função que retorna um novo ponto, calculado a partir dos parâmetros da window e da viewport da câmera (segundo parâmetro) e a partir do ponto p (primeiro parâmetro); convertendo da window para a viewport o referido ponto.

É preciso ter uma window centrada em -sx <= x <= sx e -sy <= y <= sy, onde sx = 0.5 (u_min + u_max) e sy = 0.5 (v_min + v_max) ou ter uma window centrada com os limites iguais em módulo, isto é, u_min = - u_max e v_min = - v_max.

A viewport é considerada de 0,0 no canto esquerdo superior até x_length, y_length no canto direito inferior; isso porque a saída na viewport é considerada nessa faixa (ver em /EZZE-91/ a função "setviewport").

6 - MÓDULO LIGHT.H:

"Em síntese de imagens, após o modelamento dos objetos requeridos em uma cena, parte-se para a análise da iluminação da cena, o que tem importância fundamental para a obtenção do realismo das imagens." /PRET-91/

O módulo light.h contém todas as funções e estruturas relacionadas com a simulação da iluminação.

No TOOKIMA existem dois tipos de fontes de luz (se classificadas com relação à sua área): fontes pontuais e fontes extensas. Fontes de luz *pontuais* 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.

As fontes de luz do TOOKIMA são pontuais:

- "lamp" (lâmpada), é uma fonte pontual 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 pontual 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 renderizador SIPP /YNGV-94/, em particular, permite a geração de sombras, desde que a fonte de luz seja do tipo “spot”.

Além destas possíveis fontes de luz, o modelo de iluminação permite a determinação, pelo animador, da cor de fundo e da cor da iluminação ambiente.

O fenômeno de reflexão apresenta 3 componentes: ambiente, especular e difusa. /GOME-90/. A componente ambiente incide uniformemente de todas as fontes existentes e é refletida igualmente em todas as direções pela superfície. A componente difusa vem de uma fonte e é espalhada igualmente em todas as direções na superfície do objeto (esta componente provém principalmente de superfícies rugosas ou granuladas). A componente especular (parte da energia luminosa que retorna ao meio de propagação quando ela atinge uma interface entre duas superfícies distintas) simula a reflexão da luz de uma superfície, representando os "highlights" (brilhos ao redor dos pontos de impacto dos raios incidentes); este efeito é mais evidente em superfícies polidas do que em rugosas.

Os campos das estruturas que definem as fontes de luz, assim como os campos da estrutura câmera, podem ser tratados como atores. Por exemplo: a posição de uma fonte pontual pode ser um ator do tipo POINT, a direção de iluminação de um spot pode ser um ator do tipo VECTOR, etc.

6.1 - Estruturas do light.h:

LightType - enumeração que especifica os tipos de fonte de luz possíveis:

```
typedef enum { LAMP, SUN, SPOT } LightType;
```

LightPower - enumeração que especifica a potência da lâmpada utilizada (é usada na função "set_lamp", do arquivo l_define.c):

```
typedef enum { W40, W60, W100, W150 } LightPower;
```

SIPP_ShadeType - enumeração que especifica os modelos de shading (tonalização) usados. Shading é "o tratamento da distribuição de luz sobre uma superfície iluminada e proporciona a suavização da tonalidade das cores a serem mostradas em superfícies curvas, representadas de maneira facetada." /PRET-91/. O TOOKIMA, através do SIPP, pode utilizar 3 modelos de Shading:

FLAT Shading (FLAT_SIPP)- representa uma superfície curva como um conjunto de polígonos planares e aplica o modelo a cada plano. Acelera os cálculos, entretanto fornece bons resultados apenas para objetos onde se deseja evidenciar superfícies planas (um cubo, por exemplo); não é adequado para objetos com superfícies esféricas.

GOURAUD Shading (GOUR_SIPP) - "interpola as intensidades das cores das faces, eliminando a aparência facetada da FLAT." /PRET-91/. É um modelo que representa deficientemente a componente especular da reflexão, daí sua utilização ser mais apropriada para objetos prioritariamente difusores. /GOUR-71/

PHONG Shading (PHONG_SIPP)- "interpola inicialmente os vetores normais nos pontos da borda, ao longo de cada Scanline, então calcula a intensidade a cada ponto, usando o vetor normal interpolado." /PRET-91/. É de processamento mais lento que o modelo de Gouraud, por isso deve ser usado com objetos prioritariamente reflexivos (nos quais o modelo de Gouraud não é eficiente). /PHON-75/

Além desses, uma variável do tipo SIPP_ShadeType pode assumir os valores LINE e LINE_ENVELOPE, indicando rendereização em *wireframe* e *wireframe* do envoltório convexo 3D, respectivamente.

Da listagem do programa:

```
typedef enum { FLAT_SIPP, GOUR_SIPP, PHONG_SIPP, LINE, LINE_ENVELOPE }
                                                                ShadeType;
```

ShadeType - Estrutura equivalente à anterior, mas usada no antigo renderizador, que só permite tonalização FLAT, GOURAUD ou PHONG.

Da listagem do programa:

```
typedef enum { FLAT, GOUR, PHONG } ShadeType;
```

SIPP_TextureType - Define os possíveis tipos de material para SIPP.

```
typedef enum { BASIC, PHONGS, STRAUSS, MARBLE, GRANITE, WOOD }
                                                        SIPP_TextureType;
```

SIPP_SpotType - O SIPP permite a criação de 2 tipos de spots. O SOFT, que tem atenuação da luz e o SHARP, que ilumina apenas a região do seu cone de luz, terminando abruptamente.

```
typedef enum { SOFT, SHARP } SIPP_SpotType;
```

PointLight - define uma fonte pontual (lâmpada) e possui apenas um campo "p", que é o ponto onde se localiza a fonte:

```
typedef struct { Point p; } PointLight;
```

SunLight - define a fonte do tipo sol ("sun"). Possui apenas um campo "d", que determina a direção de atuação da fonte (direção dos raios paralelos emitidos pela fonte):

```
typedef struct { Point d; } SunLight;
```

SpotLight - estrutura que armazena as informações usadas no cálculo da intensidade dessa fonte ("spot"). O campo p (Point) determina a posição da fonte no espaço. O campo sa - solid angle (double) apresenta um valor relacionado com a abertura do ângulo de espalhamento (ângulo sólido) da fonte. O campo d (Point) determina a direção de atuação da fonte e o campo type determina se ela será atenuada ou não (SOFT ou SHARP). Da listagem do arquivo:

```
typedef struct {
                    Point p;
                    double sa;
                    Point d;
                    SIPP_SpotType type;
                } SpotLight;
```

LightValue - união que contém as estruturas dos três tipos de fonte de luz:

```
typedef union {
                    PointLight lamp;
                    SunLight dist;
                    SpotLight spot;
                }           LightValue;
```

Light - estrutura com informações gerais relacionadas com as fontes de luz. O campo activity (Boolean) serve para verificar se a fonte está "acesa" (TRUE) ou não está em uso (FALSE). O campo t (LightType) determina o tipo da fonte. O campo v ("value", do tipo LightValue) contém, para cada fonte, os valores que efetivamente descrevem a fonte em questão. O campo c (Color) determina a cor da fonte. Da listagem do programa:

```
typedef struct {
                    Boolean activity;
                    LightType t;
                    LightValue *v;
                    Color c;
                } Light;
```

Ilumination - principal estrutura do módulo e engloba todas as estruturas já definidas neste módulo. Especifica a iluminação da cena. Os campos background e ambient (Color) especificam, respectivamente, a cor de fundo e a cor da luz ambiente. O campo air_index (double) especifica o índice de refração do ar. O campo n_lights (int) conta o número de fontes disponíveis na cena. O último campo é um vetor de apontadores para estruturas do tipo Light, contendo as informações sobre cada fonte de luz da cena. Da listagem do programa:

```
typedef struct {
                    Color background, ambient;
                    double air_index;
                    int n_lights;
                    Light *lights[ N_MAX_LIGHTS ];
                } Ilumination;
```

SIPP_Surface - contém informações sobre as características de cor/textura de um ator ou decoração, para serem usadas pelo SIPP:

```
typedef struct {
                    Color  color,          /* [0,1]  para cada componente*/
                    opacity;               /* [0,1]: 0, transparente; 1, opaco */
                    double ambient,        /* [0,1] */
```

```
    specular,    /* [0,1] */
     c3;         /* [0,1]: 0, brilhante; 1, não-brilhante */
    double diffuse;      /* [0,1]: usado para superfícies  PHONGS */
    int spec_exp;        /* [1,200]: 1, sem brilho; usado para PHONGS */
    Color strip;         /* cor a ser misturada com a cor principal, em
                            MARBLE, GRANITE ou WOOD */
} SIPP_Surface;
```

Surface - contém informações sobre as características de cor de um ator ou decoração, para serem usadas pelo antigo Scanline:

```
typedef struct {
    Color   ka,      /* coef. de cor de luz ambiente */
            kd,      /* coef. de cor de luz difusa */
            ks,      /* coef. de cor de luz especular */
            kt,      /* coef. de transparência */
            ps;
    double  n,       /* controla a concentração do brilho, varia
                        entre 0 e 200, normalmente */
            i_refr;  /* índice de refração */
} Surface;
```

6.2 - Macros do light.h:

N_MAX_LIGHTS - definida como sendo 20.

6.3 - Funções do light.h:

6.3.1: Arquivo l_define.c:

Arquivo com as funções que definem ("set_") e redefinem ("reset_") as fontes de luz. Contém também as funções "read_color", que lêem a cor (descrita num arquivo) da superfície de um ator ou decoração.

SIPP_read_color (String) - função que não vai ser usada diretamente pelo animador, porque é chamada internamente pelas funções "sipp_paint_" do módulo actor.h. Entretanto ela é muito importante porque é responsável pela leitura da cor definida em um arquivo cujo "path" é o parâmetro da função. A função retorna um valor do tipo SIPP_Surface. 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.

read_color (String) - Semelhante à função anterior, usada no antigo Scanline; retorna um valor do tipo Surface. O arquivo texto que define uma cor deve ter a seguinte forma:

ka_r ka_g ka_b
kd_r kd_g kd_b
ks_r ks_g ks_b
n
kt_r kt_g kt_b
ps_r ps_g ps_b
i_refr

Onde:

ka é o coeficiente de cor de luz ambiente, do objeto;
kd é o coeficiente de cor de luz difusa, do objeto;

ks é o coeficiente de cor de luz especular, do objeto;

(os sub-índices *r, g* e *b* representam cada uma das componentes primárias de cor - *red, green* e *blue* -, respectivamente)

n é o parâmetro que controla a concentração do brilho (varia de 0 a 200 e, quanto mais alto, mais concentrado é o brilho);

kt é o coeficiente de transparência do objeto (varia de 0 - totalmente transparente - a 1 - totalmente opaco);

i_refr é o índice de refração da superfície.

Embora não seja uma restrição forte, é aconselhável fazer: *kd + ks = 1*, pois quanto mais difusa for uma superfície, menos especular ela será /ROGE-85/. Para criar efeitos interessantes, pode-se instanciar outros valores.

set_back (double, double, double) - função que define a cor de fundo da imagem gerada. Os três parâmetros são, respectivamente as componentes r, g e b da cor desejada (lembrando que os valores r-g-b variam de 0. a 60000., pela convenção do ProSIm). Esta função pode ser usada tanto no script quanto na cena (quando se quiser alterar a cor de fundo a partir de certo instante da cena).

set_env_light (double, double, double, double) - define a cor ambiente. Os três primeiros parâmetros são os valores das componentes R, G e B da cor. O último é o índice de refração do meio. Esta função pode ser usada tanto no script como na cena. Veja os exemplos:

Ex.1: Uso da função no script:

```
(...)
BEGIN_SCRIPT (1)
(...)
set_env_light ( 60000.,0.,0., 1.0 );
                                /* A luz ambiente é vermelha (R=60000,
                                G=B=0 e o índice de refração do meio é 1 */
```

Ex.2: Uso da função na cena:

```
(...)
void cena1 ( cue )
Cue *cue;
{
(...)
At ( 2.0 )
{
set_env_light ( 0.,60000.,0., 1.5 );
                                /* No instante 2 seg., a cor ambiente passa a
                                 ser verde (G=60000, R=B=0) e o meio
                                tem índice de refr. 1.5 */
}
```

set_lamp (LightPower, double, double, double) - define uma lâmpada. O primeiro parâmetro determina a intensidade da mesma (W40, W60, W100 ou W150). Na verdade cada valor representa uma cor para a luz gerada pela lâmpada (W40 gera uma luz de cor escura - r = g = b = 16000 - e W150 gera luz branca; as outras têm valores intermediários). Os três parâmetros reais representam, respectivamente, as coordenadas x, y e z da posição da lâmpada. É usada no script, e não na cena.

set_point_light (double, double, double, double, double, double) - define uma fonte pontual, cuja cor é dada pelos três primeiros parâmetros e a posição é dada pelos três últimos parâmetros. É semelhante à função anterior, com a diferença de que a cor da luz gerada pode ser diferente das cores padrões para lâmpadas. Veja os exemplos:

Ex.1:
```
(...)
BEGIN_SCRIPT (2)        /* Usada no script */
(...)
set_point_light ( 50000., 40000., 0., 13., 5., 4. );
                        /* Fonte está no ponto ( 13, 5, 4 ) e tem  cor
                        com os componentes RGB = (50000,40000,0) */
```

Ex. 2:
```
(...)
BEGIN_SCRIPT (3)
(...)
set_point_light ( 16000., 16000., 16000., 5., 5., 9. );
                        /* Fonte no ponto ( 5, 5, 9 ) e de cor definida
                          por (16000,16000,16000). Como é a cor
                          definida por W40, seria equivalente usar:
                          set_lamp ( W40, 5.,5., 9. ); */
```

set_sun (double, double, double, double, double, double) - define fonte do tipo sun. Os três primeiros parâmetros novamente definem a cor da luz gerada e os três parâmetros seguintes determinam a direção dos raios pararelos da fonte (respectivamente, as coordenadas x, y e z do vetor que indica a direção). Esta função deve ser usada no script.

set_spot (double, double, double, double, double, double, double, double, double, SIPP_SpotType) - define fonte do tipo spot. Os três primeiros parâmetros definem a cor da luz gerada. Os três parâmetros seguintes determinam o ponto onde está localizada a fonte. Os três parâmetros seguintes são as coordenadas do vetor que indica a direção de atuação da fonte. O último parâmetro indica se o spot é SOFT ou SHARP. Por exemplo:

```
(...)
BEGIN_SCRIPT (2)                /* Deve estar no script */
(...)
set_spot ( 60000., 0., 0., 0., 5., 0., 0., -1., 0., SHARP);
                        /* spot de luz vermelha, no ponto ( 0, 5, 0 )
                        e iluminando na direção ( 0, -1, 0 ) */
set_spot ( 0., 0., 60000., 0., -3., 1.5, 0., 1. , 0., SOFT );
                        /* spot de luz azul, no ponto ( 0, -3, 1.5 ) e
                        iluminando na direção ( 0, 1, 0 ) */
```

reset_light_p (int, double, double, double) - função que determina uma nova posição para uma fonte de luz (se for fonte do tipo sun, não acontece nada, já que esta não tem posição). O primeiro parâmetro é o número da fonte (as fontes vão sendo numeradas à medida que vão aparecendo no script, começando em 0. Os três parâmetros seguintes dão, respectivamente, as coordenadas x, y e z da nova posição. Esta função deve ser usada na cena, e não no script.

reset_light_d (int, double, double, double) - determina nova direção de atuação para uma fonte de luz (se for do tipo lamp, nada acontece, pois esta não tem direção de atuação). O primeiro parâmetro é o número da fonte. Os três parâmetros seguintes dão, respectivamente,

as coordenadas x, y e z do vetor que indica a nova posição. Esta função deve ser usada na cena, e não no script.

reset_light_c (int, double, double, double) - determina uma nova cor para a luz gerada por uma fonte. O primeiro parâmetro é o número da fonte. Os três parâmetros seguintes dão, respectivamente, as componentes r, g e b do vetor que indica a nova direção. Esta função também deve ser usada na cena, e não no script.

6.3.2: Arquivo l_transf.c:

Arquivo com as funções de "fade" e também as que ligam e desligam as fontes de luz.

turn_on (int) - função que "liga" uma fonte de luz. Quando a fonte é definida com uma das funções "set_", o campo activity da estrutura Light é colocado em FALSE (ou seja, a fonte está "desligada"). Para que se possa observar o efeito das fontes, é necessário que se use esta função, que coloca TRUE naquele campo. O parâmetro da função é o número da fonte a ser ligada. Pode ser usada tanto na cena quanto no script.

turn_off (int) - faz o contrário de "turn_on", isto é, "desliga" uma fonte, colocando FALSE no campo activity da estrutura Light. O parâmetro é o número da fonte a ser desligada. Pode ser usada tanto na cena quanto no script (embora aí não faça sentido, pois as funções "set_" já deixam as fontes desligadas).

fade_in (int, double, double, Watch) - realiza o "fade-in", isto é, a luz vai acendendo-se, partindo da escuridão até sua cor determinada (dada pelas funções "set_..."). O primeiro parâmetro é o número da fonte a realizar o "fade-in" (se for -1, ocorrerá "fade-in" com a luz ambiente e se for -2, com a cor de fundo). Os dois parâmetros seguintes determinam o intervalo do "fade-in" (o primeiro marca o instante inicial, e o outro marca o instante final do "fade-out"). O último parâmetro é o relógio da cena (deve ser definido apenas como "clock"). Ex.:

```
fade_in ( -1, 0., 10., clock);     /* Realiza o fade-in da luz ambiente de 0 a 10s. */
```

fade_out (int, double, double, Watch) - semelhante à anterior, porém realiza o "fade-out", isto é, a luz vai apagando-se, a partir da sua cor original. O primeiro parâmetro é o número da fonte a realizar o "fade-out" (se for -1, ocorrerá "fade-out" com a luz ambiente e se for -2, com a cor de fundo). Os dois parâmetros seguintes determinam o intervalo do "fade-out". O último parâmetro é novamente o relógio da cena. Tanto "fade-in" quanto "fade-out"devem ser usadas na cena, e não no script. Veja o exemplo:

```
void cena1(cue)
Cue *cue;
{
(...)
fade_in ( -1, 0., 10., clock );          /* de 0 a 10s a luz ambiente sofre fade-in,
                                            até chegar à sua cor original, definida por
                                            "set_back", no script. */
```

```
fade_out ( -2, 30., 50., clock );      /* de 30 a 50s a cor de fundo vai
                                          escurecendo, a partir do seu valor inicial,
                                          dado por  "set_env_light", no script. */
(...)
```

fade_in_all (double, double, Watch) - todas as luzes vão sofrer o "fade-in" (daí a razão de não ter o primeiro parâmetro associado ao número de uma fonte).

fade_out_all (double, double, Watch) - idêntica a "fade_in_all", só que todas as luzes sofrerão "fade-out".

light_follow_actor (int, String) - faz com que uma fonte de luz passe a "seguir" um ator, isto é, ela terá sua posição no centro de gravidade do ator (que será invisível). O primeiro parâmetro indica o número da fonte de luz e o segundo, o nome do ator. Exemplo:

```
light_follow_actor (2, "A3");
```

No exemplo acima, a fonte de luz número 2 terá sua posição determinada pala posição do ator A3.

spot_aim_actor (int, String) - faz com que um spot esteja sempre direcionado para um ator. É como se fosse um foco iluminando sempre o ator. O primeiro parâmetro é o número da fonte de luz (só funciona se for do tipo "spot") e o segundo é o nome do ator a ser iluminado.

sun_direct_actor (int, String, String) - faz com que a direção de uma fonte do tipo "sun" (cujo número é o primeiro parâmetro da função) seja sempre dada pela diferença entre os centros de gravidade de 2 atores (cujos nomes são os dois últimos parâmetros).

7 - EXEMPLO COMPLETO:

A seguir, será dada uma animação completa:

```
                                /**** Início da animação *****/

#include <stdio.h>
#include "p_transf.h"
#include "p_vector.h"
#include "p_alloc.h"
#include "p_error.h"
#include "p_graph.h"            /* No início de cada animação, deve-se incluir todos os */
#include "p_poly3d.h"           /*arquivos.h necessários. */
#include "camera.h"
#include "actor.h"
#include "time.h"
#include "scanline.h"
#include "sl_wire.h"
#include "light.h"
#include "script.h"
#include "motion.h"

extern Ilumination lumen;       /* Quando se usa fontes luminosas, deve-se fazer esta
                                   declaração no início. */

void cena1 (cue )               /* Procedimento "cena1"; vai ser chamado no script */
Cue *cue;                       /* e está associado a uma cue. */
{
Point po;
BEGIN_SCENE
NO_CLOCK;

part ( "luz" );
part ( "esfera" );
part ( "cubo" );                /* Os atores  luz, esfera e cubo participam da cena. */

fade_in ( -2, FRAME(0), FRAME(5), clock );
                                /* Até o quinto quadro ocorrerá o fade-in da luz de
                                fundo até o azul (sua cor original). */
fade_out ( -1, FRAME(12), FRAME(15), clock );
                                /* Do quadro 12 ao 15 ocorrerá o fade-out da luz
                                ambiente. */

At ( 0. )                       /* Inicializações. */
{
 translate_actor ( ABSOLUTE, 0., 2., 0., "cubo" );
 translate_actor ( ABSOLUTE, 0., -2., 0., "esfera" );
 translate_actor_z ( ABSOLUTE, -7., "luz" );
 rotate_actor_y ( RELATIVE, 30., "cubo" );
}

At ( FRAME (6) )
{
 turn_on (0);                   /* Acende-se a luz 0. */
 turn_on (1);                   /* Acende-se a luz 1. */
```

```
 }

After ( FRAME (6) )
{
 rotate_actor_y ( RELATIVE, 10., "cubo" );
                                    /* O cubo girará 10 graus no sentido y a cada quadro. */
}

At ( FRAME ( 10 ) )
{
 turn_off ( 1);                     /* Apaga-se a luz 1. */
 }

At ( FRAME (11) )
{
 rotate_actor_x ( ABSOLUTE, 90., "luz" );
 reset_light_d ( 0, 0., -1., 0. );  /* Roda-se a posição do spot - ator "luz" - de 90 graus e
                                       muda-se sua direção de atuação, para a direção -y. */
}

END_SCENE
}                                   /* Fim da cena. */

BEGIN_SCRIPT (1)

SET_RGB ( cor2, 0., 0., 60000. );
SET_RGB ( cor3, 60000., 60000., 60000. );

set_studio ( NULL, 2, "../images/scan/anim" );
                                    /*As imagens geradas se chamarão anim e estarão no
                                     subdiretório ../images/scan. */
set_obs ( 5., 0., 0. );             /* O observador é colocado em ( 5, 0, 0 ). */

set_back ( 0., 0., 60000. );        /* Cor de fundo é azul. */
set_env_light (60000., 60000., 60000.,1.);
                                    /* Luz ambiente é branca. */
set_spot ( 60000., 0., 0., 30., 30., 30., 0., 0., 1., SOFT );
                                    /* A fonte 0 é um spot vermelho no ponto (30, 30, 30)
                                     e com direção z  ( 0, 0, 1 ). */
set_lamp ( W100, 0., 2., 5. );      /* A fonte 1 é lâmpada de 100W no ponto ( 0, 2, 5 ). */

TotalTime = 3;
Rate = 5;                           /*A animação terá 15 quadros ( 3 segundos numa taxa
                                    de 5 quadros/s ). */

cast_poly ( "cubo", “cube.brp” );
cast_poly ( "esfera", “/home/brp/esf.brp” );  /*Definição dos atores cubo1 e cubo2 como  poliedros. */
cast_point ( "luz", &lumen.lights[ 0]->v->spot.p );
                                    /* O ator luz é a posição do spot  (fonte 0). */
sipp_render(PHONG_SIPP, 1, TRUE);
sipp_paint_actor ( "cubo", BASIC, “amarelo.cor” );
sipp_paint_actor ( "esfera", BASIC, “/home/cores/lilas.cor” );
                                    /* O cubo é amarelo e a esfera é lilás. */

SET_CUE ( cue[ 0], FRAME ( 0 ), FRAME ( 15 ) );
                                    /* cue[ 0] é definida começando em 0 e terminando no
```

```
                                        final da animação - FRAME ( 15 ). */

LIST_SCENES
        cena1 ( cue[ 0] );          /* cena1 vai ocorrer no intervalo definido por cue[ 0]. */
END_LIST

END_SCRIPT
                                /***** Fim da animação  *****/
```

No início da animação ocorre um fade_in da luz de fundo, que chega até o azul no frame 5. Os atores esfera e cubo estão no eixo y - um em 2 e outro em -2. No frame 6, a lâmpada e o spot vermelho são ligados (o spot está no eixo -z iluminando no sentido z, pegando os atores de lado). A partir do frame 6, o cubo começa a girar. No frame 10, a lâmpada é apagada. No frame 11 o ator luz gira 90 graus em relação ao eixo x (como o ator luz é a posição do spot, este passa a ficar acima dos atores). Neste mesmo frame, a direção de iluminação do spot também é mudada para -y, fazendo com que a luz pegue agora os atores de cima. A partir do frame 12, ocorre um fade-out da luz ambiente.

APÊNDICE A:

ProSIm:
"Projeto em Síntese de Imagens Foto-Realistas e Animação"

No início deste trabalho, foi explicado o que é o ProSIm e onde o TOOKIMA se inclui neste projeto. Entretanto, para a criação de uma animação com o TOOKIMA, é preciso aprofundar um pouco mais o conhecimento sobre o ProSIm. Isso porque muitas vezes se faz necessário o uso de funções do ProSIm. É por esta razão que este apêndice apresenta algumas funções e estruturas do ProSIm que são comumente usadas em uma animação com o TOOKIMA.

As estruturas e funções do ProSIm estão separadas em arquivos .h e .c, de acordo com sua funcionalidade, como no TOOKIMA. Porém, como neste apêndice serão vistas apenas algumas funções e estruturas, não as separaremos de acordo com o arquivo em que elas se encontram.

A.1 - Estruturas do ProSIm:

Point - estrutura que define um ponto, com suas três coordenadas reais:

```
typedef struct { double x, y, z } Point;
```

Vector - estrutura que define um vetor, com suas três projeções ortogonais:

```
typedef struct { double x, y, z } Vector;
```

Matrix - estrutura que define uma matriz 4x4 (usada, por exemplo, em algumas funões do actor.h):

```
typedef double Matrix[ 4 ][ 4 ];
```

Color - define uma cor, através de suas três componentes: r (red), g (green) e b (blue). Os valores destas coordenadas variam de 0 a 60000 (intensidade máxima da componente).

```
typedef struct { double r, g, b } Color;
```

Face - define a face de um poliedro.

```
typedef struct
        {       int n_vert;
                Index *v_list;
        }       Face;
```

O primeiro campo determina o número de vértices da face e o segundo campo é a lista dos vértices (o tipo Index é simplesmente um inteiro).

Polyhedron - estrutura que complementa às definidas em actor.h, relacionadas com o definição de um ator. Contém os elementos topológicos e geométricos de um modelo B-rep simplificado.

```
typedef struct
        {       int n_points;
                int n_faces;
                Point *p_list;
                Face *f_list;
        }       Polyhedron;
```

Os dois campos de inteiros determinam o número de pontos e o número de faces do poliedro; o campo p_list é a lista de vértices e o campo f_list é a lista de faces que compõem o poliedro.

A.2 - Macros do ProSIm:

SET_POINT (P, X, Y, Z) - define o ponto P, com as coordenadas X, Y e Z:

```
#define SET_POINT (P, X, Y, Z)  ( (P).x = (X), (P).y = (Y), (P).z = (Z) )
```

SET_VECTOR (V, X, Y, Z) - define o vetor V, com as projeções X, Y e Z:

```
#define SET_VECTOR (V, X, Y, Z)  ((V).x = (X), (V).y = (Y), (V).z = (Z))
```

SET_RGB (C, R, G, B) - define a cor C, com as componentes R, G e B:

```
#define SET_RGB (C, R, G, B)  ( (C).r = (R), (C).g = (G), (C).b = (B) )
```

Debug - macro sem parâmetros que deve ser usada no início do script de uma animação quando se deseja que haja o "debugging" (este também pode ser feito com o dbxtool do ambiente Sun-Unix). Esta macro faz com que a variável externa debug_level seja 1. Para um animador um pouco mais experiente, o "debugging" da animação pode ser importante na detecção de falhas na mesma.

dprintf - equivalente à função "printf" da linguagem C, entretanto só vai ser realizada se debug_level = 1. Grande parte dos comentários e avisos das funções do TOOKIMA usam dprintf. É por isso que, sem a utilização da macro Debug (isto é, com debug_level = 0), não haverá o "debugging" da animação (quase nada será escrito durante a execução da mesma).

Dot (A, B) - muito útil no modelo da câmera; definida como:

```
( (A).x*(B).x + (A).y*(B).y + (A).z*(B).z )
```

Mul (P, A, k) - coloca no ponto P o valor da multiplicação de todas as coordenadas do ponto A por k:

```
(  (P).x = (A).x * (k);  (P).y = (A).y * (k);  (P).z = (A).z * (k)  )
```

Vadd (P, A, B) - coloca no ponto P o valor da soma das cordenadas dos pontos A e B:

```
(  (P).x = (A).x + (B).x;  (P).y = (A).y + (B).y;  (P).z = (A).z + (B).z  )
```

Vsub (P, A, B) - coloca no ponto P o valor da diferença das cordenadas dos pontos A e B:

```
(  (P).x = (A).x - (B).x;  (P).y = (A).y - (B).y;  (P).z = (A).z - (B).z  )
```

A.3 - Funções do ProSIm:

p_malloc ( ) - alocação dinâmica de memória, "acompanhada" pelo ambiente ProSIm (semelhante à função "malloc" da linguagem C).

p_calloc ( ) - outro tipo de alocação dinâmica de memória "acompanhada" pelo ambiente ProSIm (semelhante à função "calloc" da linguagem C).

p_free_all ( ) - função utilizada internamente à macro END_SCRIPT. Desaloca todos os ponteiros alocados, usando um contador que indica o número de ponteiros alocados e fazendo um looping com a função "free", da linguacem C.

p_error (int number, char *msg) - imprime a mensagem de erro (a String msg), desaloca todos os ponteiros (usando p_free_all) e retorna o número do erro (number). Por exemplo, na função "translate_actor" do módulo actor.h, aparecerá a seguinte mensagem de erro se o ator que se deseja mover não participar da cena:

```
p_error ( -1, "\nA_T: Sorry, can't translate unknown actor" );
```

p_init_graph( ) - função que inicializa o sistema gráfico. É chamada internamente à função "set_studio" do módulo script.h, mas no ambiente Unix não efetua nada de útil.

APÊNDICE B:

Scanline:

O sistema Scanline é destinado à síntese de imagens /PRET-91/, /PRET-93/.

A parte de visualização da cena é feita através dos sistemas para "rendering". O sistema Scanline é uma das opções existentes, embora o SIPP tenha sido a opção mais usada atualemente.

Em algumas estruturas do TOOKIMA, aparecem campos que são estruturas definidas no Scanline. Por esta razão, algumas estruturas do Scanline serão vistas neste apêndice.

B.1 - Estruturas do Scanline:

Wireframe - estrutura que aparece como um dos campos da estrutura PolyPlus, do módulo actor.h. Definida como:

```
typedef struct    {
                    int     edge_number;
                    int     vtop;
                    int     vbot;
                    int     poly_dad;
                    int     poly_mom;
                  }     Wireframe;
```

ScanIn - "a única estrutura de dados que qualquer programa integrado ao Scanline deve conhecer é a estrutura ScanIn, que define os parâmetros de entrada para a função scanline"./SILV-92/. Uma estrutura deste tipo é usada como parâmetro da função set_camera, do módulo câmera.h. É assim definida:

```
typedef struct    {
                    int             n_o;
                    Polyhedron      *geo[N_MAX_OBJ];
                    Surface         *viz[N_MAX_OBJ];
                    ShadeType       shade[N_MAX_OBJ];
                  }     ScanIn;
```

BIBLIOGRAFIA:

/BADL-87/ : BADLER, Norman I.
"3D Computer Modeling and Animation with Emphasis on Human Figures".
Course Notes of ZGDV (Zentrum für Graphische Datenverarbeitung) Seminar (based on SIGGRAPH'87 Avanced Computer Animation Course) - 1987.

/CAMA-92/ : CAMARGO, J. Tarcísio Franco de
"Métodos de Controle do Movimento de Objetos Rígidos Articulados".
Relatório de estudos especiais
Julho - 1992.

/CAMA-95/: CAMARGO, J. Tarcísio Franco de
Tese de Doutorado: "Animação Modelada por Computador: Técnicas de Controle de Movimento em Animação".
DCA - FEE - UNICAMP - Janeiro 1995.

/EZZE-91/ : EZZELL, Ben
"Programação Básica em Turbo C++".
Ed. Ciência Moderna - 1991.

/FOLE-90/: FOLEY, James D. et al.
"Computer Graphics: Principles and Practice".
2nd. edition - Addison Wesley Publ. Co. - 1990.

/GOME-90/ : GOMES, J. Miranda e VELHO, L. C.
"Conceitos Básicos de Computação Gráfica".
VII Escola de Computação, São Paulo - 1990.

/GONG-94/: GONG, K. L.
“Berkeley MPEG-1 Video Encoder - User’s Guide”
Computer Science Division, University of California, Berkeley, CA.

/GOUR-71/ : GOURAUD, H.
"Computer Display of Curved Surfaces".
IEEE Transaction C-20, pp. 623-628 - 1971.

/HAYT-83/ : HAYT Jr., William H.
"Eletromagnetismo" - 3a. edição.
Livros Técnicos e Científicos Editora - 1983

/KOCH-87/: KOCHAN, Stephen G. e WOOD, Patrick H.
"Exploring the UNIX System".
Hayden Books - 1987.

/ISAA-87/ : ISAACS, P. M. et alli.
"Controlling Dynamic Simulation with Kinematic Constrains, Behavior Functions and Inverse Dynamics".
Computer Graphics, v.21, n.4, pp. 215-224 - 1987.

/MADE-92/ : MADEIRA, Heraldo França.
"Relatório ProSIm - Geomod - Modelador Geométrico -

Especificação e Implementação".
Relatório Interno DCA - FEE - UNICAMP.
(em elaboração)

/MAGA-91/ : MAGALHÃES, Léo P. e da SILVA, Marcelo 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

/MORT-85/ : MORTENSON, Michael E.
"Geometric Modeling"
John Wiley & Sons. New York - 1985.

/PHON-75/ : PHONG, Bui-Tuong
"Ilumination for Computer Generated Images".
Communication of The ACM vol.18, #6, pp. 311-317 - 1975.

/PRET-91/ : PRETO, Tania M. e MAGALHÃES, Léo P.
"Scanline - ProSIm"
Relatório Interno DCA - 028/91 - FEE - UNICAMP.

/PRET-93/ : PRETO, Tania M.
Tese de Mestrado: "SCANLINE - Um Sistema para Visualização de Imagens Foto-Realistas".
DCA - FEE - UNICAMP - 1993.

/PUEY-88/ : PUEYO, Xavier e TOST, Daniela
"A Survey of Computer Animation".
Computer Graphics, Forum 7, pp. 281-300 - 1988.

/RAPO-95/: RAPOSO, Alberto B.
"Projeto, Especificação e Implementação de um Sistema Interativo de Animação no Contexto ProSIm".
Relatório Técnico - FAPESP - 1995.

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

/RAPO-97/: RAPOSO, Alberto B. e MAGALHÃES, Léo P.
"Projeto, Especificação e Implementação de um Sistema Interativo de Animação no Contexto ProSIm".
Relatório Interno DCA - 003/97 - FEEC - UNICAMP.

/RODR-92/ : RODRIGUES, Maria Andréia Formico
"Mané Mosquito e Corujito".
Relatório Interno DCA - 013/92 - FEE - UNICAMP

/RODR-93/ : RODRIGUES, Maria Andréia Formico
Tese de Mestrado: "Animado"
DCA - FEE - UNICAMP - 1993

/ROGE-85/ : ROGERS, David F.
"Procedural Elements for Computer Graphics".
McGraw-Hill Book Co., Singapore - 1985.

/SANT-89/ : SANTO, H. P.
"Computação Gráfica, Compugrafia ou Compugráfica?"
SIBGRAPI'89, Águas de Lindóia - Abril - 1989.

/SILV-92/ : da SILVA, Marcelo H.
Tese de Mestrado: "TOOKIMA: Uma ferramenta para Animação Modelada por Computador".
DCA - FEE - UNICAMP - Abril - 1992.

/YNGV-94/: YNGVESSON, J. e WALLIN, I.
“User’s Guide to SIPP - a 3D Rendering Library - Version 3.1”
1994.