Projeto Final - O cachorro e o gato

Joás Gonçalves Sanches[1], **Lucas Santana Lellis**[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
Rua Talim, 330, São José dos Campos, São Paulo, CEP 12231-280

joas.sanches@unifesp.br, lellis@unifesp.br

1. Introdução

A animação de personagens em um ambiente 3D envolve vários conceitos de computação gráfica, geometria analítica, estrutura de dados. Tais conhecimentos auxiliam no desenvolvimento de aplicações diversas, podendo ser feitas com a ajuda de bilbiotecas, como o OpenGL.

Os conceitos imprescindíveis à animação de personagens são os de geometria analítica em ambito geral, como o conhecimento de vetores e sistemas de coordenadas, assim como os conceitos atrelados às transformações espaciais, como a translação, rotação, e também a escala, que são feitas através de marizes de transformação, que multiplicam coordenadas homogêneas, ou através de funções mais simples.

O OpenGL, possibilita a modelagem de polígonos, retas, e algumas primitivas básicas, abstraindo as funções de preenchimento de polígonos e retas, assim como funções mais avançadas de recorte de polígonos e retas, simulação de câmeras e iluminação, projeções espaciais e muitos outros recursos. Ainda assim, em alguns casos precisamos implementar nossas próprias funções para fins específicos, já que a biblioteca suporta a transformação de coordenadas se não for na area de desenho.

Algumas funcionalidades podem facilitar a modelagem de objetos, como as sweeps, splines, fractais, entre outros. Esses tipos de operações automatizam operações, exigindo menos esforço do programador em calcular as coordenadas manualmente, e permitindo modelos mais realistas, conforme os seus propósitos.

2. Descrição do Projeto

Este projeto tem o objetivo de simular a interação entre um gato e um cachorro, de forma que, ao inserir o gato na cena, o cachorro começa a persegui-lo e eventualmente consegue pega-lo.

Particularmente, pretende-se apresentar uma implementação de sweeps translacionais, utilizada para a modelagem dos personagens e do cenario, além de demonstrar a utilização de técnicas conhecidas da computação gráfica, como projeções paralelas e perspectivas e a interação dos dispositivos de entrada com o ambiente tridimensional.

3. Metodologia

O desenvolvimento do projeto foi feito em quatro etapas, sendo estas:

- **Estrutura básica:** Implementação das funções básicas de visualização, e primeiros modelos dos personagens.
- **Técnicas de modelagem:** Implementação de técnica de sweep.
- **Reestruturação do sistema de câmeras e Animação:** O programa foi reformulado e espalhado em diversas estruturas, facilitando a junção dos dois personagens na mesma cena, e posterior, animação em conjunto.

- **Aprimoramento da interface de usuário:** Adição de menu e diversos recursos na interface.
- **Textura e Iluminação:** Implementação de aspectos visuais como texturas e iluminação.
- **Finalização e correções:** Correções de bugs e inserção de recursos restantes.

3.1. Estrutura Básica

No início foi implementada uma estrutura básica para a modelagem dos personagens. Isto é, foi configurado o projeto com uma projeção perspectiva, retas coloridas para definir os eixos cartesianos, e funções básicas de rotação para melhor visualizar o modelo.
Foi implementado um modelo, baseado em TAD, utilizando apenas as primitivas oferecidas pela GLUT, definindo assim, a estrutura de movimentação e animação do personagem, porém com uma modelagem muito pobre e pouco realista. Assim, foi feita uma tentativa de desenhá-los utilizando apenas os vértices, gerando polígonos. Mas a dificuldade era muito grande em calcular manualmente as coordenadas. Assim, foi feita uma pesquisa com relação à outras técnicas de modelagem, e, baseado na facilidade oferecida pela extrusão no Software de computação gráfica Blender, foi escolhida a técnica de sweeps translacionais.

3.2. Técnicas de modelagem

Visto o resultado ruim obtido utilizando apenas primitivas da biblioteca, foi implementada a técnica de sweeps, de acordo com as funcionalidades descritas por [Hearn and Baker 2004], foi definido um TAD chamado Object, que continha uma matriz de coordenadas e o número de vértices e camadas do objeto em questão. Sua estrutura baseia-se em uma matriz de coordenadas do tipo "Pt3D".
As coordenadas do tipo pt3D interagem com funções de transformações espaciais obtidas em [Hearn and Baker 2004]. Tais operações baseiam-se em matrizes de transformação, rotação baseada em quaternions. Foi implementada também a opção de alternar entre desenho em forma de linha e de polígono.
Facilitando a modelagem do cachorro e do gato, em uma estrutura de dados robusta e inteligente para representar as partes integrantes dos personagens. Cada integrante fez a modelagem de seus respectivos personagens, baseados em imagens e vídeos obtidos na web, criando faces em 2D que podiam sofrer extrusões, translações, rotações e escala. A combinação destas operações geraram resultados muito bons, e permitiram uma modelagem menos trabalhosa.
Foram desenhadas também as cercas que restringem, de certa forma a região de locomoção dos personagens.

3.3. Reestruturação do sistema de câmeras e Animação

Foi feita a junção dos dois personagens na mesma cena, sendo implementadas funções de geometria analítica, como produto vetorial e produto escalar, que agem sobre coordenadas do tipo Pt3D. Tais funções foram necessárias para garantir a corretude da perseguição, uma vez que era necessário considerar o ângulo do cachorro com relação ao eixo Z, e também o ângulo entre o seu deslocamento, e a posição do gato. Para ter uma animação personalizada do cachorro, foram criadas as animações de sentar e de levantar.
Para possibilitar recursos adicionais de visualização, e representar diferentes formas

de utilização das ”câmeras” foi implementado um TAD específico para elas, com funcionalidades diferenciadas, baseadas em uma variável do tipo enum, que definia qual tipo de câmera estaria sendo utilizada, sendo implementadas câmeras fixas que seguem os personagens, câmeras móveis que podem ser manipuladas através do mouse, e uma câmera que representa a visão do cachorro durante a sua caça pelo gato.

3.4. Aprimoramento da interface de usuário

Para permitir um controle mais intuitivo das câmeras, e demais funcionalidades, foi criado um menu com opções diversas, podendo ser acessado através do botão do meio do mouse. Além de inserir comandos através de teclas do teclado.
Foi implementada a inversa da projeção perspectiva, funcionando de acordo com a câmera ativa no momento, isso é necessário para que seja possível a interação do mouse com o espaço tridimensional, permitindo a inserção do gato na posição correspondente do mouse na tela. Como requisito do projeto, também foram adicionados funções de transformação que agem diretamente nos personagens a partir de teclas específicas do teclado, como escala, translação e rotação.

3.5. Textura e Iluminação

Para melhor representação do cenário, foram introduzidas texturas no chão e no céu. A iluminação foi feita através de 3 tipos diferentes de luz, sendo estes a ambiente, a spot e um ponto de luz comum. Os spots foram programados para seguir o gato na cena.

4. Implementação

A implementação foi espalhada em diversos TADs, sendo o mais importante, o ”Object”.

4.1. Object

Object é a estrutura utilizada para aplicar a técnica de sweeps. Onde é feito um modelo 3D, que é duplicado e transformado, formando objetos tridimensionais, compostos por diversas camadas no eixo Z. Sua estrutura básica é:

```
struct SObject {
    int currentSlice; //Camada atual da transformação
    int currentVertex; //Número do último vértice desenhado
    int nSlices; //Quantidade máxima de camadas
    int nVertex; //Quantidade máxima de vértices
    struct sPt3D **polygon; //Matriz de coordenadas
    Pt3D sliceCtr; //Guarda o ponto central da camada atual
} TObject;
typedef struct SObject *PObject;
```

A matriz de coordenadas é alocada dinâmicamente, através de uma espécie de construtor, baseados na quantidade máxima de vértices e camadas.
A operação AddVetex itera o vértice atual, e copia os valores recebidos para o novo vérice, ela é utilizada para desenhar o modelo 2D de base.

A operação de ”extrusão” incrementa a camada atual e a preenche com uma cópia da anterior, de forma que as transformações sejam cumulativas. As transformações espaciais são aplicadas em conjunto sobre todas as coordenadas de uma camada, através das funções contidas em 3DTransforms, obtidas em [Hearn and Baker 2004].
O objeto é desenhado utilizando a ferramenta oferecida pelo openGl para desenho de elementos, podendo ser preenhido ou linha. A sua implementação baseia-se na ordem em que são feitas as coordenadas de cada face, e pode ser vista abaixo:

```
glPushMatrix();
int slice, vtx;
glBegin(format);
//Desenha a primeira e última camada, fechando as pontas do
   polígono
for (vtx = 0; vtx < obj->currentVertex; ++vtx) {
    Pt3D pt = &(obj->polygon[0][vtx]);
    glVertex3d(pt->x, pt->y, pt->z);
}
glEnd();
glBegin(format);
for (vtx = 0; vtx < obj->currentVertex; ++vtx) {
    Pt3D pt = &(obj->polygon[obj->currentSlice][vtx]);
    glVertex3d(pt->x, pt->y, pt->z);
}
glEnd();
//Desenha as camadas laterais, que evolvem dois vértices, e duas
    camadas em cada face.
for (vtx = 0; vtx < obj->currentVertex - 1; ++vtx) {
    for (slice = 0; slice < obj->currentSlice; ++slice) {
        glBegin(format);
        //Vértice vtx da camada atual
        Pt3D pt = &(obj->polygon[slice][vtx]);
        glVertex3d(pt->x, pt->y, pt->z);
        //Vértice correspondente na camada seguinte
        pt = &(obj->polygon[slice + 1][vtx]);
        glVertex3d(pt->x, pt->y, pt->z);
        //Vértice vtx+1 na camada seguinte
        pt = &(obj->polygon[slice + 1][vtx + 1]);
        glVertex3d(pt->x, pt->y, pt->z);
        //Vértice correspondente na camada atual
        pt = &(obj->polygon[slice][vtx + 1]);
        glVertex3d(pt->x, pt->y, pt->z);
        glEnd();
    }
}
glPopMatrix();
```

Por fim, existe uma função freeObject, que auxilia a liberação hierárquica de memória.

4.2. Cat

O gato é representado pelo TAD cat, que se baseia neste struct:

```
struct cat {
        //Strcts que representam as pernas
    Leg * leg1, * leg2, * leg3, * leg4;
        //Partes do gato, do tipo Object
    PObject body, neck, head, earLeft, earRight, tail;
    Pt3D position;  //Posição do gato na animação
    Pt3D rotation;  //Rotação do gato ( pelo teclado )
    Pt3D translation; //Translação do gato ( pelo teclado )
    float angle; //Angulo de rotação no eixo y ( animação )
    //Contador de iterações, utilizado para alimentar animação
    float time;
    float stamina; //Energia do gato, ele se cansa depois de um
       tempo
    float speed; //Velocidade do gato
    float scale; //Escala (teclado)
    int enabled; //Booleano, diz se gato está visível
    int stop;
};
```

Para criar o gato, existe a função createCat. Nela são inicializadas todas as variáveis e estruturas. Nela é onde ocorre o desenho do gato, através das operações fornecidas por Object.
O desenho da orelha esquerda, por exemplo, acontece da seguinte maneira:

```
//Inicializa o objeto, dizendo quantos vértices e camadas terá
   no total
cat->earLeft = CreateObject(7, 3);
//Desenha o modelo 2D
AddVertex(cat->earLeft, 0.0, 0.0);
AddVertex(cat->earLeft, -0.1, 0.0);
AddVertex(cat->earLeft, -0.12, 0.14);
AddVertex(cat->earLeft, -0.04, 0.28);
AddVertex(cat->earLeft, 0.1, 0.14);
AddVertex(cat->earLeft, 0.1, 0.0);
AddVertex(cat->earLeft, 0.0, 0.0);
//Transformações espaciais sobre a camada atual, e extrusões
//Criou a camada intermediaria, onde fica a ponta da orelha
Extrude(cat->earLeft);
//Estica a camada, para ficar maior que a anterior
ScaleSlice(cat->earLeft, 1.1, 1.3);
//Translação da camada para frente do fundo da orelha
TranslateSlice(cat->earLeft, 0, 0, 0.09);
//Camada final, forma a parte interna da orelha
Extrude(cat->earLeft);
//Escala encolhe a orelha
ScaleSlice(cat->earLeft, 0.6, 0.6);
//Translação da camada para dentro da orelha
TranslateSlice(cat->earLeft, 0.01, 0, -0.05);
//Aplica as trsnaformações da última camada
FinalizeObject(cat->earLeft);
```

Para desenhar o gato na tela, foi criado a função drawCat, contendo como parametros de entrada o objeto gato e o estado da camera. Nela todas as partes criadas do gato são transladadas do seu ponto de origem, modelanado o personagem.
A plotagem da orelha, por exemplo, acontece da seguinte maneira:

```
void drawCat(Cat cat, int mode)
{
            glPushMatrix(); //Transformações de Translação,
               Rotação e Escala para a modelagem final
        glTranslatef(0.26, 0.45, 0.065);
        glScalef(0.4, 0.4, 0.4);
        glRotatef(10, 1, 0, 0);
        glRotatef(90, 0, 1, 0);
        DrawObject(cat->earLeft, mode);
        glPopMatrix();

        glPushMatrix();
        glTranslatef(0.26, 0.45, -0.065);
        glScalef(0.4, 0.4, 0.4);
        glRotatef(-10, 1, 0, 0);
        glRotatef(90, 0, 1, 0);
        DrawObject(cat->earRight, mode); //Plotagem do gato na
           tela
        glPopMatrix();
}
```

A animação do gato foi implementada a partir da cooperação de duas funções criadas, a animateCat e a walkCat.
Em animateCat, a variável tempo do gato é incrementada aos poucos, sendo utlizada para toda função que envolva animação. Também há um decremento da variável Estamina, diminuindo a velocidade total do gato com o passar do tempo. Por último tem-se a animação de todas as pernas do personagem. A função está descrita abaixo:

```
void animateCat(Cat cat)
{

    if (cat->stamina > 0.3) {
        cat->stamina -= 0.0005; //Decremento da variável stamina
    }
    cat->time += 0.15; //Incremento da variável time
    cat->leg1->time += 2 * cat->speed; //Animação das
    cat->leg3->time += 2 * cat->speed;  //pernas
    cat->leg2->time += 2 * cat->speed;  //do
    cat->leg4->time += 2 * cat->speed;  //gato
}
```

Em walkCat, tendo como um dos parametros de entrada a variável bounds, que determina o limite máximo de distância que o personagem pode atingir em relação ao centro, possui a animação translacional do gato, além de chamar a função animateCat. Nela a posição do gato é incrementada em função da variável time, permitindo que haja movimento. Note

que se a posição ultrapassa a variável bounds, ocorre-se uma rotação, mudando o sentido de movimento do gato.

```
void walkCat(Cat cat, float bounds)
{
    if ( cat->stop == 0) {

        animateCat(cat);
        if (abs(cat->position->x) > bounds  || abs(cat->position
           ->z) > bounds) { //Limite determinado para a posição
            cat->angle += 10;
        }
        float ang = ((int)cat->angle) % 360;
        cat->position->x += cat->stamina * cat->speed * cos(PI *
            ang / 180.0f) * (2 + sin(cat->time * 5)); //Animação
            no eixo x
        cat->position->z -= cat->stamina * cat->speed * sin(PI *
            ang / 180.0f) * (2 + sin(cat->time * 5)); //Animação
            no eixo z
    }
}
```

4.3. Dog

O cachorro é representado por um TAD, chamado dog, seu struct é semelhante ao do gato. Ficando:

```
struct dog {
        //Pernas do cachorro.
    Leg * leg1, * leg2, * leg3, * leg4;
    //Partes do cachorro
    PObject body, neck, head, ear, tail;
    Pt3D position;  //Posição do cachorro na animação
    Pt3D direction; //Direção da movimentação do cachorro
    Pt3D rotation;  //Rotação do gato ( pelo teclado )
    Pt3D translation; //Translação do gato ( pelo teclado )
    float scale; //Escala do cachorro ( teclado )
    Pt3D position, direction, rotation, translation;
    int stop, sit, enabled;
    float angle; //Angulo de rotação no eixo y ( animação )
    float angle2; //Angulo de rotação no eixo x ( animação )
    float maxSpeed; //Velocidade máxima;
    float speed; //Velocidade atual
    float time; //Contador de tempo
};
//Definição do tipo Dog
typedef struct dog * Dog;
```

A função CreateDog é utilizada para inicializar todas as variávis e coordenadas, assim como para fazer a modelagem das partes do cachorro, com a mesma técnica demonstrada no desenho do gato. E a função freeDog facilita a hierarquia de liberação de memória.

A função drawDog, desenha o cachorro, de acordo com uma hierarquia de animação. Primeiramente, são feitas as transformações globais do cachorro, como as obtidas pelo teclado, a posição do cachorro na animação, e as rotações durante sua movimentação. Em seguida, as peças são rotacionadas, transladadas e dimensionadas, de forma que fiquem na sua posição correta na animação, tomando cuidado para que os eixo de rotação estivesse na posição correta.
A função animateDog itera os tempos de animação do cachorro e das pernas, estes são utilizados para alimentar rotações baseados numa senóide, possibilitando animações periódicas.
A função startDog é utilizada para fornecer um conceito de aceleração e também, para fazer o cachorro se levantar. A função stopDog ajuda o cachorro a desacelerar e deixar o ângulo da perna em uma posição desejável para iniciar a operação de sentar. Enquanto a função sit o faz sentar.
Como o cachorro só anda quando está caçando, sua única função de movimentação é chamada hunt, que recebe uma referência para o cachorro e também para o gato. A sua movimentação se dá da seguinte forma:

```
1
2  void hunt(Dog  dog, Cat cat)
3  {
4      if (cat->enabled) { //Se o gato está visível o cachorro pode
           caçar
5          startDog(dog); //Se já está em vel maxima, a função não
               faz nada
6      } else {
7          if (dog->sit)
8              sitDog(dog); //Operação de sentar
9          else
10             stopDog(dog); //Operação de parar ativa sit em hora
                   oportuna
11         return;
12     }
13     //Vetor entre a posição do gato e do cachorro
14     Pt3D dir = subtractPt3D(cPOs, dPos);
15     //Vetor da direção da moviemntação do cachorro
16     Pt3D dDir = CreatePt3D(cos(PI * ang / 180.0f), 0, - sin(PI *
           ang / 180.0f));
17     //Normalizando os vetores
18     normalizedPt3D(dDir);
19     normalizedPt3D(dir);
20     //Ãngulo de correção da rotação do cachorro
21     float theta = acos(dotProduct(dir, dDir) / (normalPt3D(dir)
           * normalPt3D(dDir)));
22     if ((dir->z * dDir->x - dir->x * dDir->z) > 0) {
23         theta = -theta;
24     }
25     //Rotaciona o cachorro na direção do gato
26     dog->angle = ang + theta / PI * 15.0f;
27     //Multiplica a direção pelo vetor velocidade
```

```
28      multiplyPt3D(dDir, dog->speed * (2 + sin(dog->time * 7)));
29      //Desloca o cachorro
30      addToPt3D(dPos, dDir);
31      //Copia dDir para o vevtor de direção dentro do cachorro
32      copyPt3D(dog->direction, dDir );
33      freePt3D(dDir);
34      freePt3D(dir);
35      animateDog(dog);
36      if( distance(dog->position, cat->position) <= 2.0 ) {
37                  //Se o cachorro se aproxima demais, o gato
                       acelera
38          cat->angle += 0.001;
39          cat->speed += cat->stamina * cat->stamina * cat->speed *
                0.007 * ((rand()%100)/100.0);
40          if( cat->stamina < 0.5 )
41              cat->stamina += 0.05;
42      }
43
44      if (distance(dog->position, cat->position) <= 0.5) {
45          //Se o cachorro se aproxima suficientemente, o gato é
               capturado..
46          cat->enabled = 0;
47      }
48  }
```

4.4. Scene

A estrutura scene guarda, inicializa e desenha todos os componentes da cena, como os personagens, câmeras, cercas, luzes e texturas. Sendo guardada na seguinte estrutura de dados:

```
1   struct scene {
2       //Câmera atual
3       Camera currentCam;
4       //Variações de câmeras
5       Camera freeCam, staticCam, followDog, followCat, dogEye;
6       Dog dog; //Cachorro
7       Cat cat; //Gato
8       Wall wall; //Cerca
9       GLuint texture_id[2]; //ID das texturas
10      BMPImage skyTexture; //Imagem com o céu
11      BMPImage grassTexture; //Imagem com o padrão da grama
12      Pt3D spotPosition[4]; //Posição inicial dos 4 spots
13  };
14  //Tipo de dado Scene
15  typedef struct scene * Scene;
```

A função CreateScene inicializa todos os elementos do cenário, e está implementada da seguinte forma:

```
Scene CreateScene(CameraMode mode)
{
    Scene  scn = (Scene) malloc(sizeof(struct scene));
    //Spotlight position
    scn->spotPosition[0] = CreatePt3D( 14,2, 14);
    scn->spotPosition[1] = CreatePt3D(-14,2, 14);
    scn->spotPosition[2] = CreatePt3D(-14,2,-14);
    scn->spotPosition[3] = CreatePt3D( 14,2,-14);
    //reading textures
    getBitmapImageData("sky.bmp", &scn->skyTexture);
    getBitmapImageData("Grass.bmp", &scn->grassTexture);
    glGenTextures(2, &scn->texture_id[0]);
    //Setting up texture
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
       ;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
       ;
    //Setting up Personages
    scn->dog = CreateDog();
    scn->cat = CreateCat();
    setRandomPositions(scn);
    //Setting up Cameras
    scn->freeCam = createCamera(0, 5, -35, FREE);
    rotateCamera(scn->freeCam,0,100);
    scn->staticCam = createCamera(0, 10, -20, STATIC);
    rotateCamera(scn->staticCam,0,100);
    scn->followCat = fCatCamera(0, 10, -20, scn->cat);
    scn->followDog = fDogCamera(0, 10, -20, scn->dog);
    scn->dogEye = dogEyeCamera(scn->dog);
    //Setting up Scenario
    scn->wall = CreateWall();
    //Setting up current camera
    setSceneMode(scn,mode);
    return scn;
}
```

A função drawScene desenha todos os elementos, e é a única função chamada pela função display, na main:

```
void drawScene(Scene  scn, int mode)
{
//Chama função glLookAt da câmera atual
setCamera(scn->currentCam);
glPushMatrix();
//Ativa as spot lights, baseado na posição do gato
spot(GL_LIGHT0,scn->spotPosition[0],scn->cat->position);
spot(GL_LIGHT1,scn->spotPosition[1],scn->cat->position);
```

```
spot(GL_LIGHT2,scn->spotPosition[2],scn->cat->position);
spot(GL_LIGHT3,scn->spotPosition[3],scn->cat->position);
//Ativa luz ambiente
ambient();
//Ativa ponto de luz
light(GL_LIGHT4);
//Desenha 4 vezes a cerca, já transladada a uma distancia do
   centro da cena, rotacionando 90 graus com relação ao centro
   da cena a cada vez.
glColor3f(0.2, .1f, 0.1);
glPushMatrix();
glRotatef(90, 0, 1, 0);
DrawWall(scn->wall, mode);
glRotatef(90, 0, 1, 0);
DrawWall(scn->wall, mode);
glRotatef(90, 0, 1, 0);
DrawWall(scn->wall, mode);
glRotatef(90, 0, 1, 0);
DrawWall(scn->wall, mode);
glPopMatrix();
if( tex == 1 ) {
    textured(scn); //Desenha céu e chão com textura
} else {
    notTextured(); //Desenha céu e chão sem textura
}
//Desenha gato
glPushMatrix();
glColor3f(0.3, 0.3, 0.3);
drawCat(scn->cat, mode);
glPopMatrix();
//Desenha cachorro
glPushMatrix();
glColor3f(0.4, 0.2, .1);
drawDog(scn->dog, mode);
glPopMatrix();
glPopMatrix();
}
```

A inserção do gato através do mouse funciona perfeitamente, independente da câmera utilizada, sendo uma das funcionalidades mais difíceis de abstrair, pois deve-se considerar a projejão perspectiva, assim como a posição da câmera e sua direção.

```
void setCatPos(Scene scn, double fovy, double height, double
   width, double zNear, double x, double y)
{
        //Pega vetor direção da câmeras
    Pt3D view = getDir(scn->currentCam);
    //Normaliza o vetor
    normalizePt3D(view);
    //Produto vetorial entre view e o vetor up da câmera
```

```
8       Pt3D h = crossProduct(view, scn->currentCam->up);
9       normalizePt3D(h);
10          //Produto vetorial entre h e view
11      Pt3D v = crossProduct(h, view);
12      normalizePt3D(v);
13          //Aspect ratio
14      float aspect = (float)width / (float)height;
15      //Conversão para radianos
16      double rad = fovy * PI / 180.0;
17      //Tamanho de v
18      float vLength = tan(rad / 2) * zNear;
19      //Tamanho de h
20      float hLenght = vLength * aspect;
21          //Dimensiona vetor
22      multiplyPt3D(v, vLength);
23      multiplyPt3D(h, hLenght);
24          //Transformação das coordenadas do mouse
25      x -= width / 2.0;
26      y = height - y;
27      y -= height / 2.0;
28      y /= (height / 2.0);
29      x /= (width / 2.0);
30          //Multiplicação por escalar
31      multiplyPt3D(view, zNear);
32      multiplyPt3D(h, x);
33      multiplyPt3D(v, y);
34          //Posição da câmera
35      Pt3D pos = createCopyPt3D(scn->currentCam->pos);
36      addToPt3D(pos, view);
37      addToPt3D(pos, h);
38      addToPt3D(pos, v);
39
40      Pt3D dir = subtractPt3D(pos, scn->currentCam->pos);
41      normalizePt3D(dir);
42
43      if (dir->y != 0) {
44          //Calcula a posição do gato
45          double t = pos->y / dir->y;
46          multiplyPt3D(dir, -t);
47      }
48
49      freePt3D(h);
50      freePt3D(v);
51      freePt3D(view);
52      freePt3D(pos);
53      freePt3D(dir);
54  }
```

4.5. Cameras

As câmeras possuem uma estrutura de dados em comum, mas são intercambiáveis. Sua implementação é dada por:

```
enum cameraMode { FOLLOWDOG, FOLLOWCAT, FREE, STATIC, DOGEYE };
typedef enum cameraMode CameraMode;

struct camera {
    Pt3D pos; //Camera position
    Pt3D dir; //Camera direction
    Pt3D up; //Up direction
    CameraMode mode; //Camera mode
};
typedef struct camera * Camera;
```

Sabe-se a diferença entre os tipos de câmera através de um enum. A criação de cada uma também é diferenciada, existindo construtores diferentes para cada uma:

```
Camera createCamera(GLfloat x, GLfloat y, GLfloat z , CameraMode
    mode);

Camera fDogCamera(GLfloat x, GLfloat y, GLfloat z, Dog dog);

Camera fCatCamera(GLfloat x, GLfloat y, GLfloat z, Cat cat);

Camera dogEyeCamera(Dog dog);
```

A função glLookat, que faz as transformações da câmera, é chamada de forma diferente para cada tipo, desta forma:

```
void setCamera(Camera cam)
{
    switch (cam->mode) {
    case DOGEYE: //Posição da camera segue o cachorro
                         //Direção da câmera segue o
                            deslocamento do cachorro
        gluLookAt( cam->pos->x, cam->pos->y + 1.5, cam->pos->z,
                   cam->pos->x + cam->dir->x,
                   cam->pos->y + cam->dir->y + 1.5,
                   cam->pos->z + cam->dir->z,
                   cam->up->x, cam->up->y, cam->up->z);
        break;
    case FOLLOWCAT:
    case FOLLOWDOG:
        //POsição fixa, mas direção segue o animal
        gluLookAt( cam->pos->x, cam->pos->y, cam->pos->z,
                   cam->dir->x, cam->dir->y, cam->dir->z,
                   cam->up->x, cam->up->y, cam->up->z);
        break;
    default:
```

```
20          //Câmera convencional, olha na direção estipulada pelo
               vetor unitario cam->dir
21          gluLookAt( cam->pos->x, cam->pos->y, cam->pos->z,
22                     cam->pos->x + cam->dir->x,
23                     cam->pos->y + cam->dir->y,
24                     cam->pos->z + cam->dir->z,
25                     cam->up->x, cam->up->y, cam->up->z);
26          break;
27      }
28  }
```

4.6. Transformação 3D dos personagens

Como parte do requerimento do projeto, foram implementadas funções de Translação, Rotação e Escala nos personagens, a partir do teclado. Para realizar tal operação foram armazenadas dentro dos personagens, os ângulos de rotação, em x, y, z, e também os fatores de translação e escala. Sendo que quando o usuário aperta as respectivas teclas no teclado, estas variáveis são modificadas. As transformações acontecem na hora de desenhar na tela, através das funções do OpenGL, debaixo de uma hierarquia de matrizes de transformação.

5. Conclusão

Este trabalho demonstrou vários conceitos de geometria analítica e de computação gráfica, com uma representação interessante dos sweeps, e diferentes tipos de câmeras e projeções. A utilziação de técnicas adicionais de computação facilita grandemente a modelagem de personagens e objetos do cenário, e com um estudo mais aprofundado, pode facilitar a utilização de texturas e cálculo de normais.

References

Hearn, D. and Baker, P. (2004). *Computer graphics with OpenGL*. Pearson Custom Computer Science Series. Pearson Prentice Hall.