Índice

1. INTRODUÇÃO E BREVE HISTÓRICO **1**

2. CARACTERÍSTICAS BÁSICAS DE VRML **2**

3. GERANDO MUNDOS VRML **3**

3.1. FORMAS PRIMITIVAS E SUAS COMBINAÇÕES **3**
3.1.1. EXPERIMENTANDO COM TEXTOS 5
3.2. TRANSFORMAÇÕES GEOMÉTRICAS **6**
3.3. ALTERANDO APARÊNCIA - NÓ APPEARANCE **8**
3.4. REUTILIZANDO FORMAS , ELEMENTOS E DEFINIÇÕES **11**
3.5. COMPONDO FORMAS COMO CONJUNTO DE FACES **13**
3.6. FUNDOS E CENÁRIOS **14**
3.7. ILUMINAÇÃO **16**
3.7.1. DIRECTIONALLIGHT 17
3.7.2. POINTLIGHT 17
3.7.3. SPOTLIGHT 19
3.8. CONTROLANDO DETALHAMENTO DE ELEMENTOS - NÓ LOD **20**
3.9. ANIMANDO AS FORMAS **22**
3.9.1. ANIMAÇÕES BÁSICAS 22
3.9.2. CONTROLANDO AS ANIMAÇÕES 26
3.10. CONTROLANDO O PONTO DE VISTA E A NAVEGAÇÃO **29**
3.11. ADICIONANDO SONS E FILMES AO CENÁRIO **30**
3.12. SENTINDO A PROXIMIDADE DO USUÁRIO **32**
3.13. UNINDO CENÁRIOS VIRTUAIS - LINKS **34**

4. COMBINANDO VRML E JAVASCRIPT **36**

4.1. INTRODUÇÃO **36**
4.2. DESENVOLVENDO EXEMPLOS **37**

5. COMBINANDO VRML E JAVA - DICAS **39**

6. REFERÊNCIAS BIBLIOGRÁFICAS **39**

Índice de Figuras

Figura 1 - Elementos Geométricos do Cone 4

Figura 2 - Elementos Geométricos do Cilindro.. 4

Figura 3 - Exemplo de texto em VRML... 6

Figura 4 - Estrutura de árvore da Chaminé Exemplo 7

Figura 5 - Cone colorido - uso de 'appearance'.. 9

Figura 6 - Cone com aplicação de Textura 10

Figura 7 - Plano de aplicação de textura S x T 10

Figura 8 - Alterando Aplicação de Textura - textureTransform............................ 11

Figura 9 - Coluna com uso de DEF e USE.. 12

Figura 10 - Grupo de Colunas - uso de Inline 13

Figura 11 - Cubo - uso de IndexedFaceSet... 14

Figura 12 - Background aplicado 15

Figura 13 - Background com elementos de imagem ".gif".................................... 16

Figura 14 -uso de DirectionalLight.. 17

Figura 15 - uso de PointLight - caso 1... 18

Figura 16 - uso de PointLight - caso 2... 19

Figura 17 - Elementos de SpotLight.. 20

Figura 18 - uso de SpotLight.. 20

Figura 19 - Elementos do nó TimeSensor 22

Figura 20 - Elementos do nó Sound 32

VRML - a Web em 3D

Alexandre Cardoso

1. Introdução e breve histórico

VRML (*Virtual Reality Modeling Language*) tem sido aplicada em diversos projetos para concepção de mundos virtuais [ELLIS 1994, CHEN 1997, MATSUBA 1999 CARDOSO et al. 1999] e é uma importante aliada no desenvolvimento de mundos tridimensionais interativos na Web. VRML tem uma história fundamentada na colaboração de diversos pesquisadores e importantes empresas relacionadas com a Computação Gráfica.

A partir de um projeto, iniciado em 1989, na *Silicon Graphics Inc.* por Rikk Carey e Paul Strass, que propunha o desenvolvimento de uma infra-estrutura para aplicações gráficas 3D interativas com duas características básicas: capacidade de criação de uma gama variada de aplicações 3D e a possibilidade de usar este ambiente para construir uma nova interface para 3D, nasce o *Scenario.*

Em 1992, fundamentado nas proposições do *Scenario*, surge o *Iris Inventor 3D*, uma ferramenta fundamentada em C++ que, posteriormente, fundamenta muito da semântica de VRML. A revisão do *Inventor* realizada em 1994 origina o *Open Inventor* caracterizada por sua portabilidade a uma grande variedade de plataformas e baseada no *Open GL* da *Silicon*. O manual de referência da mesma descrevia os objetos e formatos de arquivos que, adequados por *Gavin Bell*, originariam a especificação do VRML 1.0.

Em 1994, *Mark Pesce* e *Tony Parisi* construíram um protótipo de um navegador 3D para a *World Wide Web - WWW*, chamado *Labyrinth*. Um ano depois, *Mark* e *Brian Behlendorf* criaram a lista de discussão do VRML e publicaram uma chamada para propostas de uma especificação formal para 3D na *WWW*.

De discussões nesta lista, iniciadas por *Gavin BellI,* usando como ponto de partida o manual do *Inventor* nasce a especificação do VRML 1.0. Em 1995, VRML 1.0 foi alvo de muitas correções e novas discussões, de forma que nasce a proposta de uma segunda especificação, a do VRML 1.1 (ou VRML 2.0).

Aprimorada e capaz de definir comportamentos (com mais interação e animação) para elementos 3D, a nova proposição foi submetida à lista em 1996. As propostas apresentadas pela *Silicon* em colaboração com a *Sony* e com a *Mitra* receberam a grande maioria dos votos e originaram o documento de definição de VRML 2.0. Em agosto de 1996, esta versão foi publicada no *SIGGRAPH´96* em *New Orleans*.

A versão final deste texto, com diversas correções e modificações técnicas foi publicada em Abril de 1997, ficando conhecida como VRML´97 e é a versão em uso até os dias atuais. Suas principais características estão relacionadas com o desenvolvimento de cenários mais realísticos, prototipação (capacidade de encapsular novos de forma a criar novos nós), interação direta com o usuário através de sensores, interpoladores e criação de animações usando *scripts*.

2. Características básicas de VRML

Arquivos que simulam mundos 3D em VRML, são na verdade uma descrição textual, na forma de textos ASCii. Assim, por meio de qualquer processador de textos, um desenvolvedor pode conceber tais arquivos, salvá-los e visualizar resultados no navegador de Internet associado a um *plug-in*. Estes arquivos definem como estão as formas geométricas, as cores, as associações, os movimentos, enfim, todos os aspectos relacionados com a idéia do autor [AMES 1997]. Quando um dado navegador - *browser* - lê um arquivo com a extensão .wrl, o mesmo constrói o mundo descrito, ativando um *plug-in* compatível.

De forma simplificada, um arquivo VRML se caracteriza por quatro elementos principais: - o cabeçalho (obrigatório); - os protótipos; - as formas, interpoladores, sensores, scripts e as rotas. Assim, um arquivo VRML pode conter:

- *Header*;
- *Prototypes*;
- *Shapes*, *Interpolators*, *Sensors*, *Scripts*;
- *Routes*.

Nem todos os arquivos contêm todos estes componentes. Na verdade o único item obrigatório em qualquer arquivo VRML é o *header*. Porém, sem pelo menos uma figura, o navegador não exibirá nada ao ler o arquivo. Outros componentes que um arquivo VRML também pode conter, são:

- *Comments*;
- *Nodes*;
- *Fields*, *field values*;
- *Defined node names*;
- *Used node names*;

O cabeçalho (*header*) é composto pela instrução "#VRML V2.0 utf8" e sua omissão impossibilita o *plug-in* do navegador de ler o arquivo em questão. Os protótipos (*proto*) contém a definição de novos nós que podem ser usados no arquivo em definição. A seção de descrição de formas (*shapes* etc) apresenta a descrição das formas que serão exibidas no navegador e a seção de rotas (*routes*) contém a definição das trocas de mensagens entre os nós de descrição de formas, interpoladores, sensores e *scripts*.

A concepção de cenários tridimensionais, usando VRML, se baseia na elaboração de uma grafo direcionado acíclico, contendo diferentes ramos - nós - que, associados de forma correta podem ser agrupados ou estarem independentes uns dos outros. A grande diversidade destes nós (54 pré-definidos), incluindo primitivas geométricas, propriedades de aparência, sons (e propriedades) e vários tipos de nós de agrupamentos, é uma das principais características e qualidades da linguagem.

É permitido reutilização de código através da prototipação, baseada na definição de novos nós (*protos*) que podem ser utilizados por outros arquivos e ativados dentro de um arquivo como um nó externo, sem duplicação de códigos.

A concepção de formas se dá através da associação de elementos 3D geométricos pré-definidos, tais como Cones, Cilindros, Esferas, Caixas etc de atributos variáveis e que podem estar associados a texturas.

A modificação de fundos está possibilitada pelo uso de nós específicos - *backgrounds*, - que permitem simular ambientes diferenciados que se assemelham a condições que variam de um lindo dia de sol, um dia nublado ou com muita neblina até a noites.

É possível o controle de aparência de elementos do cenário, bem como a inserção de diferentes formas de fontes de luz (pontuais, direcionais, ambiente), visando dar mais realismo ao cenário concebido. Recursos de acrescentar sons e filmes também estão disponíveis por utilização de nós específicos e são compatíveis com os principais formatos de áudio e vídeo: .mpeg, .mpg, .mid., .wav.

Podem ser elaborados *scripts* que facilitam as animações utilizando-se *Java* ou *JavaScript* de forma a complementar a troca de informações entre os elementos do mundo virtual. Esta propriedade provê possibilidade de animações e de dinamismo às formas concebidas e inseridas no cenário. O código em *JavaScript* pode fazer parte do arquivo original.

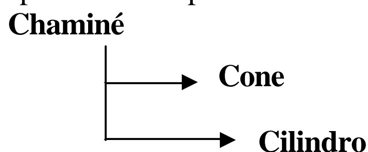
3. Gerando Mundos VRML

3.1. Formas primitivas e suas combinações

A produção de arquivos VRML, de forma a definir um objeto, um cenário ou um ambiente 3D com diversos objetos e links começa pela distinção do que deve ser associado, em termos de primitivas e formas gráficas, para obter o resultado desejado. A associação de tais elementos relaciona-se com a concepção de uma árvore, onde as folhas representam os nós VRML (geralmente formas geométricas) que devem ser utilizados.

Assim, para gerar um mundo VRML inicie pelo planejamento das formas que serão necessárias e sua hierarquia. Como exemplo, suponhamos que se deseje produzir uma chaminé, onde serão associados um cone e um cilindro. A estrutura da chaminé necessitará de duas primitivas disponíveis.

Chaminé



Para gerar um cone, teríamos de definir sua características fundamentais, relativas a altura, raio da base, cor etc, como pode ser verificado na Figura 1.

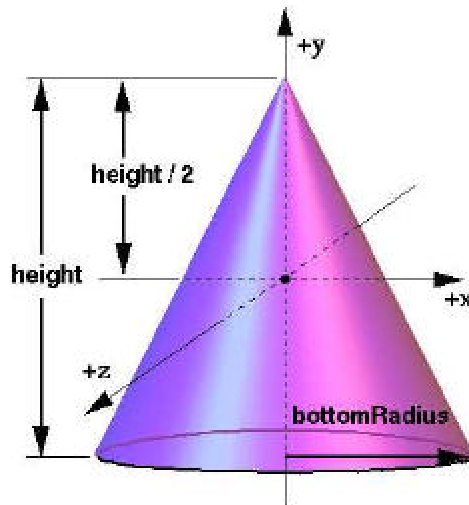


Figura 1 - Elementos Geométricos do Cone

Há dois aspectos importantes na definição: elementos da geometria (como o raio da base e a altura) e elementos da aparência da forma (cor, textura etc). Os nós VRML diferenciam tais elementos por dois campos distintos: *'geometry'* e *'appearance'*.

Assim, para definir um cone como o proposto, teríamos a descrição textual abaixo como arquivo VRML:

```
Shape {
        appearance Appearance {}
        geometry Cone {
            height 2.0
            bottomRadius 1.0 }
    }
```

Observe que não foram feitas quaisquer definições de aparência, levando a forma gerada, de acordo com o texto acima, a ter aparência branca.

Para gerar o Cilindro, os elementos de geometria seriam os apresentado na Figura 2.

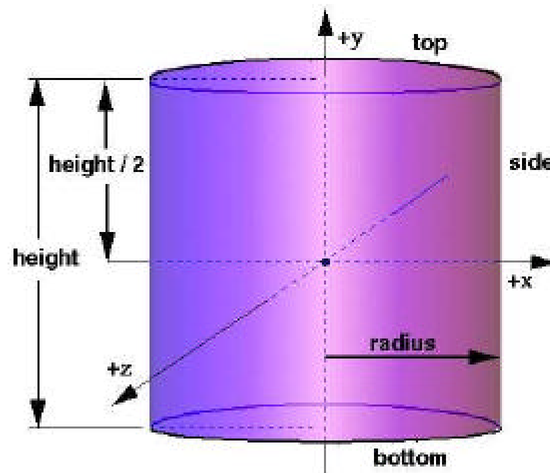


Figura 2 - Elementos Geométricos do Cilindro

A definição, em VRML, do nó *Cylinder* pode ser dada por:

```
Shape {
        appearance Appearance { }
        geometry Cylinder {
                bottom TRUE
                top     TRUE
                radius 2.0
                height 3.0
        }
}
```

Visando a construção da chaminé, temos de deslocar o cone, de forma que sua base coincida com o topo do cilindro. Tal deslocamento refere-se a uma translação da forma "*Cone*" no eixo Y, do valor da metade da altura do cilindro (*height*/2) mais a metade da altura do Cone (*height_cone*/2), uma vez que, conforme ilustra a Fig. 2 , teríamos metade do cilindro acima e metade do cilindro abaixo do eixo 'x' (o mesmo vale para o Cone).

Destacam-se as formas geométricas básicas:

- Box
- Cone
- Cylinder
- Sphere
- Text - formatar textos
- ElevationGrid
- Extrusion
- IndexedFaceSet
- IndexedLineSet
- PointSet

3.1.1. Experimentando com Textos

É possível inserir textos nos ambientes 3D concebidos através do nó *Text*, cuja sintaxe é dada por:

```
Text {
        string []
        length []
        maxExtent 0.0
        fontStyle NULL
}
```

A apresentação do texto pode ser formatada com uso do nó *FontStyle*, que permite definir a fonte que será utilizada, seu tamanho, espaçamento de caracteres, como o texto será justificado etc.

O trecho de código abaixo, relativo à geometria, gera um texto simples como exemplo - Figura 3.

```
geometry Text {
                string  ["SVR 03 - Curso de VRML 2.0"]
                fontStyle FontStyle {
                                size 0.9
                                justify "MIDDLE"
```

```
                spacing         0.3
                style "BOLD"
                family "SERIF"
                        }
        }
```

Figura 3 - Exemplo de texto em VRML

3.2. Transformações Geométricas

O nó VRML que permite que façamos transformações geométricas (translação, rotação e escala) sobre as formas definidas é o nó *'Transform'*. *'Transform'* é, de fato, um nó de agrupamento, que, para todas as formas de seus filhos (*'children'*) aplica as transformações definidas nos campos *'scale'*, *'translation'* e *'rotation'*:

Sintaxe básica do nó *'Transform'*:

```
Transform {
        scale 1.0 1.0 1.0
        translation 0.0 0.0 0.0
        rotation 0.0 0.0 0.0 0.0
        children [
        ]
}
```

'Scale' escala em X,Y,Z, respectivamente;

'Translation': translação nos eixos X,Y,Z respectivamente;

'rotation': rotação em X, Y, Z pelo argumento (em rad) do quarto campo;

'children': lista dos elementos a qual devem ser aplicadas as transformações.

Assim, o arquivo final para geração da chaminé deveria ter um código semelhante às linhas abaixo:

```
Group{
        children [
                        Transform {
                                scale 1.0 1.0 1.0
                                translation     0.0 3 0.0
                                rotation 0.0 0.0 0.0 0.0
                                children [
                                        Shape {
                                        appearance Appearance {}
                                        geometry Cone {
                                                height 3.0
                                                bottomRadius 2.5 }
                                }
                                ]
```

```
                        }
                        Shape {
                                appearance Appearance { }
                                geometry Cylinder {
                                        bottom TRUE
                                        top     TRUE
                                        radius 1.0
                                        height 3.0
                                }
                        }
                ]
        }
```

As linhas de código eqüivalem à uma árvore como a apresentada na Figura 4.

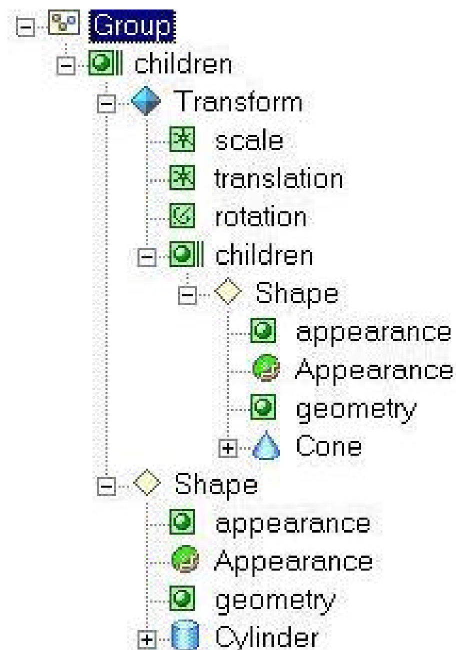


Figura 4 - Estrutura de árvore da Chaminé Exemplo

Observa-se que um código simples, iniciado por um cabeçalho padrão (#VRML V2.0 utf8) e seguido por uma árvore que contém a descrição de dois elementos geométricos, um cilindro e um cone gera o efeito da construção de uma pequena chaminé. O grafo que descreve a cena é dado por um nó principal *Group* que contém dois nós como filhos - *children*:

- nó *Transform*, capaz de promover a translação de uma dada forma, no caso, de um dado cone;

- nó *Shape* que é usado para definir uma outra forma, no caso, um cilindro.

A facilidade de implementar formas geométricas é facilmente observada através deste exemplo, que permite verificar que um arquivo que tem menos de 1kb é capaz de produzir uma forma geométrica 3D,que a partir dos recursos do navegador VRML pode ser vista de diferentes ângulos, rotacionada para verificação, aproximada ou afastada, girada, apontada etc, tornando a navegação uma agradável experiência interativa com a forma.

Até meados do ano de 2000, a edição de textos que descreviam formas em VRML não dispunha de um editor de textos exclusivo para este fim. De olho neste nicho de mercado, a empresa *Parallel Graphics* lançou o VrmlPad® [VRMLPAD 2000], que se encontra na versão 2.0 e que facilita sobremaneira a edição de tais textos.

3.3. Alterando Aparência - nó appearance

De maneira a acomodar a definição adequada da aparência de uma forma, o nó *appearance* tem a seguinte sintaxe:

```
appearance Appearance {
        material Material {
            ambientIntensity 1.0
            diffuseColor 1.0 0.0 0.0
            emissiveColor 1.0 0.0 0.0
            specularColor 1.0 0.0 0.0
            transparency 0.3
            shininess 0.4
        }
        texture ImageTexture {url ""}
}
}
```

As variáveis *diffuseColor*, *emissiveColor* e *specularColor* referem-se à definição das cores do objeto relativas à iluminação difusa, especular e a cor que o mesmo emite. A transparência do mesmo é dada pelo campo *'transparency'* e caso seja necessário, podemos aplicar a textura relativa à uma imagem (JPEG, GIF etc) ao objeto dando o endereço da mesma no campo *ImageTexture*. Neste caso, as definições de cores feitas em 'Material' são perdidas.

Como exemplo, para alterar o cone já definido anteriormente, dando-lhe uma aparência acinzentada, poderíamos definir o nó *appearance* como:

```
appearance Appearance {
        material Material {
            diffuseColor .8 0 .13
            specularColor .77 .77 .77
            emissiveColor .1 0 .02
            ambientIntensity .0767
                }
        }
```

O resultado da alteração da forma do cone pode ser visto na Figura 5.

Figura 5 - Cone colorido - uso de 'appearance'

Algumas vezes, no entanto, deseja-se aplicar uma textura a uma dada forma, ao invés de definir propriedades de cor para a mesma. Neste caso, imagens do tipo JPEG, GIF e PNG podem ser aplicadas à forma desejada com uso do nó *'Material'*. São campos possíveis de serem utilizados: *ImageTexture*, *PixelTexture* e *MovieTexture.*

Como exemplo, a aplicação de uma textura feita ao cone, similar ao definido anteriormente, conforme o código abaixo pode ser visto na Figura 6 .

```
Transform {
        scale 1.0 1.0 1.0
        translation     0.0 3 0.0
        rotation 0.0 0.0 0.0 0.0
        children [
                Shape {
                appearance Appearance {
                        material Material {
                                diffuseColor .8 0 .13
                                specularColor .77 .77 .77
                                emissiveColor .1 0 .02
                                ambientIntensity .0767
                        }
                        texture ImageTexture {
                                url "brick.jpg"}
                }
                geometry Cone {
                        height 3.0
                        bottomRadius 1.5 }
                }]}
```

Figura 6 - Cone com aplicação de Textura

Há possibilidade de controlar o mapeamento de texturas com utilização dos nós *TextureCoordinate* e *TextureTransform*. O nó *TextureTransform* pode ser usado como um valor do campo de valor no nó *Appearance*.

O trecho de código abaixo altera a aplicação de textura no cone, por uma escala de 2.0 para as direções S e T, respectivamente. A Figura 7 apresenta o plano de aplicação de texturas S x T.

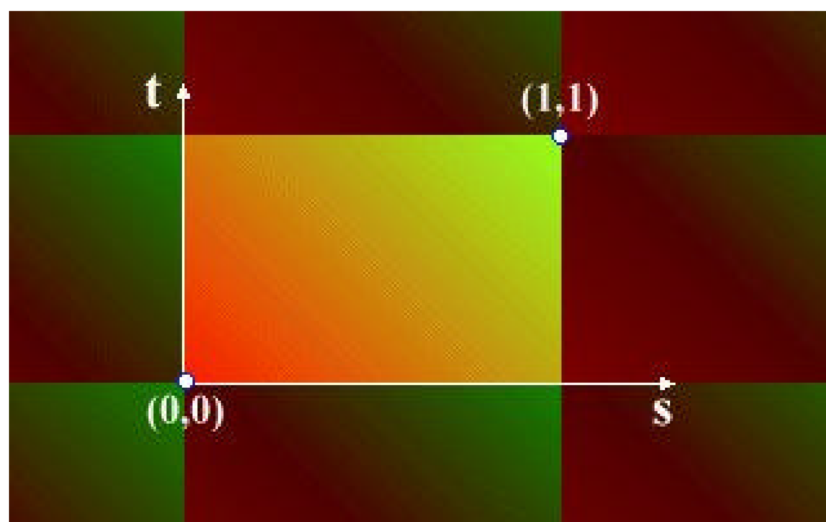


Figura 7 - Plano de aplicação de textura S x T

O resultado pode ser visto na Figura 8.

```
textureTransform TextureTransform {
                          scale 2.0 2.0
                                    }
                          texture ImageTexture {
                          url "brick.jpg"}
                                    }
```

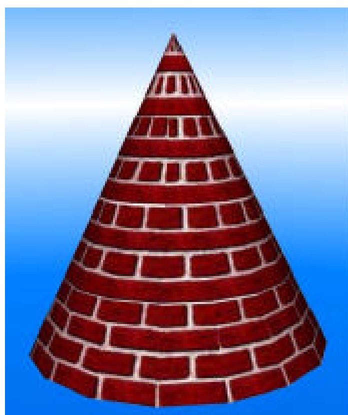


Figura 8 - Alterando Aplicação de Textura - textureTransform

3.4. Reutilizando Formas, Elementos e Definições

Quando se faz necessário reutilizar uma dada forma, são usados os termos DEF e USE. DEF define o nome de um dado elemento, forma ou definição. A reutilização deste nome a partir de sua chamada por USE dispensa a redefinição.

Como exemplo, será definido um conjunto de colunas, a partir da definição de uma única coluna. A coluna básica será definida por "Coluna" e terá aparência definida por "White". Ao longo da descrição do conjunto a reutilização se dará pela chamada destes termos com USE, como pode ser visto no código abaixo:

```
Group {
	children [
	# Coluna da Esquerda
		DEF ColEsquerda Transform {
			translation -2.0 3.0 0.0
			children DEF Coluna Shape {
				appearance DEF White Appearance {
					material Material { }
				}
				geometry Cylinder {
					radius 0.3
					height 6.0
				}
			}
		},
	# Coluna da Direita:
		DEF ColDireita Transform {
			translation 2.0 3.0 0.0
			children USE Coluna
		},
	# Coluna Superior
		DEF	Superior Transform {
```

```
                translation     0.0 6.0 0.0
                rotation 0.0 0.0 1.0 1.57
                children Shape{
                        appearance USE White
                        geometry  Cylinder {
                                radius 0.3
                                height 4.6
                        } }
        }

    ]}
```

O resultado é mostrado na Figura 9, onde um elemento foi reutilizado com uso de DEF e USE.

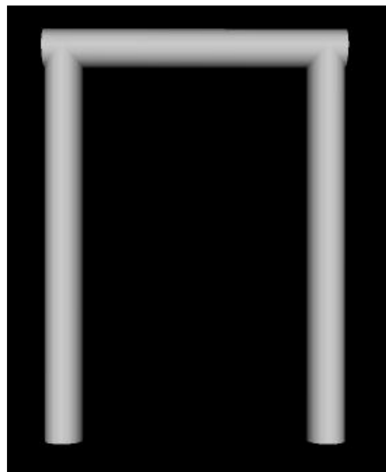


Figura 9 - Coluna com uso de DEF e USE

Podem ser inseridos elementos definidos em um dado arquivo dentro de um cenário 3D, com uso do nó *Inline*. A sintaxe de *Inline* é dada por:

```
Inline{
        url []
        bboxCenter -1.0 -1.0 -1.0
        bboxSize 0.0 0.0 0.0
}
```

No exemplo abaixo, o grupo de colunas é combinado, formando um conjunto de grupos de colunas, através da chamada feita pelo *Inline* combinada com uso de "DEF" e "USE". O resultado pode ser visto na Figura 10.

```
Group {
        children [
                Transform {
                        translation     0.0 -2.0 0.0
                        children DEF coluna Inline {url "colunas.wrl"}      }
                Transform {
                        translation     0.0 -2.0 -2.0
```

```
                children USE coluna
        }
        Transform {
                translation     0.0 -2.0 -4.0
                children USE coluna
        }
        Transform {
                translation     0.0 -2.0 -6.0
                children USE coluna
        }
]}
```

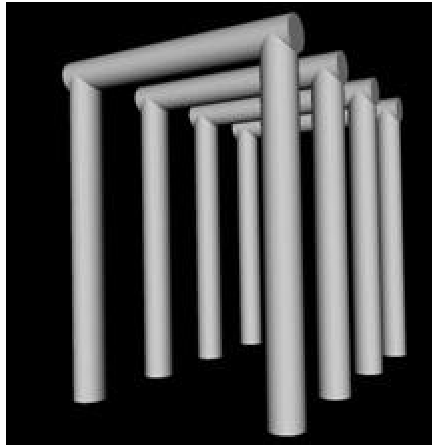


Figura 10 - Grupo de Colunas - uso de Inline

3.5. Compondo formas como conjunto de Faces

A utilização do nó *IndexedFaceSet* permite construir conjuntos de faces interligadas, de forma que tal combinação gere efeitos de formas complexas. Na descrição de um conjunto de faces, uma lista de pontos coordenados deve ser explicitada, além das conexões entre estes pontos. Supondo que se deseje a construção de um cubo seriam necessárias as coordenadas da parte superior do cubo e da parte inferior, além das descrições de ligações entre estes pontos.

```
geometry IndexedFaceSet {
   coord Coordinate {
      point [
      # Coordenadas da parte superior do cubo
         -1.0  1.0  1.0,
          1.0  1.0  1.0,
          1.0  1.0 -1.0,
         -1.0  1.0 -1.0,
       # Coordenadas da parte inferior do cubo
         -1.0 -1.0  1.0,
          1.0 -1.0  1.0,
          1.0 -1.0 -1.0,
```

```
            -1.0 -1.0 -1.0
         ]
      }
      coordIndex [
      # face superior
         0, 1, 2, 3, -1,
      # face de fundo
         7, 6, 5, 4, -1,
      # face de  frente
         0, 4, 5, 1, -1,
      # face da direita
         1, 5, 6, 2, -1,
      # face de costas
         2, 6, 7, 3, -1,
      # face da esquerda
         3, 7, 4, 0
      ]
   }
}
```

E o resultado pode ser visto na Figura 11, onde um Cubo foi construído a partir de suas faces.

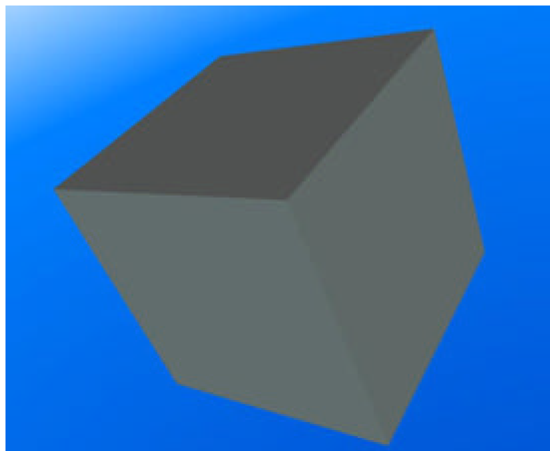


Figura 11 - Cubo - uso de IndexedFaceSet

3.6. Fundos e Cenários

O nó *Background* permite criar fundos para os mundos virtuais usando uma caixa de textura, elementos de piso e elementos do céu. A sintaxe (resumida) do nó *Background* é:

```
Background{
     skyColor [0.0 0.0 0.0]
     skyAngle []
     groundColor []
```

```
        groundAngle[]
        backUrl []
        bottomUrl []
        leftUrl []
        rightUrl []
}
```

O trecho de código a seguir, relativo à Figura 12, apresenta um cenário com um céu azulado e o chão em tom marrom:

```
Background {
        skyColor [
                0.0 0.2 0.7
                0.0 0.5 1.0
                1.0 1.0 1.0
        ]
        skyAngle [1.309 1.571]
        groundColor[
                0.1 0.1 0.1
                0.4 0.25 0.2
                0.6 0.6 0.6
        ]
        groundAngle   [1.309 1.571]
}
```

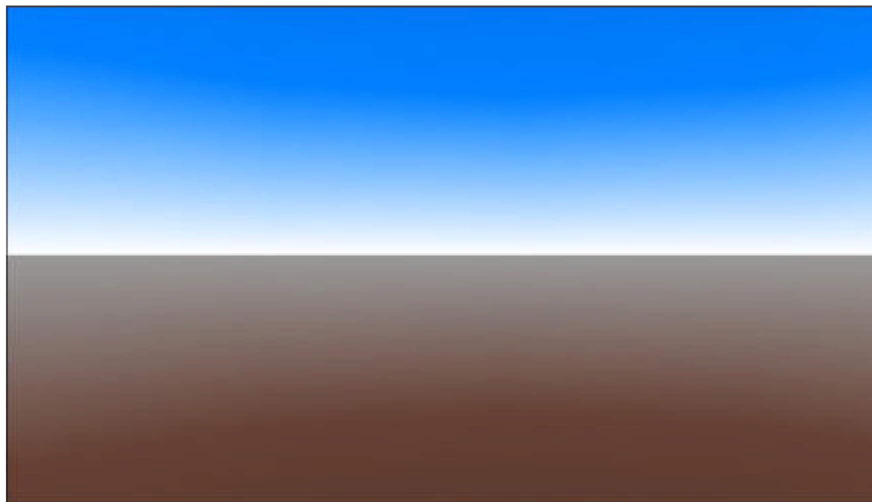


Figura 12 - Background aplicado

A definição de uma textura que é aplicada ao fundo pode ser feita como apresentado no trecho de código abaixo:

```
Background {
   skyColor [
      0.0 0.2 0.7,
      0.0 0.5 1.0,
      1.0 1.0 1.0
   ]
```

```
  skyAngle [ 1.309, 1.571 ]
  groundColor [
     0.1 0.10 0.0,
     0.4 0.25 0.2,
     0.6 0.60 0.6,
  ]
groundAngle [ 1.309, 1.571 ]
  frontUrl "montanha.gif"
  backUrl  "montanha.gif"
  leftUrl  "montanha.gif"
  rightUrl "montanha.gif"
}
```

E o resultado pode ser visto na Figura 13, onde um elemento de imagem auxilia na definição mais realística de um fundo.

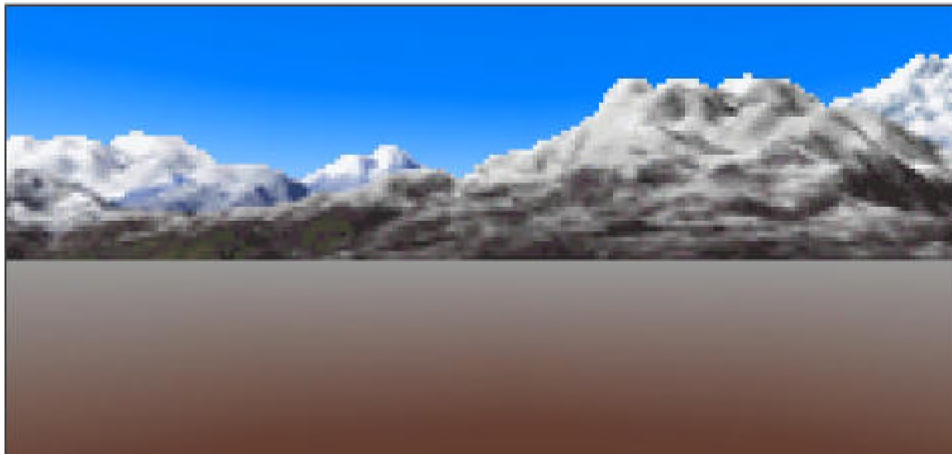


Figura 13 - Background com elementos de imagem ".gif"

3.7. Iluminação

VRML provê diversos nós para uma iluminação adequada de cenário. Todos estes nós apresentam os seguintes campos: *color*, *ambientIntensity* e *intensity*. As funções dos nós podem ser:

- iluminação do tipo direcional, com os raios emanando de forma radial em todas as direções: *PointLight;*
- iluminação do tipo direcional, com os raios pertencem a um pincel de luz paralela: *DirectionalLight*;
- iluminação do tipo 'spot', com os raios em um pincel na forma de cone, com a fonte de luz no ápice do cone: *SpotLight*;

Em todos os casos, a contribuição de uma fonte de luz é definida pelo campo *intensityColor*, enquanto que a contribuição da luz ambiente é dade em função do valor do campo *ambientIntensity*. Objetos com textura não são afetados pela fontes de luz. Pode-se desligar a luz na cabeça do usuário (navegador) através da definição a *headlight FALSE* no nó *NavigationInfo*, de acordo com a sintaxe:

```
NavigationInfo {
        headlight FALSE
}
```

3.7.1. DirectionalLight

A sintaxe do nó *DirectionalLight* é dada por:

```
DirectionalLight{
    on TRUE
    intensity 1.0
    ambientIntensity 0.0
    color 1.0 1.0 1.0
    direction 0.0 0.0 -1.0
}
```

O pincel de luz paralelo tem a orientação dada pelo campo *direction*, a cor definida no campo *color* e a intensidade definida em *intensity.* A Figura 14 apresenta um conjunto de esferas que foi iluminado com uma luz direcional, no sentido do eixo x.

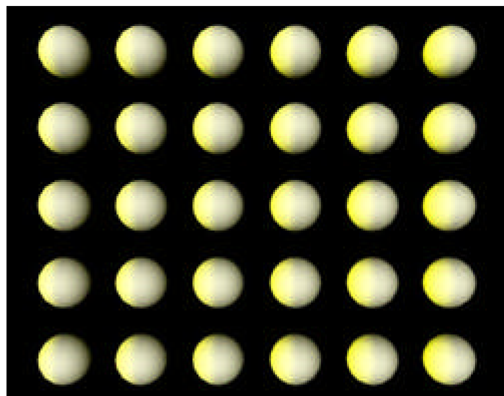


Figura 14 -uso de DirectionalLight

3.7.2. PointLight

A sintaxe do nó *PointLight* é dada por:

```
DirectionalLight{
    on TRUE
    location 0.0 0.0 0.0
    radius 100.0
    intensity 1.0
    ambientIntensity 0.0
    color 1.0 1.0 1.0
    attenuation 1.0 0.0 0.0
}
```

O pincel de luz puntiforme tem a localização no ambiente 3D dada pelo campo *locaion,* o raio da esfera de iluminação dado pelo campos *radius,* a intensidade definida em *intensity*. Os campos *intensity* e *color* combinam-se para definir o nível de cor provida pela fonte de luz em definição. Assim, a equação que define a cor da luz é:

lightColor = color X *intensity.*

A intensidade da luz ambiente é dada por:

lightAmbientColor = color X *intensity* X *ambientIntensity*.

As Figuras 15 e 16 apresentam um conjunto de esferas que foi iluminado com uma luz puntiforme, que está definida no centro do eixo de coordenadas do sistema. Na figura temos uma fonte de luz com grande intensidade e baixa atenuação (caso 1) e na figura temos o mesmo conjunto de esferas associado a uma fonte de luz com baixa intensidade e grande atenuação (caso 2).

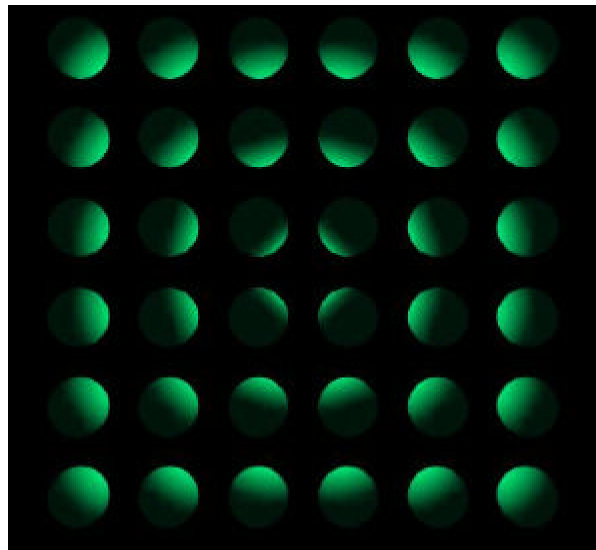


Figura 15 - uso de PointLight - caso 1

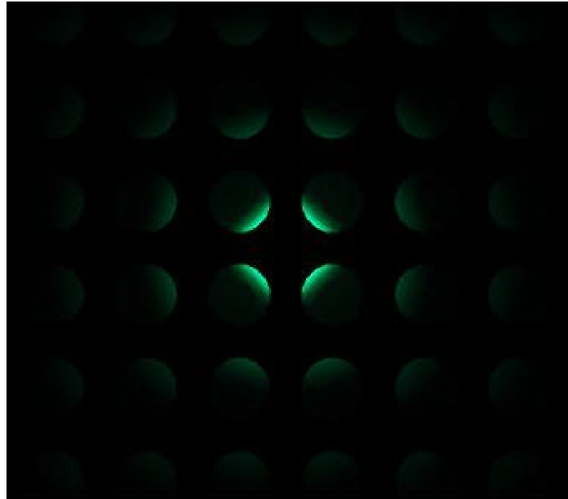


Figura 16 - uso de PointLight - caso 2

3.7.3. SpotLight

A sintaxe do nó *SpotLight* é dada por:

```
SpotLight{
        on TRUE
        location 0.0 0.0 0.0
        direction 0.0 0.0 -1.0
        radius 100.0
        intensity 1.0
        ambientIntensity 0.0
        color 1.0 1.0 1.0
        attenuation 1.0 0.0 0.0
        beamWidth 1.570789
        cutOffAngle 0.785398
}
```

As definições se assemelham às da *PointLight.* Os demais elementos podem ser vistos na Figura 17, onde apresenta-se, graficamente, os elementos que compõem o nó em questão.

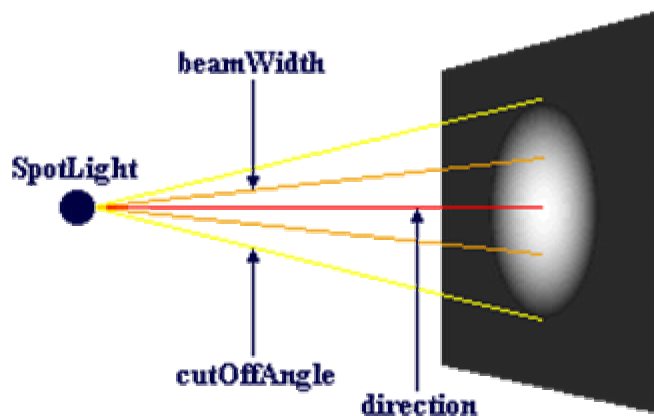


Figura 17 - Elementos de SpotLight

A Figura 18 apresenta o mesmo conjunto de esferas iluminado por uma *SpotLight*

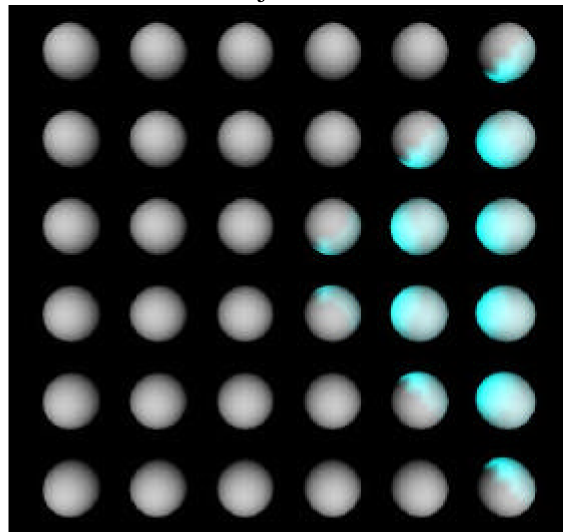


Figura 18 - uso de SpotLight

3.8. Controlando Detalhamento de Elementos - nó LOD

A técnica de controlar o detalhamento de elementos aparece na Computação Gráfica quando simuladores de vôo foram usados para treinamento de pilotos. Em tais simuladores, ao apresentar o cenário de um terreno, onde uma dada aeronave deveria pousar, o realismo é muito importante.

Para dar mais realismo à tais cenas, os terrenos necessitavam de grande detalhamento, associados à presença de edificações, árvores, carros etc. Para melhorar o desempenho de tais simuladores, observa-se que não é necessária a definição de muitos detalhes quando o elemento está situado a grande distância do observador e o inverso é válido para elementos muito próximos.

A técnica de controlar o detalhamento dos elementos está associada à criação de diferentes versões do mesmo que serão apresentadas ao navegante à partir de sua distância à forma em questão. O nó *LOD* ativa a chamada de cada forma em função de tais distâncias e sua sintaxe é dada por:

```
LOD{
        center 0.0 0.0 0.0
        level  []
        range []
}
```

O valor de *center* especifica uma posição no espaço 3D para o centro da forma construída com uso de *LOD*.

O campo *level* apresenta as diferentes ativações de formas que serão efetuadas para diferentes distâncias do usuário à forma, enquanto o campo *range* especifica as distâncias que definirão as ativações dos elementos de *range*. O código abaixo apresenta uma forma de controlar a apresentação de três formas - *box*, *sphere* e *cone* à medida que o observador aproxima-se ou afasta-se do conjunto.

```
LOD {
        center 0.0 0.0 0.0
        range [15,25]
        level [
        #primeiro nivel
                Group{
                children[
                Shape {  geometry  Box{}}
                Transform {
                translation 3 0 0
                children[
                        Shape {  geometry Sphere {  }}
                Transform{
                        translation 3 0 0
                        children Shape { geometry   Cone {}}}]}]}

        # Segundo Nivel
                Group {
                        children[
                        Shape {  geometry  Box {}}
                        Transform{
                        translation 3 0 0
                        children Shape { geometry  Sphere{}}}]}
        #Terceiro nível
                Shape {  geometry  Box{}}
                ]
                }
```

3.9. Animando as formas

3.9.1. Animações básicas

Para tornar o mundo mais dinâmico, podem ser animadas as posições, orientações e escalas das formas que estão definidas. Uma animação é uma mudança de algo em função de um intervalo de tempo. Uma animação requer dois elementos:

- Um elemento de tempo (relógio) que controla a animação;
- A descrição de como alguma coisa altera-se ao longo do tempo da animação.

A animação de algum elemento deve ser feita imaginando que o mesmo pode sofrer alterações de posição, orientação e escala em função de um dado tempo (fração do tempo em questão). O nó responsável pelo controle de tempos (e frações) é o nó *TimeSensor*. A sintaxe do nó *TimeSensor* é:

```
TimeSensor {
        enabled        TRUE
        startTime 0.0
        stopTime 0.0
        cycleInterval 1.0
        loop FALSE
}
```

A Figura 19 apresenta os elementos do nó *TimeSensor*. É importante destacar que o tempo é sempre uma fração do valor máximo (que é sempre 1.0). Assim, quando define-se que o intervalo de tempo (*cycleInterval*) será de 10.0 s, em 2 s teremos o tempo relativo à fração 0.2.

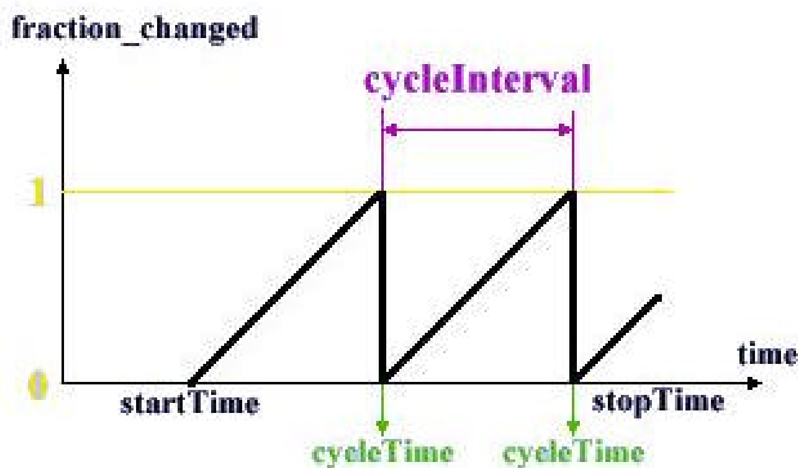


Figura 19 - Elementos do nó TimeSensor

O nó *TimeSensor* por si é incapaz de promover a animação. A associação do mesmo com nós que controlam posição, orientação e escala é essencial. O nó que controla a posição de uma forma em função de uma fração de tempo é o nó *PositionInterpolator*, cuja sintaxe é dada por:

```
PositionInterpolator {
        key     []
        keyValue []
}
```

O nó *PositionInterpolator* necessita de receber uma informação de tempo, enviada por um elemento de tempo (geralmente um *TimeSensor*). Com tal informação, compatibiliza a posição da forma (*keyValue*) com a fração de tempo (*key*). A associação é sempre feita por uma rota que combina a saída de tempo do *TimeSensor* com a entrada do nó *PositionInterpolator*.

Como exemplo, uma animação será aplicada à uma esfera que deve mudar de posição o tempo todo (descrevendo um S), em um loop, de forma que será desnecessária a intervenção do usuário para disparar a animação. Devem ser definidas as posições inicial e final do elemento, além das posições intermediárias (uma interpolação define o percurso do mesmo no cenário 3D). O trecho de código abaixo descreve esta animação:

```
DEF elemento Transform {
        children[
                DEF     cont_tempo TimeSensor {
                        startTime 0.0
                        loop TRUE
                        cycleInterval 5.0
                },
                DEF     cont_pos PositionInterpolator {
                        key     [ 0.0  0.2  0.4  0.6  0.8  1.0]
                        keyValue [
                                0.0  0.0  0.0
                                1.0  0.0  0.0
                                1.0  1.0  0.0
                                0.0  1.0  0.0
                                0.0  2.0  0.0
                                1.0  2.0  0.0
                        ]
                }
                DEF     esfera Shape{
                        appearance Appearance {
                                material Material {
                                        diffuseColor .46 .37 .14
                                        specularColor .38 .31 .12
                                        emissiveColor .14 .11 .04
                                        ambientIntensity 0
                                        shininess .04
                                }
                        }
                        geometry Sphere         {
                                radius 0.3}
                        }

        ]
```

```
}
ROUTE cont_tempo.fraction_changed  TO  cont_pos.set_fraction
ROUTE cont_pos.value_changed       TO  elemento.set_translation
```

Há necessidade de inserir rotas, de forma que valores obtidos em um nó sejam mapeados para outro nó, possibilitando a animação. Neste caso, os valores de frações de tempo produzidos pelo nó *cont_tempo* são enviados como entrada para o nó *cont_pos*. Com a combinação de tais frações com os valores chave de tempo descritos no campo *key* do nó *cont_pos* são associados com posições chave (*keyValue*), que devem ser enviadas par o nó de controle das tranformações geométricas (*elemento*), de forma que a esfera seja animada.

Se, ao invés de controlar posições, fosse necessário controlar a orientação, o nó adequado seria o nó *OrientationInterpolator*, que permite que sejam aplicadas rotações às formas desejadas.

A sintaxe do nó *OrientationInterpolator* é:

```
OrientationInterpolator        {
        key      []
        keyValue []
}
```

Onde o campo *key* define as frações de intervalos de tempo e o campo *keyValue* define as frações da animação (na forma de rotação). O trecho de código abaixo promove a rotação de uma caixa em torno do eixo 'x':

```
DEF elemento Transform {
        children[
                DEF   cont_tempo TimeSensor {
                        startTime 0.0
                        loop TRUE
                        cycleInterval 5.0
                },
                DEF   cont_or OrientationInterpolator  {
                        key      [ 0.0 0.25 0.5 0.75 1.0]
                        keyValue [
                                1.0  0.0  0.0    0.0
                                1.0  0.0  0.0    1.57
                                1.0  0.0  0.0    3.14159
                                1.0  0.0  0.0    4.7123
                                1.0  0.0  0.0    6.28
                                ]
                }
                DEF   forma Shape{
                        appearance Appearance {
                                material Material {
                                        diffuseColor .55 .09 .2
                                        specularColor .29 .31 .05
                                        emissiveColor .21 .03 .08
                                        ambientIntensity .03
                                        shininess .19
                                }
```

```
                    }
                    geometry Box{
                            size 0.3 0.5 0.8}
                    }

        ]
}
ROUTE cont_tempo.fraction_changed   TO   cont_or.set_fraction
ROUTE cont_or.value_changed         TO   elemento.set_rotation
```

A definição da rotação está nos valores de *keyValue*, onde define-se que será feita sobre o eixo 'x', que é o único dos três eixos que está com valor diferente de '0.0' (como pode ser visto na linha 1 do campo *keyValue*. A rotação será de 0 a 2π e deve ser definida com ângulos em radianos.

A última forma de animação relativa às transformações geométricas é a mudança de escala. O nó que permite mudança de escala é o nó *PositionInterpolator* (o mesmo que permite a animação de posição). Neste caso, no entanto, devem ser colocados valores de escala no campo *keyValue* e a rota de saída de valores deve ser entrada para valores de escala ao invés de valores de posição. Supondo a esfera apresentada anteriormente, a modificação do nó de controle e das rotas seria dada por:

```
DEF   cont_esc PositionInterpolator {
                    key     [ 0.0 0.2 0.4 0.6 0.8 1.0]
                    keyValue [
                            1.0 1.0 1.0
                            1.5 1.5 1.5
                            2.0 2.0 2.0
                            2.5 2.5 2.5
                            1.5 1.5 1.5
                            1.0 1.0 1.0
                    ]
            }
            DEF   esfera Shape{
                    appearance Appearance {
                            material Material {
                                    diffuseColor .46 .37 .14
                                    specularColor .38 .31 .12
                                    emissiveColor .14 .11 .04
                                    ambientIntensity 0
                                    shininess .04
                            }
                    }
                    geometry Sphere      {
                            radius 0.5}
                    }
        ]
}
ROUTE cont_tempo.fraction_changed  TO  cont_esc.set_fraction
ROUTE cont_esc.value_changed       TO  elemento.set_scale
```

3.9.2. Controlando as animações

De maneira a controlar as animações, podem ser inseridos nós sensores de ações dos usuários. Há três formas de ações que podem ser percebidas:

- Movimento (*Move*): sem pressionar o mouse, o usuário move o cursor sobre um item qualquer;
- Clique: quando o cursor está sobre um elemento, o botão do mouse é pressionado e liberado, sem movimento do mesmo;
- Arraste (*Drag*): com o cursor sobre um item, o botão do mouse é pressionado e, mantido pressionado, o mouse é arrastado.

O nó capaz relativo à captura de toque é o nó *TouchSensor*. A sintaxe do nó *TouchSensor* é:

```
TouchSensor {
        enabled        TRUE
}
```

O nó *TouchSensor* tem sensibilidade ao toque (campo *touchTime*), ao cursor estar sobre uma forma (*isOver*) e pressionado (*isActive*). Supondo que se desejasse disparar a animação de uma esfera como mostrado anteriormente, ao toque na esfera, a sintaxe poderia ser:

```
DEF elemento Transform {
        children[
                DEF     cont_tempo TimeSensor {
                        cycleInterval 5.0
                },
                DEF     cont_pos PositionInterpolator {
                        key     [ 0.0 0.2 0.4 0.6 0.8 1.0]
                        keyValue [
                                0.0 0.0 0.0
                                1.0 0.0 0.0
                                1.0 1.0 0.0
                                0.0 1.0 0.0
                                0.0 2.0 0.0
                                1.0 2.0 0.0
                        ]
                }
                DEF     toque TouchSensor { }
                DEF     esfera Shape{
                        appearance Appearance {
                                material Material {
                                        diffuseColor .46 .37 .14
                                        specularColor .38 .31 .12
                                        emissiveColor .14 .11 .04
                                        ambientIntensity 0
                                        shininess .04
                                }
                        }
```

```
geometry Sphere {
        radius 0.3}
}
    ]
}
ROUTE toque.touchTime                 TO  cont_tempo.startTime
ROUTE cont_tempo.fraction_changed  TO  cont_pos.set_fraction
ROUTE cont_pos.value_changed         TO  elemento.set_translation
```

Neste exemplo, quando o usuário toca na esfera, que está definida no mesmo grupo do nó 'toque', um evento de disparo de tempo é enviado para o nó de controle de tempo - 'cont_tempo'. Só então, frações de tempo são recebidas pelo nó 'cont_pos', disparando a animação (que acontece uma única vez).

De maneira a permitir que um dado elemento gráfico seja manipulado pelos movimentos do usuário, existem três nós distintos;

- *PlaneSensor*: converte as ações do usuário em movimentos em um plano 2D;
- *SphereSensor*: converte as ações do usuário em movimentos ao longo de uma esfera que envolve a forma;
- *CylinderSensor*: converte as ações do usuário em movimentos ao longo de um cilindro definido sobre um dos eixos.

Sintaxe do nó PlaneSensor:

```
PlaneSensor   {
    enabled         TRUE
    autoOffset     TRUE
    offset           0.0 0.0 0.0
    maxPosition  -1.0 -1.0
    minPosition   0.0 0.0
}
```

Os campos maxPosition e minPosition definem os limites da translação que será aplicada à forma. O trecho de código abaixo permite que, com a movimentação do mouse, uma dada esfera sofra mudança de posição ao longo do plano 'XY':

```
DEF elemento Transform {
    children[
        DEF sensor_pos PlaneSensor{
            enabled        TRUE
            autoOffset TRUE
            offset 0.0 0.0 0.0
            maxPosition   -1.0 -1.0
            minPosition   0.0 0.0
            }
        DEF   esfera Shape{
            appearance Appearance {
                material Material {
                    diffuseColor .62 0 .62
                    specularColor .5 0 .5
                    emissiveColor .15 0 .15
                    ambientIntensity 0
```

```
                    shininess .15
                }
            }
            geometry Sphere     {
                radius 0.8}
            }
    ]
}
ROUTE  sensor_pos.translation_changed TO elemento.translation
```

O nó SphereSensor tem sintaxe dada por:

```
SphereSensor {
    autoOffset TRUE
    enabled TRUE
    offset 0.0 1.0 0.0      0.0
}
```

O campo offset permite definir o eixo de rotação que será usado. O trecho de código abaixo apresenta a utilização de SphereSensor para rotacionar uma dada caixa (box):

```
DEF elemento Transform {
    children[
        DEF sensor_esf SphereSensor {
            autoOffset TRUE
            enabled TRUE
            offset 0.0 1.0 0.0      0.0
        }
        DEF   esfera Shape{
            appearance Appearance {
                material Material {
                    diffuseColor .6 .23 0
                    specularColor .5 .2 0
                    emissiveColor .3 .12 0
                    ambientIntensity 0
                    shininess .15
                }
            }
            geometry Box {
                size 2.0 3.0 5.0}
            }

    ]
}
ROUTE  sensor_esf.rotation_changed     TO     elemento.rotation
```

O nó CylinderSensor tem sintaxe dada por:

```
CylinderSensor {
    autoOffset TRUE
    enabled         TRUE
```

```
        diskAngle 0.262
        offset 0.0
        maxAngle -1.0
        minAngle 0.0
    }
```

O trecho de código abaixo apresenta a utilização de CylinderSensor para movimentar uma dada caixa (box):

```
DEF elemento Transform {
    children[
        DEF sensor_esf CylinderSensor {
                autoOffset TRUE
                enabled     TRUE
                diskAngle 0.262
                offset 0.0
                maxAngle -1.0
                minAngle 0.0
            }
        DEF   esfera Shape{
            appearance Appearance {
                material Material {
                    diffuseColor .6 .23 0
                    specularColor .5 .2 0
                    emissiveColor .3 .12 0
                    ambientIntensity 0
                    shininess .15
                }
            }
            geometry Box{
                size 2.0 3.0 5.0}
            }

    ]
}
ROUTE  sensor_esf.rotation_changed    TO    elemento.rotation
```

3.10. Controlando o Ponto de Vista e a Navegação

A navegação pode ser controlada pelo nó *NavigatonInfo*. A sintaxe do nó *NavigationInfo* é dada por;

```
NavigationInfo {
    speed 1.0
    type "WALK"
    avatarSize [0.25 1.6 0.75]
    headlight TRUE
    visibilityLimit 0.0
}
```

A velocidade de movimentação do *avatar* é dada pelo campo *speed*. Os tipos de navegação, relativos ao campo *type* são:

- "WALK": quando o usuário caminha no mundo e é afetado pela gravidade;
- "FLY": quando o usuário pode se mover sem ser afetado pela gravidade;
- "EXAMINE": quando o usuário fica estático, mas pode se mover ao redor do mundo em diversos ângulos;
- "NONE", quando o usuário não pode controlar os movimentos.

O campo *headlight* define se o *avatar* terá ou não uma fonte de luz na sua cabeça. Se TRUE, uma fonte de luz estará ligada na cabeça do *avatar*.

Um ponto de vista (*Viewpoint*) é uma posição pré-definida com uma dada orientação no cenário. Podem ser alteradas as formas de navegação do usuário e suas posições de visualização.

A sintaxe do nó Viewpoint é:

```
Viewpoint {
        description     ""
        jump TRUE
        orientation     0.0 1.0 0.0 0.0
        position 0.0 0.0 1.0
        fieldOfView  0.7854
}
```

O campo *orientation* define um eixo de rotação e um ângulo de rotação. O campo *position* define a posição de visualização do *avatar* em relação ao cenário apresentado. O trecho de código abaixo mostra como pode ser mudada a forma de navegação a partir do toque em uma dada forma:

```
Group {
                children [
                                DEF nav NavigationInfo {
                                type "NONE"
                                speed 2.0
                                headlight FALSE
                                avatarSize [0.5 1.6 0.5]}
                                DEF nav2 NavigationInfo {
                                        type "WALK"}
                        DEF toque TouchSensor       {},
                        Shape{
                                geometry Cone {}
                        }
                        ]
                        }
        ROUTE toque.isOver TO nav2.set_bind
```

3.11. Adicionando Sons e Filmes ao Cenário

De maneira a permitir que o cenário seja mais realístico, podem ser adicionados sons, na forma de sons ambientes ou de sons controlados por animações. Os nós relativos a adição de sons são:

- *AudioClip*: nó que suporta alguns tipos de sons digitais (não é permitido o mp3): MIDI e WAV;
- *Sound*: nó que cria um emissor de som que pode ser ouvido dentro de uma região elipsoidal;
- *MovieTexture*: permite a inserção de filmes (movie).

A sintaxe do nó *AudioClip* é:

```
AudioClip {
	startTime 0.0
	stopTime 0.0
# se loop = TRUE, o som não pára - som ambiente
	loop TRUE
	description	""
# relativo ao arquivo que será executado:
	url	[]
}
```

A sintaxe do nó *Sound* é:

```
Sound{
	source NULL
	intensity 1.0
# localização do emissor de som:
	location 0.0 0.0 0.0
# direção do som emitido
	direction 0.0 0.0 1.0
	minFront 1.0
	maxFront 10.0
	minBack	1.0
	maxBack	10.0
	priority 0.0
# Controle de som espacial:
	spatialize TRUE
}
```

Os elementos do nó *Sound* podem ser vistos na Figura 20.

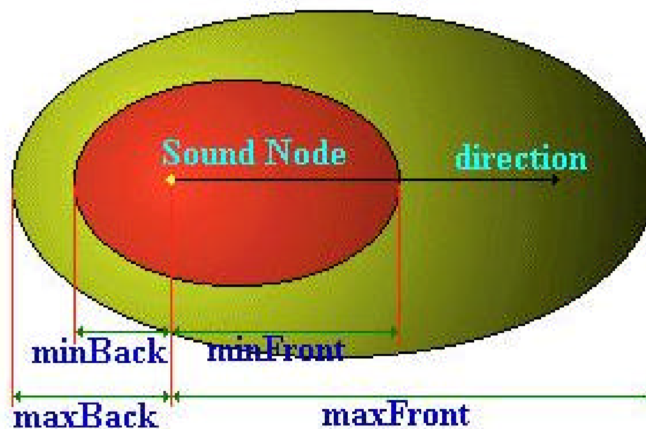


Figura 20 - Elementos do nó Sound

A sintaxe do nó *MovieTexture* é:

```
MovieTexture {
	startTime 0.0
	stopTime 0.0
	speed 1.0
	loop FALSE
	repeatS		TRUE
	repeatT		TRUE
# Relativo ao filme que será carregado:
	url	[]
}
```

3.12. Sentindo a proximidade do usuário

Como visto, pode se usar o nó *TouchSensor* para detectar quando o usuário toca uma forma do cenário. Há, no entanto, três outros nós capazes de identificar ações do usuário, que podem ser utilizados para controlar animações, da mesma forma que o nó *TouchSensor*:

- *VisibilitySensor*: usado para identificar a visibilidade de um observador, através de uma região que tem o formato de uma caixa (região de visibilidade);
- *ProximitySensor*: nó que permite detectar quando o observador entra e/ou se move em uma região fechada (em formato de uma caixa);
- *Collision*: permite detectar quando o usuário colide com alguma forma, sendo também um nó de agrupamento (como *Group* ou *Transform*). Este nó gera o tempo absoluto em que o choque ocorreu ou alerta o navegador o fato;

Sintaxe do nó *VisibilitySensor*:

```
VisibilitySensor {
	enabled		TRUE
# Centro:
	center 0.0 0.0 0.0
# tamanho da região em termos de X, Y e Z:
	size 0.0 0.0 0.0
}
```

Sintaxe do nó *ProximitySensor*:

```
ProximitySensor{
	enabled		TRUE
	center 0.0 0.0 0.0
	size 0.0 0.0 0.0
}
```

Sintaxe do nó *Collision*:

```
Collision {
# Descrição dos nós que serão controlados:
	children []
```

```
        bboxCenter 0.0 0.0 0.0
    # uma caixa que contém todas as formas pertencentes ao nó:
        bboxSize -1.0 -1.0 -1.0
    # controle de habilitação da detecção de colisão:
        collide TRUE
        proxy NULL
    }
```

O trecho de código abaixo apresenta uma animação controlada pelo nó *ProximitySensor*. Neste exemplo, quando o usuário se aproxima de uma caixa de texto, um som intermitente é emitido, como um som de alerta:

```
DEF SENSOR ProximitySensor {
  size 8 8 8
}

Sound {
  minFront 10
  minBack 10
  maxFront 50
  maxBack 50
  source DEF SOUND AudioClip {
    loop TRUE
    url "pop.wav"
    startTime -1
  }
}

Group {
        children [
                        Shape {
                          appearance Appearance {
                         material Material {
                        emissiveColor .1 .1 0
                                        diffuseColor .75 .75 0
                                        specularColor .86 .86 .86
                                        ambientIntensity .127
                                        shininess .38
                         }
                          }
                geometry Box { size 0.8 0.5 0.1}
        }
        Transform {
                translation     0.0 0.0 0.12
                children
                Shape {
                        appearance Appearance {
                                material Material {
                                        diffuseColor .9 .05 .5
```

```
                    specularColor .1 .1 .1
                    emissiveColor .1 0 .05
                    ambientIntensity .12
                    shininess .08
                }
            }
            geometry Text {
                string "Alerta"
                fontStyle FontStyle    {
                    size 0.27
                    justify  "MIDDLE"
                }
            }
        }
    }
]}

ROUTE SENSOR.enterTime TO SOUND.startTime
ROUTE SENSOR.exitTime TO SOUND.stopTime
```

No exemplo anterior, se ao invés de utilizar o nó ProximitySensor, fosse utilizado o nó Collision, o som só seria emitido no caso de colisão com a forma.

3.13. Unindo Cenários Virtuais - Links

Usando VRML é possível simular que, ao toque em uma porta, um novo cenário será carregado. Tal ativação pode carregar um novo cenário 3D, uma página Web ou uma animação. O nó adequado à tais simulações é o nó *Anchor*. A sintaxe do nó *Anchor* é:

```
Anchor {
    children []
    bboxCenter 0.0 0.0 0.0
    bboxSize -1.0 -1.0 -1.0
    url      []
    parameter []
}
```

É importante destacar ocampo *url* que conterá a informação da página com a qual se deseja ativar ao tocar os elementos descritos em *children*. O trecho de código abaixo apresenta um exemplo de utilização de *Anchor*:

```
Anchor {
    # inserção do link para onde se deseja ir:
            url [ "http://www.sbc.org.br/svr"]
            description     "Página do SVR´03"
    # nós que ativarão o link:
            children [
        Transform {
            translation 0.0 0.0 0.0
            children [ Shape {
```

```
                        appearance Appearance {
                                material Material {
                                        diffuseColor .62 .31 .62
                                        specularColor .5 .25 .5
                                        emissiveColor .15 .07 .15
                                        ambientIntensity 0
                                        shininess .15
                                }
                        }
                        geometry DEF Door Box {size 2.0 1.0 0.1}

                }
                Transform {
                        translation      0.0 0.0 0.2
                        children Shape {
                                appearance Appearance {
                                        material Material {
                                                diffuseColor .68 .68 .43
                                                specularColor .5 .5 .31
                                                emissiveColor .3 .3 .19
                                                ambientIntensity 0
                                                shininess .14
                                        }
                                }

        # inserção de um texto para identificar o link:
                                geometry Text {
                                        string "SVR 03"
                                        fontStyle FontStyle    {
                                                size 0.4
                                                justify "MIDDLE"}
                                        }
                                }
                        }
                ]
        },]}

Anchor {
    # inserção do link para onde se deseja ir:
                url [ "http://www.compgraf.ufu.br/alexandre"]
                description      "Homepage do Autor"
    # nós que ativarão o link:
                children [
        Transform {
                translation 2.0 0.0 0.0
                children [ Shape {
                        appearance Appearance {
```

```
                    material Material {
                        diffuseColor .03 .03 .18
                        specularColor .18 .27 .69
                        emissiveColor .03 .03 .17
                        ambientIntensity .0367
                        shininess .03
                    }
                }
                geometry USE Door

            }
            Transform {
                translation     0.0 0.0 0.2
                children Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor .6 .55 .24
                            specularColor .5 .46 .2
                            emissiveColor .3 .27 .12
                            ambientIntensity 0
                            shininess .14
                        }
                    }
        # inserção de um texto para identificar o link:
                    geometry Text {
                        string "Autor"
                        fontStyle FontStyle    {
                            size 0.4
                            justify "MIDDLE"}
                        }
                    }
                }
            ]
        }
    ]
}
```

4. Combinando VRML e JavaScript

4.1. Introdução

Os nós já apresentados para animação, tais como *TouchSensor*, *TimeSensor*, *PlaneSensor* etc são muito limitados quando se desejam animações mais complexas ou controle de elementos de cena a partir de equações (matemáticas, físicas, químicas etc). Em

tais situações é imperativa a necessidade de uso dos nós *Script*, que associam VRML com JavaScript ou Java.

Um nó Script pode ser entendido como uma forma particular de controle ou de sensor. Como qualquer nó de controle ou de sensor, este nó requer uma lista de campos (*field*) , eventos de entrada (*eventIns*) e eventos de saída (*eventOuts*). A descrição do nó deve complementar a definição destes campos, dando a eles uma dada finalidade . Um nó *Script* deve ser entendido como um nó que recebe informações de outros nós através dos eventos de entrada, processa-as e envia informações na forma de eventos de saída.

Logicamente, não serão encontrados nós *Script* independentes da descrição de rotas, que mapeiam a troca de informações entre nós externos e o nó *Script*. Uma exigência importante é a necessidade de equivalência de tipos de elementos entre os nós que trocam informações.

Em VRML, temos diferentes tipos de elementos, como exemplo:

- SFBool - para valores booleanos
- SFInt32 - para valores inteiros
- MFInt32 - para uma lista de valores inteiros
- SFFLoat - para valores reais
- MFFloat - para uma lista de valores reais
- SFNode - para a descrição de um nó
- SFColor - para a descrição de RGB de uma dada cor
- Etc

Característica básica de um *Script*:

```
Script {
# definição de campos, suas naturezas e valores iniciais:
        field SFBool booleano1 TRUE
# definição de eventos de entrada e sua natureza:
        eventIn         SFBool entrada
# definição de eventos de saída e sua natureza:
        eventOut SFBool      saida
        url      "javascript:
        function entrada(param){
// processamento e determinação da saída
                saida = param;
        "
}
```

4.2. Desenvolvendo Exemplos

Como exemplo, a função abaixo representa um Script capaz de inserir um novo elemento, no caso, uma esfera em uma área de um mundo virtual:

```
DEF Criador Script {
  eventIn SFTime sphere_touchTime
  field SFNode nopai USE TOP
  field SFFloat x 0.0
  eventOut MFNode new_child
  url "javascript:
```

```
function sphere_touchTime(value,time) {
     newVRML =  'Group {';
     newVRML += '   children [';
    newVRML += '      DEF SENSOR PlaneSensor {';
     newVRML += '         maxPosition 0.45 0.45';
     newVRML += '        minPosition  -0.45 -0.45'
     newVRML += '      }';
     newVRML += '      DEF OBJECT Transform {';
     newVRML +=                   'translation '
     newVRML +=                     x;
     newVRML +=           ' 0.0 ';
     newVRML +=           ' 0.0';
     newVRML += '        children [';
     newVRML += '          Shape {';
     newVRML += '             appearance Appearance {';
     newVRML += '                material Material {';
     newVRML += '                  diffuseColor 0 1  1';
     newVRML += '                }';
     newVRML += '             }';
     newVRML += '             geometry Sphere {';
     newVRML += '               radius 0.05';
     newVRML += '             }';
     newVRML += '          }';
     newVRML += '        ]';
     newVRML += '      }';
     newVRML += '   ]';
     newVRML += '}';
    newVRML+=' ROUTE SENSOR.translation_changed TO
OBJECT.set_translation';
     node = Browser.createVrmlFromString(newVRML);
    new_child = node;
    nopai.addChildren = new_child;
}
"
}
ROUTE SPHERESENSOR.touchTime TO Criador.sphere_touchTime
```

É possível alterar dinamicamente um texto, que funcione como um contador de esferas adicionadas (continuação do exemplo anterior), se um dado nó de texto for atualizado, a partir de um campo que conte quantas esferas foram adicionadas. O trecho de código abaixo refere-se à função sphere_touchTime atualizada, de tal forma que a mesma é capaz de modificar o texto, enviando, através de uma rota, o valor atual de esferas inseridas:

```
field SFFloat x2 0.0
field MFString str_sphere "0"
eventOut       MFString str_sphere_out
```

```
function sphere_touchTime(value,time) {
        x2 = x2 + 1.0;
        str_sphere[0] = String(x2);
        str_sphere_out[0] = str_sphere[0];
```

A rota combinada seria do tipo:

ROUTE Criador.str_sphere_out TO textoS.string

Onde textoS é um campo de texto pertencente a um nó Shape.

5. Combinando VRML e Java - dicas

A combinação de VRML, *Java* e HTML pode ser efetuada de seis diferentes maneiras, de forma a facilitar a utilização e tirar máximo proveito desta possibilidade. A lista abaixo é um sumário das hipóteses:

1. código VRML inserido em um arquivo HTML: usando caracteres de formatação do tipo <EMBED> ou <OBJECT>, códigos de VRML são inseridos em arquivos HTML;
2. código *Java* inserido em arquivo VRML: este é o padrão para VRML 2.0, um nó do tipo *Script* refere-se a um código *Java* pré-compilado;
3. *applet Java* comunicando com o navegador VRML: esta versão - não padrão para VRML 2.0 - constitui uma extensão e é conhecida como *External Authoring Interface (EAI)* e tende a se tornar o padrão;
4. classes *Java* correspondentes a nós VRML: algumas companhias estão desenvolvendo ferramentas de programação que definem representações de nós VRML que podem ser usadas por programadores quando estão escrevendo
5. código em *Java*. Esta hipótese é mais usual para implementar navegadores ou ferramentas VRML, mas, não constitui padrão para VRML ou *Java*;
6. HTML dentro do código VRML: usando um arquivo HTML como um mapa de textura para apresentar dentro de um mundo 3D, podemos ter uma interessante extensão para VRML, mas, esta proposição ainda não é suportada por qualquer navegador VRML e não é parte do VRML 2.0, sendo uma proposta dos criadores em discussão;
7. *applet Java* dentro de um arquivo VRML: usando um *applet Java* como um mapa de textura temos outra interessante possibilidade de extensão, também em discussão e ainda não implementada.

Por permitir maior flexibilidade e constituir uma forma mais usual de desenvolvimento, este trabalho opta pela forma de comunicação estabelecida no item 2 acima, na qual um *applet Java* se comunica com o navegador VRML.

6. Referências Bibliográficas

AMES, L. A.; NADEAU, R.D.; MORELAND D. **VRML Sourcebook - Second Edition**, John Wisley & Sons, Inc - USA, 1997.

CARDOSO, A.; LAMOUNIER E.; TORI R. Sistema de Criação de Experiências de Física em Realidade Virtual para Educação a Distância. In: II WORKSHOP BRASILEIRO DE REALIDADE VIRTUAL, **WRV´99**, Marília , São Paulo, Brasil, 1999, p. 174-181.

CARDOSO, A. Alexandre Cardoso - Página do Pesquisador. Contém informações sobre aplicações de Realidade Virtual, pesquisa e publicações do pesquisador, tutoriais sobre VRML e artigos indicados para leitura. Disponível em: <http://www.compgraf.ufu. br/alexandre/>. Acesso em Set./2003

CHEN, S.; MYERS, R.; PASSETO, R. The Out of Box Experience - Lessons Learned Creating Compelling VRML 2.0 Content. In: Proceedings of the Second Symposium on Virtual Reality Modeling Language, p. 83-92, 1997.

ELLIS R. S. What are Virtual Environments**. IEEE Computer Graphics and Applications**, p. 17-22, Jan. 1994.

LEMAY, MURDOCK, COUCH **3D Graphics and VRML 2** Sams.Net, Indiana - USA - 1996

MATSUBA, N.; STEPHEN; ROEHL, B. Bottom, Thou Art Translated: The Making of VRML Dream. **IEEE Computer Graphics and Applications**, v. 19, n. 2, p. 45-51, Mar./Apr. 1999.

Vapor Tutorial VRML - Tutorial de VRML com exemplos e fontes. Disponível em http://web3d.vapourtech.com/. Acesso em Set./2003

VRMLPAD. Parallellgraphis. O sítio disponibiliza diversos programas computacionais de grande utilidade para desenvolvimento de ambientes virtuais em VRML. Disponível em: <http://www.parallellgraphics.com>. Acesso em: 02 nov. 2000.