Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
E-mail: pozzer@inf.ufsm.br
Data: 9/10/2007

Grafo de Cena

1. Introdução

O crescente poder computacional faz com que **cenas cada vez mais complexas** e mais realistas possam ser geradas e visualizadas em tempo real com o uso de simples computadores. Essa complexidade também se aplica ao desenvolvedor da cena, que precisa especificar e modelar várias características. Isso implica na exigência de estruturas de dados e técnicas para organizar os elementos da cena e mostrar a informação na tela de forma rápida e eficiente.

Para isso existem os grafos de cena, que são **estruturas de dados**, organizadas através de classes, onde por meio de hierarquia de objetos e atributos, pode-se mais facilmente especificar cenas complexas. Cada objeto ou atributo é representado por uma classe, que possui informações sobre sua aparência física, dentre outros fatores.

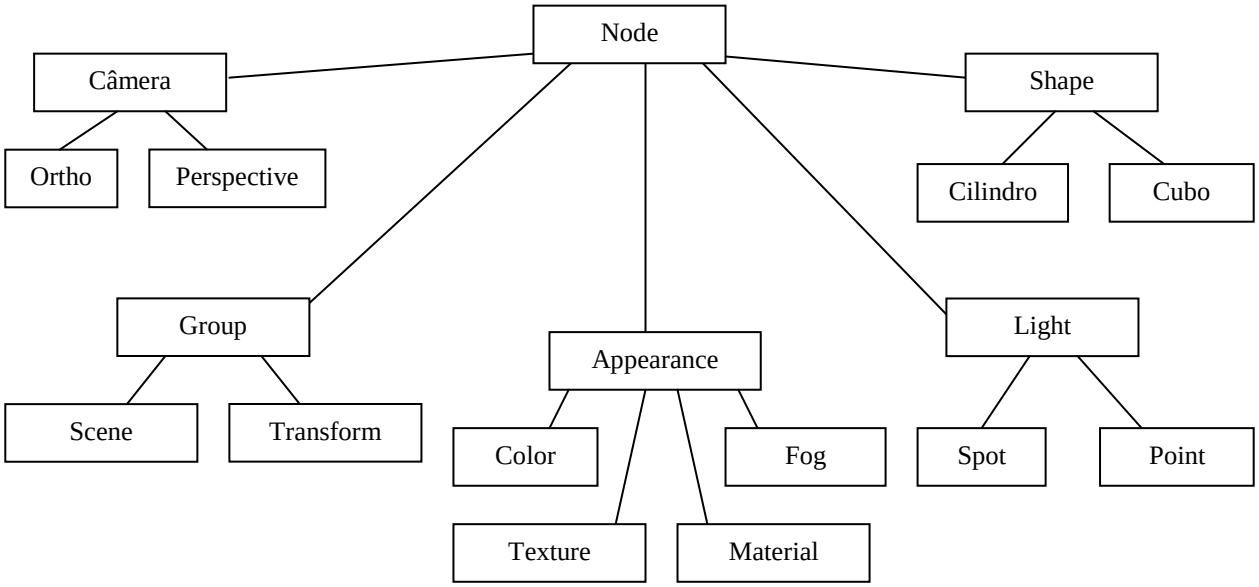
Mesmo sendo uma estratégia dinâmica e estruturada, que permite agrupar objetos comuns, sente-se dificuldade no posicionamento dos objetos, principalmente quando não orientados aos eixos cartesianos. Esse problema é um incentivo ao uso ou estudo de ferramentas gráficas que permitam o posicionamento dos objetos por meio visual, como no caso do **3D Studio Max**.

2. Hierarquia de Classes

Uma maneira simples de representar cenas é pela definição explícita e isolada de cada objeto que a representa. O grande problema desta estratégia pode ser facilmente visualizado caso seja necessário fazer alguma transformação (rotação, translação, etc.) sobre um objeto "composto" por diversas primitivas simples. Neste caso deve-se aplicar a transformação sobre cada entidade, o que em caso de rotação não é uma tarefa simples.

O uso de hierarquia de objetos é uma maneira mais plausível de representar a cena, onde agrupando entidades básicas, como caixas, esferas, cilindros, dentre outros, pode-se gerar objetos ou cenas mais complexos. Apesar de exigir uma maior quantidade de informações (inter-relacionamento entre primitivas), pode facilmente resolver o problema acima exposto de aplicar uma transformação sobre objetos complexos.

Para se implementar um grafo de cena pode-se utilizar uma hierarquia de classes, como mostrada na seguinte figura. A principal classe é a `Node`, de onde derivam todos as demais classes. Quando o construtor de qualquer classe é chamado, o construtor da classe `Node` também é, recebendo como argumento uma string com o nome do objeto alocado. Este nome pode ser útil para poder visualizar a árvore que define o cenário, pois a função `Render()` de qualquer objeto chama o método `MostraDados()` da classe `Node`, que mostra o nome do objeto em questão, de forma endentada, de acordo com as transformações associadas.



2.1. Classe Group e Derivadas

A classe `Group` contém a descrição de um ou mais elementos da cena. Quando aplicada alguma transformação sobre um grupo, todos os elementos geométricos sofrem o mesmo efeito da transformação, o que é de grande utilidade no posicionamento de objetos complexos e hierarquizados. Cada grupo pode conter no máximo 100 elementos associados.

Esta classe tem duas classes derivadas: `Transform` e `Scene`. A classe `Transform` pode possuir transformações de rotação e/ou translação associadas e, recursivamente, outros grupos de elementos da cena, que são compostos por transformações, shapes, materiais, etc. Na implementação, sempre que um objeto é composto por um vetor de transformações, a `transf[0]` representa a transformação que engloba todas as demais, e será a única anexada diretamente a árvore da cena. A classe `Transform` também tem um parâmetro animação, que permite aplicar rotações (animações) aos objetos que estão sob sua estrutura.

A classe `Scene` contém todos os objetos da cena, que estão agrupados segundo nós de transformação. Esta classe possui o método `Render()`, que durante o processo de renderização da cena, chama todos os demais métodos `Render()` dos demais elementos da árvore. A classe também possui métodos para fazer o posicionamento da câmera e configuração de fontes luminosas.

Classe Scene

```
class scene: public group{
public:
  ...
  void Render()
  {
    if(SetupCamera())
        ;//Encontrou Camera
    else
        ;//Nao Encontrou Camera
    if (SetupLights())
    {
       glEnable(GL_LIGHTING);
    }
    else
    {
       ;//Nao encontrou fonte
    }
    group::Render();
    glFlush();
  }
```

Class Transform

```
class transform: public group{
  ...
public:
  void Render()
  {
    glPushMatrix();
    glTranslated(tx, ty, tz);
    glRotated(ang, eixo_x, eixo_y, eixo_z);

    if( anim_ang > 0.01 )
    {
      ang_anim += anim_ang;
      glRotated(ang_anim, a_x, a_y, a_z);
    }
    group::Render();
    glPopMatrix();
  }

  int SetupCamera()
  {
```

```
};
```

```
    if(group::SetupCamera())
    {
       //Posicionou camera pela transformacao
       return 1;
    }
    return 0;
  }

  int SetupLights()
  {
    int lights = 0;
    glPushMatrix();
    glTranslated(tx, ty, tz);
    glRotated(ang, eixo_x, eixo_y, eixo_z);

    if( anim_ang > 0.01 )
    {
      ang_anim += anim_ang;
      glRotated(ang_anim, a_x, a_y, a_z);
    }

    lights =  group::SetupLights();
    glPopMatrix();
    return lights;
  }
};
```

```
void display (void)
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       //renderiza a cena
       sc->Render();

       glutSwapBuffers();
}
```

Class SpotLight

```
class spotlight: public lightsource{
public:
  int cutoff;
  GLfloat direcao[3];
  spotlight()
  {
     cutoff = 10;
     direcao[0] = 0;
     direcao[1] = -1;
     direcao[2] = 1;
  }


//dados da fonte direcional.
  void SetApperture(int apperture)
  {
    cutoff = apperture;
  }
```

```
void Render()
{
  glPushMatrix();
  GLfloat color[ ] = {1,1,0.2F,1};
  glMaterialfv(GL_FRONT,GL_EMISSION,color);
  glTranslated(...);
  glutSolidSphere(raio,14,16);
  glPopMatrix();

  /* restore material */
  GLfloat none[ ] = {0,0,0,1};
  glMaterialfv(GL_FRONT,GL_EMISSION,none);
}

int SetupLights()
{
  if(ligada==1)
    glEnable (GL_LIGHT1);
  else
    glDisable (GL_LIGHT1);

  glLightfv (GL_LIGHT1, GL_SPOT_DIRECTION, dir);
  glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, cut);
  glLightfv(GL_LIGHT1, GL_POSITION, pos);
  glLightfv(GL_LIGHT1, GL_DIFFUSE,  dif);
  return 1;
}
```

Class Group

```
class group: public node{
   node *chield[100];
   int num_chields;
public:
   group(char *name):node(name)
   {
      num_chields = 0;
   }

   int SetupCamera()
   {
      int i;
      for (i=0; i<num_chields; i++)
      {
         if(chield[i]->SetupCamera())
            return 1;
      }
      return 0;
   }
   int SetupLights()
   {
      int n=0, i;
      for (i=0; i<num_chields; i++)
      {
         n += chield[i]->SetupLights();
      }
      return n;
   }

   void add(node *n)
   {
      if(num_chields<100)
         chield[num_chields++] = n;
      else
         printf("Vetor cheio");
   }

   void Render()
   {
      int i;
      for (i=0; i<num_chields; i++)
      {
         chield[i]->Render();
      }
   }
};
```

2.2. Classe Appearance

Todas as características que representam aparências de cor de objetos são representados pela classe `Appearance`. Quando uma `Appearance` é associada à árvore de cena, todos os elementos abaixo da hierarquia passam a usar as características da `Appearance` corrente. A aparência só será alterada se outra Appearance for anexada à árvore. Esta classe deriva quatro novas classes: material, textura, fog e cor.

2.3. Classe Light

Nesta classe são especificadas as fontes luminosas, que podem ser do tipo *point source* ou *spot*. Da mesma forma como na classe Material, esta também possui descrição dos valores de reflexão ambiente, difuso e especular da luz sobre as superfícies. Pode-se também especificar uma geometria para a fonte, no caso uma esfera, que terá uma posição espacial e um fator emissivo de luz, no caso na cor amarela.

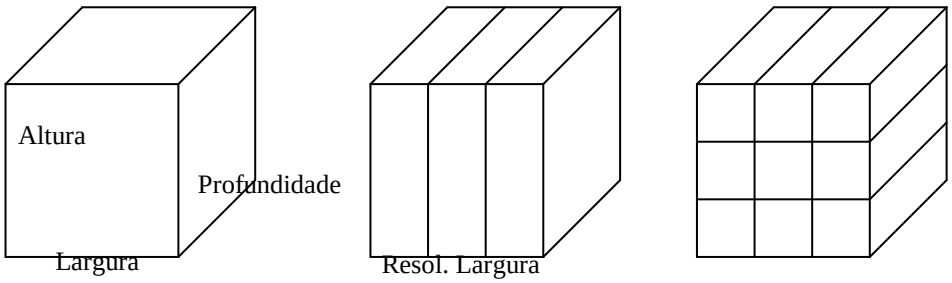
A classe derivada `Plight` especifica uma luz pontual e possui o método `SetupLights()`, chamado da classe `Scene`, que habilita ou desabilita a fonte, em função da variável *ligada*, especificada pelo método `setLuzLigada()`, que indica se a fonte está ligada ou não. Com esta variável pode-se ativar ou desativar a fonte durante o processo de visualização da cena.

A subclasse `Spotlight`, além das variáveis e métodos da classe `Plight`, também possui variáveis para indicar abertura e direção da luz.

2.4. Classe Shape

A classe `Shape` é uma classe pai para a definição de objetos geométricos. Neste trabalho, dela são derivas duas classe para representar cubos e cilindros. Todas as primitivas são geradas centradas na origem do sistema global, e após transladadas e rotacionadas pelas transformações que as possuem.

O cubo é definido por três parâmetros: altura, largura e profundidade. Além destes parâmetros, outros três, usados para refinar a superfície, são usados: res_altura, res_largura e res_profund. Com a superfície mais dividida, pode-se melhorar simular o processo de iluminação. A seguinte figura mostra exemplos de 3 cubos com diferentes resoluções.



Pode-se especificar diferentes resoluções para cada dimensão do cubo, dependendo da região que irá sofrer maior interferência da luz. Independente da resolução, apenas são armazenados no objeto as três variáveis de dimensão. Os parâmetros de resolução são usados apenas no processo de geração dos polígonos enviados para o estágio de geometria, na chamada do método `Render()` .

Para aplicar textura sobre o cubo, são definidas as seguintes variáveis e métodos:

- **textura:** variável do tipo long int que possui representação hexadecimal no formato 0x111111, que indica qual face irá receber textura. Neste caso, as 6 faces irão receber a mesma textura. A verificação por face é baseada em uma operação lógica bitwise. Caso a condição for falsa, é desabilitada a textura para a dada face do cubo. Se textura=0x100000, apenas a face superior receberá textura.
- **max_ext:** indica qual é o maior lado do cubo. Esta variável é usada na aplicação da textura, de forma a evitar que a mesma sofra distorções em S e T.
- **escala_textura:** Indica como será replicada a textura sobre o objeto. Se escala_textura=1, então será aplicada uma única textura em função de max_ext, ou seja, a textura será cortada em faces menores que a maior face. Desenvolveu-se o seguinte cálculo para determinar incrementos da textura para não haver distorção, levando em consideração a resolução das faces:

O cilindro é definido em função dos seguintes parâmetros:

- **raio_d1** e **raio_d2:** determinam os raios internos (de dentro) das duas extremidades do cilindro. Para um cilindro sem furo, este valor deve ser zero.
- **raio_f1** e **raio_f2:** determinam os raios externos (de fora) das duas extremidades do cilindro.
- **altura:** especifica a altura em y do cilindro.
- **n_lados:** número de lados que o cilindro vai ter. Se n_lados for 4, tem-se uma representação do cubo.

Fazendo-se diferentes os raios das extremidades do cilindro, pode-se gerar cones, ou uma seção de cone.

O cálculo dos vértices é realizado quando o objeto é criado, e armazenado uma matriz. Isso é feito para evitar recalcular todos os vértices cada vez que o objeto é renderizado. Usando-se este algoritmo sucessivamente para diferentes valores de y e raio pode-se facilmente gerar esferas. Na implementação, a primitiva cilindro não permite o uso de texturas.

3. Geração da cena

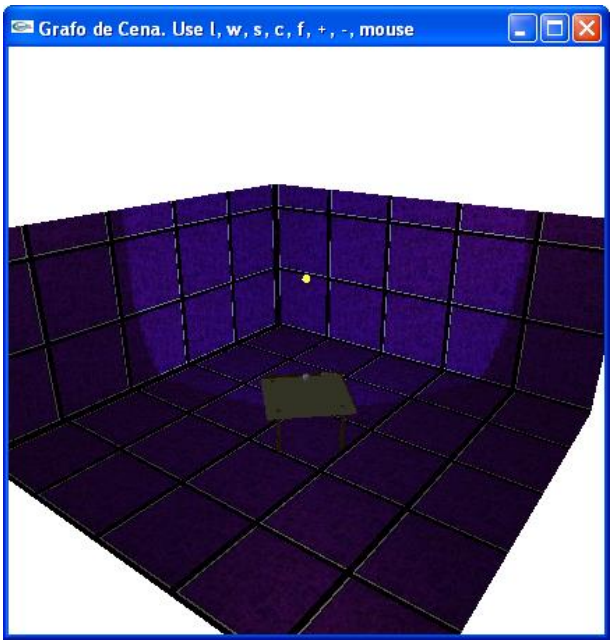
Para gerar a cena, deve-se inicialmente criar os objetos e depois uni-los de forma hierárquica com o uso de transformações. O objeto cena deve ser o nó raiz. Para mais detalhes, veja o demo Grafo_de_Cena.

```
void gera_mesa()
{
  cb_mesa[PERNA] = new cubo("pernas");
  cb_mesa[PERNA]->setDimensoes(1.3, 12, 1);
  cb_mesa[PERNA]->setResolucao(1,1,1, 0.9);
  cb_mesa[PERNA]->setTextura(0x111111);

  cb_mesa[TAMPO] = new cubo("tampo da mesa");
  cb_mesa[TAMPO]->setDimensoes(22,1,22);
```

```
void gera_luminaria()
{
   spot = new spotlight("luz spot");
   spot->SetShiness(4);
   spot->SetRaio(0.5);
   spot->SetPosition(0.2, -0.2, 0.3);
   spot->SetApperture(62);
   spot->setOrientacao(0, -1, 0);
```

```
  cb_mesa[TAMPO]->setResolucao(48, 1, 48, 1);
  cb_mesa[TAMPO]->setTextura(0x101111);

  tr_mesa[1] = new transform("tr perna 1");
  tr_mesa[1]->setTranslacao(-8, -6, 8);
  tr_mesa[1]->add(cb_mesa[PERNA]);

  tr_mesa[2] = new transform("tr perna 2");
  tr_mesa[2]->setTranslacao(8, -6, 8);
  tr_mesa[2]->add(cb_mesa[PERNA]);
  ...
  tr_mesa[0] = new transform("tr de toda mesa");
  tr_mesa[0]->setTranslacao(0,-15,0);
  tr_mesa[0]->setAnimacao(0.2, 0,1,0);

  tr_mesa[0]->add(mt_cena);
  tr_mesa[0]->add(cb_mesa[TAMPO]);
  tr_mesa[0]->add(tr_mesa[1]);
  tr_mesa[0]->add(tr_mesa[2]);
  tr_mesa[0]->add(tr_mesa[3]);
  tr_mesa[0]->add(tr_mesa[4]);

  tr_mesa[0]->add(tr_cone);

  tr_mesa[0]->add(tt_retrato );
  tr_mesa[0]->add(tr_retrato[0]);
}

void gera_sala()
{
  sala = new cubo("Sala" );
  sala->setDimensoes(115,55,115);
  sala->setResolucao(10, 10, 10, 5);
  sala->setNormal(true);
  sala->setTextura(0x111111);
}
```

```
   spot->SetAmbient(1,1,1);

   cl_cone = new cilindro("cone de luz");
   cl_cone->setAltura(2);
   cl_cone->setResolucao(11,2);
   cl_cone->setRaios_1(0,0.1);
   cl_cone->setRaios_2(1,1.1);

   tr_cone = new transform("transform cone");
   tr_cone->setTranslacao(0.0, 4.1, 0);
   tr_cone->setRotacao(-105, 0 ,0, 1);
   tr_cone->setAnimacao(6.75, 0.2, 1, 0.2);

   tr_cone->add(cl_cone);
   tr_cone->add( spot );
}

void init()
{
  ...
  gera_materiais();
  gera_luzes();
  gera_retrato();
  gera_luminaria();
  gera_mesa();
  gera_sala();

  sc = new scene();
  sc->add( luz1 );
  sc->add( mt_cena );
  sc->add( tt_chao );
  sc->add( sala   );
  sc->add( tt_mesa );
  sc->add( tr_mesa[0] );
  ...
}
```



4. Referências

[1] Woo, M., Neider, J., Davis, T., Shreiner, D. **OpenGL, Programming Guide**, Third Edition, 1999.
[2] Moller, T., Eric, H. **Real-Time Rendering**, 1999.