UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DE DEFORMAÇÃO DE MALHAS NO HUMANÓIDE V-ART

KARLYSON SCHUBERT VARGAS

BLUMENAU
2008

2008/1-21

KARLYSON SCHUBERT VARGAS

IMPLEMENTAÇÃO DE DEFORMAÇÃO DE MALHAS NO HUMANÓIDE V-ART

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Dalton Solano dos Reis , M.Sc. - Orientador

BLUMENAU
2008

2008/1-21

IMPLEMENTAÇÃO DE DEFORMAÇÃO DE MALHAS NO HUMANÓIDE V-ART

Por

KARLYSON SCHUBERT VARGAS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Paulo César Rodacki Gomes, Dr. – FURB
Membro:	Prof. Antônio Carlos Tavares, M.Sc. – FURB

Blumenau, 09 de julho de 2008

Dedico este trabalho a todas as pessoas que me auxiliaram na minha caminhada na faculdade.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que sempre esteve presente ao meu lado, me ajudando nos momentos difíceis.

Aos meus amigos, por toda a ajuda que me deram durante toda a faculdade.

Ao meu orientador, Dalton, por ter acreditado e auxiliado para a conclusão deste trabalho.

Ao Rodrigo pelo enorme auxílio na criação do *abstract*.

A provação vem, não só para testar o nosso valor, mas para aumentá-lo; o carvalho não é apenas testado, mas enrijecido pelas tempestades.

Lettie Cowman

RESUMO

O presente trabalho tem por finalidade descrever o desenvolvimento de um algoritmo de deformação de malhas no *framework* V-Art, permitindo uma maior aproximação das animações do seu humanóide para com um humano real. A aplicação é especificada com orientação a objetos, usando UML. Neste trabalho são explicados os dois tipos de algoritmos de deformação de malha, os de deformação poligonal e os de deformação paramétricos, sendo que foi implementado o algoritmo paramétrico FFD. Também é explicado sobre o que são e para o que servem os humanóides. É explicado sobre o funcionamento do *framework* V-Art. O resultado alcançado foi a implementação do algoritmo FFD e a deformação do humanóide, com restrições devido ao modelo do humanóide do V-Art.

Palavras-chave: Deformação de malhas. Humanóides. V-Art.

ABSTRACT

The present work aims to describe the development of a mesh deformation algorithm in the V-Art framework, providing a bigger similarity of your humanoid animations with a real human being. The application is specified with object orientation, using UML. In this work, two types of mesh deformation algorithms are explained, the polygonal deformation and the parametric deformation type, but the one which was implemented is the FFD parametric algorithm. What are humanoids and what is their use is also explained. There is also an explanation about the V-Art framework functioning. The result was the FFD algorithm implementation and the humanoid deformation, with some restrictions because of the V-Art's humanoid model.

Key-words: Mesh deformation. Humanoid. V-Art.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

FFD – *Free Form Deformation* (Deformação de formas livres)

DOFs – *Degrees Of Freedom* (Graus de liberdade)

XML - *eXtensible Markup Language*

STL - *Standard Template Library*

UML - *Unified Modeling Language*

FPS - *Frames Per Second* (Quadros por Segundo)

DTD – *Document Type Definition*

SUMÁRIO

1 INTRODUÇÃO

Humanóides são representações articuladas 3D de personagens animados. Atualmente os humanóides estão presentes no cotidiano, sendo observados em filmes, jogos e telejornais, auxiliando assim diversas áreas, como na psicologia, na educação física e na educação. Na maioria das vezes é necessário que o espectador "veja" os movimentos realizados pelos humanóides como sendo reais, aparentando para o mesmo que os humanóides são realmente seres humanos. Apesar dos humanóides terem a intenção de representar um humano real, em um conceito geral ele não se limita a ter a mesma estrutura de um humano, podendo ter diferentes quantidades de membros, cabeças, ou outras partes de um corpo normal (H | ANIM..., 2004).

Realizar animações baseadas em gestos humanos é uma atividade muito complexa, visto que o corpo humano possui uma estrutura complexa geralmente representado por um objeto articulado em um programa de computador. Esse objeto possui também uma grande quantidade de juntas, aproximadamente 200. Dentre programas que criam ou manipulam humanóides será destacado o V-Art.

O V-Art é um *framework* desenvolvido em C++ para facilitar a criação de programas com ambientes 3D, principalmente os que utilizam-se de humanóides. Dentre os seus principais diferenciais com relação a outros programas, destacam-se o fato deste *framework* ser totalmente orientado a objetos, possuir um sistema de suporte a animações e possuir a representação das articulações humanas biologicamente corretas (NEDEL; FREITAS, 2007). Nesse *framework* o usuário pode interagir em tempo real ou definir *scripts* para alterar a movimentação das articulações que podem resultar em animações que representam ações (caminhar, sentar, etc.). Atualmente apenas as articulações do humanóide podem ser movimentadas, mas em um humano real durante uma ação ocorre uma deformação na estrutura do corpo, como por exemplo, ao se dobrar o joelho a pele na parte de trás do joelho é contraída enquanto a parte da frente é esticada.

Com este intuito, acrescentou-se a esse framework a implementação de um algoritmo de deformação de malhas.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é adicionar a funcionalidade de deformação de malhas no *framework* V-Art.

Os objetivos específicos do trabalho são:

a) definir um algoritmo de deformação de malhas;
b) implementar o algoritmo selecionado no *framework* V-Art;
c) associar pontos de controle da deformação da malha as articulações.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em quatro capítulos. No capítulo seguinte é descrita a fundamentação teórica utilizada para embasar este trabalho. É apresentada na seção 2.1 uma explicação sobre algoritmos de deformação de malha, assim como os tipos de algoritmos que existem, na seção seguinte (2.2), é explicado o que são e para que servem os humanóides. Na seção 2.3 é explicado o que é, para que serve e como funciona o *framework* V-Art, assim como um programa que utiliza-se dele, o Action Builder. O capítulo é finalizado com os trabalhos correlatos.

O capítulo 3 traz a especificação e implementação da ferramenta. Ao final do capítulo são apresentados os resultados alcançados a partir dos testes realizados.

O capítulo 4 contém a conclusão do trabalho, juntamente com sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados alguns aspectos teóricos relacionados ao trabalho, tais como: algoritmos de deformação de malhas, humanóides e framework V-Art. Na última seção são apresentados alguns trabalhos correlatos.

2.1 ALGORITMOS DE DEFORMAÇÃO DE MALHAS

Deformação de malhas é a modificação de modelos geométricos 3D pela distorção deles para outro lugar. Para ocorrer esta modificação são utilizadas operações matemáticas, onde se manipula a lista de vetores do modelo geométrico, gerando uma nova lista de vetores para o modelo (ARMSTRONG, 2001). A forma de representação do modelo geralmente possui restrições de como realizar a deformação. Isso é melhor demonstrado falando sobre as duas mais comuns representações chamadas de poligonal e de paramétrica.

2.1.1 Deformação de Representações Poligonais

A deformação de representações poligonais é feita a partir da movimentação dos vértices do modelo 3D, que são chamados de pontos de controle, mas, ao se fazer uma deformação, é necessário que seja respeitada a conectividade existente entre os vértices. A complexidade desta deformação é ocasionada por ter que respeitar a representação do objeto podendo ainda causar um problema de *aliasing* na imagem, reduzindo assim a resolução do objeto e degradando o modelo gerado pela deformação. A principal vantagem desta forma de representação é sua facilidade de implementação (WATT; WATT, 1992, p. 396).

2.1.2 Deformação de Representações Paramétricas

A deformação de representações paramétricas possui como vantagem mais significante em relação à poligonal o fato de ela poder manipular uma deformação de qualquer

complexidade e ainda assim parecer suave. Pois ao invés dos pontos de controle serem fixos nos vértices, eles podem se encontrar em qualquer lugar dentro do modelo 3D. O maior problema desta representação é que representar objetos de forma paramétrica é muito mais complexo do que de forma poligonal. Além disso, os objetos devem ser representados com uma topologia retangular para ser possível realizar a deformação (WATT; WATT, 1992, p. 396).

Existem várias formas de implementação de algoritmos que realizam a deformação de malhas em representações paramétricas. Entre esses algoritmos pode-se citar o *Free Form Deformation* (FFD), que será explicada mais detalhadamente por ser o algoritmo de deformação de malhas implementado.

O algoritmo FFD, segundo Parent (2002, p. 132-135), cria uma grade de coordenadas em uma configuração padrão (um paralelepípedo) sobre o objeto. Para cada vértice do objeto, coordenadas relativas a esta grade local são determinadas, que registram o vértice na grade. A grade é manipulada, e com esta manipulação cada vértice é mapeado novamente dentro da grade modificada, a qual os re-aloca no espaço. Inicialmente o sistema de coordenadas local é definido por um conjunto não necessariamente ortogonal de três vetores (`S`,`T`,`U`). Para se encontrar este três vetores no objeto são realizadas as operações descritas no Quadro 1.

```
Vetor max = maior(Grade);
Vetor min = menor(Grade);
Vetor P0 = Vetor(min.x, min.y, min.z);
Vetor S  = Vetor(max.x, min.y, min.z) – P0;
Vetor T  = Vetor(min.x, max.y, min.z) – P0;
Vetor U  = Vetor(min.x, min.y, max.z) – P0;
```

Quadro 1 – Forma para localização dos vetores `S`, `T` e `U`

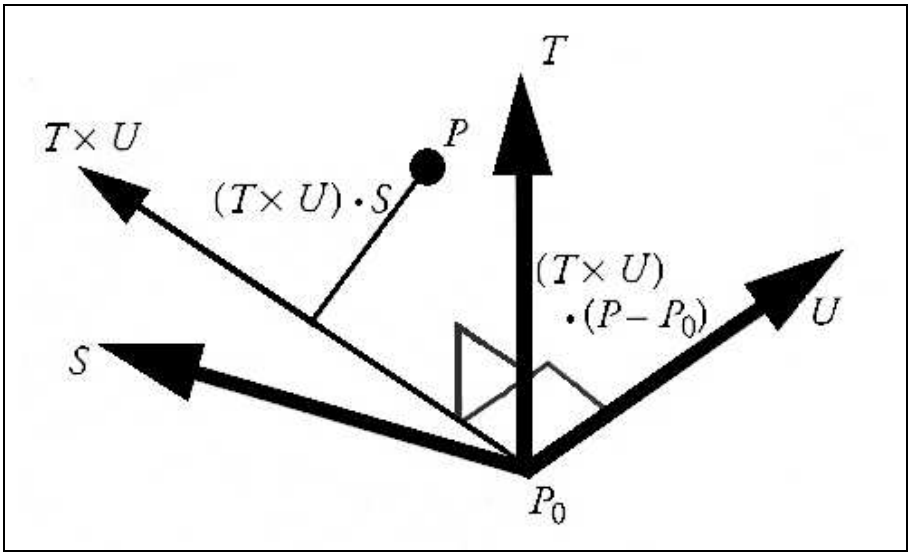
Nas operações mostradas no Quadro 1, inicialmente são encontradas as coordenadas máximas e mínimas da grade gerada em volta da malha. Após isso, é criado um vetor com os valores mínimos da grade, o qual foi atribuído para a variável `P0`. Com o valor de `P0` é gerado o vetor `S` que correspondem ao valor máximo da grade no eixo x e os valores mínimos nos eixos y e z, subtraindo os valores do vetor `P0`. O vetor `T` é gerado com os valores mínimos no eixo x e z e o valor máximo no eixo y subtraindo-se os valores do vetor `P0`, e o vetor `U` possui os valores mínimos nos eixos x e y e o valor máximo no eixo z, subtraindo-se os valores de `P0`.

Com os vetores `S`, `T` e `U` um vértice `P` qualquer da malha é registrado no sistema de coordenadas local determinando seus interpolantes trilineares, como mostrado no Quadro 2. Na Figura 1 são mostrados os três vetores (`S`, `T`, `U`) com o ponto registrado e seus interpolantes.

$$s = (T \times U).(P - P0)/((T \times U).S)$$
$$t = (U \times S).(P - P0)/((U \times S).T)$$
$$u = (S \times T).(P - P0)/((S \times T).U)$$

Fonte: Parent (2002).

Quadro 2 – Fórmulas dos interpolantes trilineares



Fonte: Parent (2002).

Figura 1 – Vetores S, T e U com um ponto registrado no sistema de coordenadas

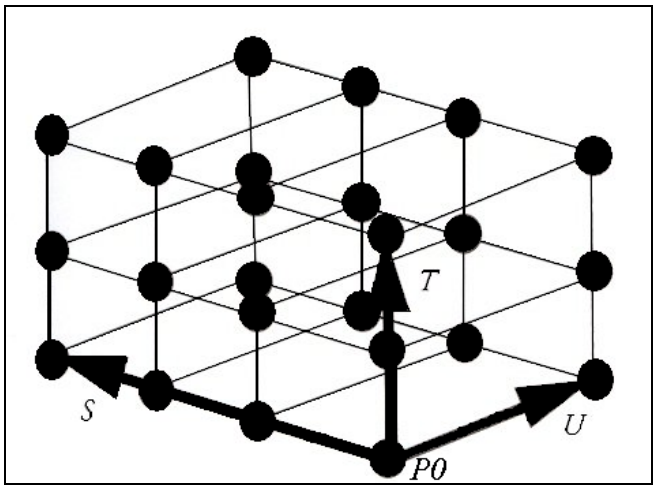
Nas equações mostradas no Quadro 2, o produto de dois vetores gera um terceiro vetor que é ortogonal aos dois primeiros. O denominador normaliza o valor que está sendo computado. Dadas as coordenadas locais (s, t, u) de um ponto e a grade de coordenadas locais não modificada, gerada pelos vetores S, T e U, uma posição de um ponto pode ser reconstruída em espaço global, simplesmente movendo na direção dos eixos de coordenadas local. A fórmula que reconstitui o ponto no espaço global é mostrada no Quadro 3.

$$P = P0 + s.S + t.T + u.U$$

Fonte: Parent (2002).

Quadro 3 – Fórmula para reconstrução da posição de um ponto

Para facilitar a modificação do sistema de coordenada local, uma grade de pontos de controle é criada no paralelepípedo definido pelos eixos S, T e U. Pode haver um número desigual de pontos nas três direções, na Figura 2 são mostrados os eixos S, T, U com os pontos de controle adicionados.



Fonte: Parent (2002).

Figura 2 – Pontos de controle no paralelepípedo criado com os eixos S, T e U

Se existem n_s pontos na direção `S`, n_t pontos na direção `T` e n_u pontos na direção `U`, os pontos de controle são localizados de acordo com a equação que encontra-se no Quadro 4.

$$P_{ijk} = P0 + \frac{i}{l}.S + \frac{j}{m}.T + \frac{k}{n}.U$$

Fonte: Parent (2002).

Quadro 4 – Fórmula para localização dos pontos de controle

Na fórmula mostrada no Quadro 4, `i`, `j` e `k` representam a quantidade de pontos de controle que existem em cada eixo `S`, `T` e `U` respectivamente e `l`, `m` e `n` são contadores que vão do intervalo de 0 até a quantidade de pontos de controle em cada eixo. Esses pontos `i`, `j` e `k` estão igualmente espalhados pelo paralelepípedo gerado pelos eixos `S`, `T` e `U`.

A deformação é gerada pela movimentação dos pontos de controle de sua posição inicial. A função que realiza a deformação é uma função trivariada da interpolação de Bézier, é utilizado esta interpolação por ela tender a fazer uma curva no modelo das malhas, assim diminuindo a distorção gerada no modelo. A posição deformada do ponto P_{stu} é determinado pela utilização das coordenadas locais `s`, `t`, `u`, definidas pelas fórmulas do Quadro 2, e a função de interpolação de Bézier é mostrada no Quadro 5. Nesta equação, `P(s,t,u)` representa a coordenada global do ponto deformado e P_{ijk} representa as coordenadas globais dos pontos de controle.

$$P(s,t,u) = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i . \left(\sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j . \left(\sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^k P_{ijk} \right) \right)$$

Fonte: Parent (2002).

Quadro 5 – Fórmula da interpolação de Bézier

No Quadro 5, `i`, `j` e `k` são contadores que começam em 0 e vão até `l`, `m` e `n` que são as quantidades de pontos de controle que existem em cada vetor `S`, `T` e `U`.

2.2 HUMANÓIDES

Humanóides são representações articuladas 3D de personagens animados com formato de um ser humano. O corpo de um humanóide pode ser dividido em duas partes: o esqueleto e a casca. O esqueleto de um humanóide é onde encontra-se a estrutura que este humanóide possui, isto é, de quais partes o humanóide é constituído. No esqueleto são descritas todas as partes geométricas que compõem um humanóide. A casca de um humanóide é colocada por cima de seu esqueleto, dando uma forma para este esqueleto e fazendo com que este se pareça com um humano (H | ANIM, 2004).

A modelagem do humanóide utilizada no V-Art envolve a determinação das articulações (ou juntas), juntamente com seus graus de liberdade, ou seja, dado um sistema de coordenadas cartesianas 3D, é definido em quais eixos a junta pode sofrer rotação e quais os seus limites (ângulos mínimo e máximo). Esses limites podem ser chamados de *Degrees Of Freedom* (DOFs). O modelo gerado é incluído na cena e usado para gerar as ações a serem animadas. A geração das ações é feita definindo-se o tempo de execução da ação, bem como o tipo de interpolação da movimentação dos DOFs, além do tempo inicial e final da execução do movimento dos DOFs.

2.3 FRAMEWORK V-ART

O *framework* V-Art foi desenvolvido com o intuito de criar ambientes virtuais 3D, especialmente os que contenham humanóides (NEDEL; FREITAS, 2007). Este *framework* surgiu do projeto VPAT (FREITAS; NEDEL, 2002), cujo objetivo principal era de permitir a criação de “pacientes virtuais” para sua utilização na área médica.

O V-Art é projetado para funcionar em multiplataformas (Linux, Windows, etc.) e funciona independente de *Application Programming Interface* (API) gráfica (OpenGL, Direct3D, etc.). Atualmente nesse *framework* é possível gerar animações com os humanóides movendo suas articulações de modo a representar ações dos seres humanos.

A estrutura do humanóide utilizado pelo V-Art é descrita em um arquivo no formato XML. Nesse arquivo existem os dados das juntas do humanóide, bem como os DOFs que esta junta possui, e em que arquivo em formato *WaveFront* se encontra a estrutura de vértices

desta junta. Na Figura 3 é mostrada a estrutura de uma junta no arquivo XML. Os locais apontados pelo número 1 representa onde se encontra os dados dos DOFs da junta. O atributo `description` da *tag*[1] `dof` é composta pela concatenação do tipo de movimentação da junta com o nome da mesma. A *tag* `position` possui o valor da posição inicial da junta, a tag `axis` possui o ponto onde deve ser gerada a transformação da estrutura da junta na movimentação e a *tag* `range` possui os valores máximo e mínimo que esta junta suporta nesse tipo de movimentação. O local mostrado pelo número 2 representa o arquivo binário onde se encontra a estrutura dos vértices desta junta, onde `filename` representa o nome do arquivo com os vértices, `description` é o nome da junta e `type` é o tipo do arquivo a ser lido. A definição da DTD completa do arquivo das juntas se encontra no Anexo A.

```
<node>
    <joint description="R_thighJoint" type="polyaxial">
        <dof description="flexR_thighJoint">
            <position x="-0.799850" y="0.155381" z="-0.911524"/>
            <axis x="0.998605" y="0.033865" z="-0.040510"/>
            <range min="-1.570796" max="0.523599" rest="0.750000"/>
        </dof>
(1)     <dof description="adductR_thighJoint">
            <position x="-0.799850" y="0.155381" z="-0.911524"/>
            <axis x="-0.033843" y="0.999426" z="0.001233"/>
            <range min="-0.349066" max="0.523599" rest="0.400000"/>
        </dof>
        <dof description="twistR_thighJoint">
            <position x="-0.799850" y="0.155381" z="-0.911524"/>
            <axis x="-0.040529" y="-0.000139" z="-0.999178"/>
            <range min="-0.698132" max="0.698132" rest="0.500000"/>
        </dof>
    </joint>
    <node>
(2)     <meshobject filename="policial_01.obj"  description="R_thighMesh"
         type = "obj"/>
    </node>
</node>
```

Figura 3 – Exemplo de um arquivo XML com estrutura de uma junta

Após a leitura dos dados do arquivo XML com as informações da junta, o V-Art cria uma estrutura de objetos de classes para guardar os dados da junta, essa estrutura com todas as classes utilizadas neste processo são mostradas na Figura 4. No Quadro 6, existe uma breve explicação do que é guardado em cada uma das classes que serão utilizadas posteriormente para geração da deformação de malha.

[1] São estruturas de linguagem de marcação que consistem em breves instruções, tendo uma marca de início e outra de fim.

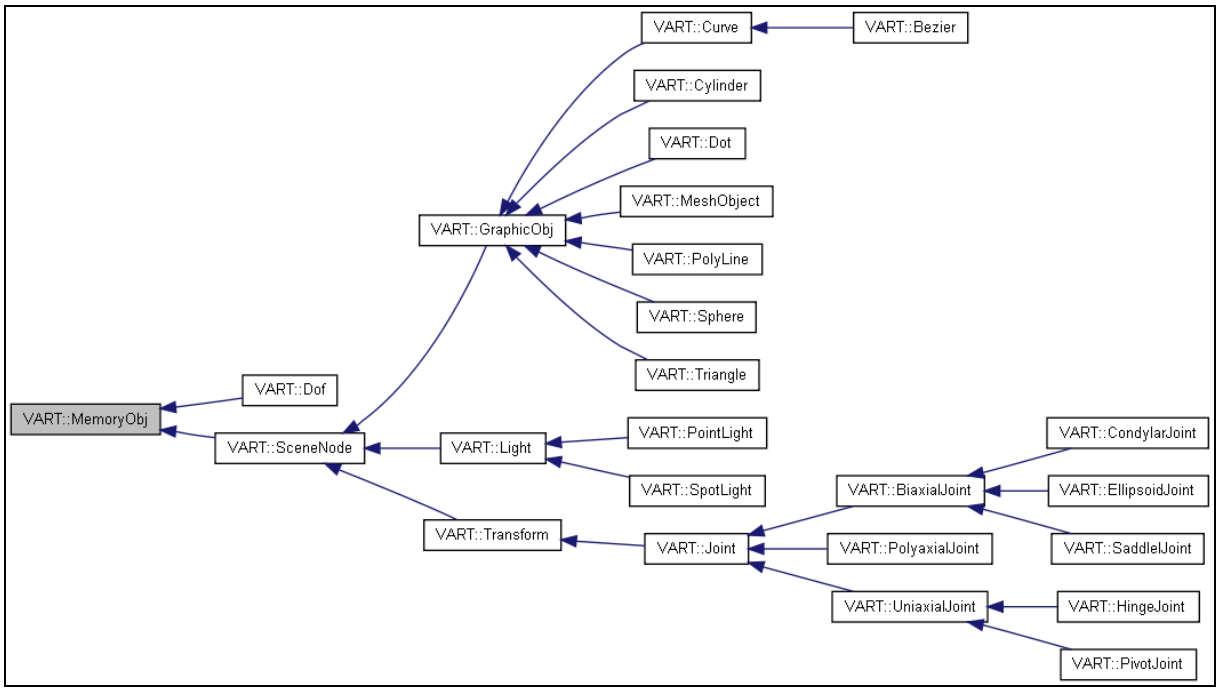


Figura 4 – Estrutura de classe de uma junta

CLASSE	DESCRIÇÃO
VART::MemoryObj	Representa uma junta
VART::Dof	Guarda os DOFs que a junta possui
VART::SceneNode	Guarda todos os dados de uma junta
VART::Joint	Guarda os dados da junta, como por exemplo seu nome
VART::MeshObject	Guarda a estrutura de vértices da junta

Quadro 6 – Explicação das principais classes da junta

Além de guardar a estrutura do humanóide, o V-Art possui a parte de animação do humanóide. Para isto também é lido outro arquivo XML, onde se encontra a estrutura da animação, esta estrutura é mostrada na Figura 5. No Anexo B encontra-se a DTD completa do arquivo XML das animações.

```
<action action_name= "caminha" speed = "1.0000" cycle = "true">
    <joint_movement joint_name = "R_legJoint" duration = "1.6000">
        <interpolation type= "ease-in_ease-out"/>
        <dof_movement dofID = "FLEX" initialTime = "0.0000" finalTime = "0.3250"
                      finalPosition = "0.2270"/>
    </joint_movement>
    <joint_movement joint_name = "R_armJoint" duration = "1.6000">
        <interpolation type= "ease-in_ease-out"/>
        <dof_movement dofID = "ADDUCT" initialTime = "0.0000" finalTime = "0.5000"
                      finalPosition = "0.9100"/>
    </joint_movement>
    <joint_movement joint_name = "R_armJoint" duration = "1.6000">
        <interpolation type= "ease-in_ease-out"/>
        <dof_movement dofID = "TWIST" initialTime = "0.0000" finalTime = "0.5000"
                      finalPosition = "0.6700"/>
    </joint_movement>
</action>
```

Figura 5 – Exemplo de um arquivo XML com os dados da movimentação

Neste arquivo existe primeiramente na *tag* `action` os dados do nome da ação, a

velocidade com que deve ser feita e se a animação é cíclica ou não. Dentro dessa *tag* existem várias *tags* `joint_movement`, onde cada uma é um movimento da animação, em cada movimento existe o nome da junta em que deve ser realizada a movimentação (`joint_name`), o tipo de interpolação usada (*tag* `interpolation`), qual tipo de DOF que este sendo modificado (*tag* `dof_movement`, atributo `dofID`), e quais os tempos inicial e final para que a animação seja realizada (`initialTime` e `finalTime` respectivamente).

Pela estrutura de classe que armazena e realiza a animação ser bastante separada dentro da estrutura de classes do V-Art, apenas será feita a descrição das principais classes utilizadas, dados que se encontram na Quadro 7.

CLASSE	**DESCRIÇÃO**
VART::Action	Inicializa e para a realização das movimentações no humanóide
VART::BoundingBox	Possui a *bounding box* da junta, utilizada para verificar contados que existam entre as juntas
VART::JointMover	Guarda os dados da movimentação descrita no arquivo XML
VART::XmlAction	Lê e transforma os dados do arquivo XML para dados que V-Art entenda
VART::Transform	Guarda a transformação que deve ser realizada em uma junta (quanto de rotação e translação deve ser feito na junta pra realocar ela no espaço)

Quadro 7 – Principais classes no armazenamento e execução da animação

Uma dos programas que atualmente utiliza-se do V-Art é o ActionBuilder, que é uma aplicação que permite importar um arquivo XML, com a estrutura do humanóide. Nessa estrutura encontra-se uma referência ao arquivo (em formato *WaveFront*) onde se encontra os vetores que correspondem a uma determinada malha. Nessa aplicação é possível selecionar uma parte do humanóide (formada por várias malhas) e a partir de teclas de atalho é possível criar uma animação no humanóide. A partir de outras teclas é possível inicializar a animação criada na aplicação ou se importar uma animação por um arquivo XML. Na Figura 6 encontra-se uma imagem de um humanóide no ActionBuilder. A caixa vermelha representa uma malha selecionada na aplicação.

Figura 6 – Humanóide no ActionBuilder

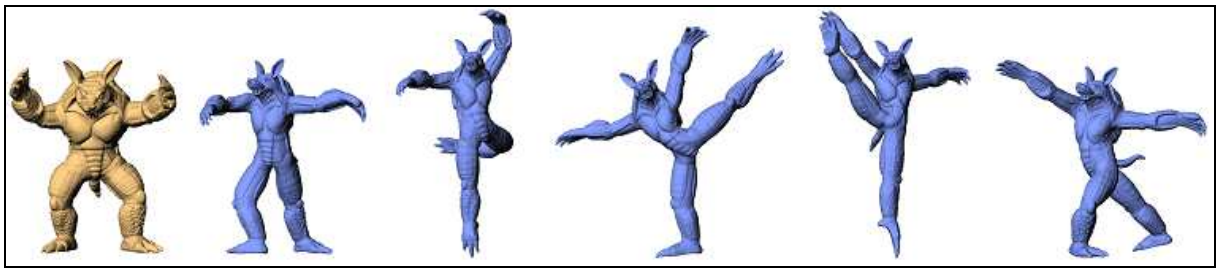
Nesse aplicativo é possível exportar a animação gerada no formato de arquivo XML, explicado anteriormente. É possível estender o código atual do *framework* desde que sejam respeitadas algumas regras (SCHNEIDER; VILLAMIL, 2006). Nessas regras encontra-se a descrição de como deve ser feita edição de um novo código fonte, como devesse chamar os métodos, as classes, os atributos e como deve ser realizado a documentação de todo o código fonte.

2.4 TRABALHOS CORRELATOS

Existem aplicações que possuem características semelhantes ao proposto neste trabalho, cada qual com suas peculiaridades. Dentre eles foram selecionados: Fast Multigrid Algorithm for Mesh Deformation, Free Form Deformation e o 3D Deformation Mesh Simulator.

2.4.1 Fast Multigrid Algorithm for Mesh Deformation

Neste trabalho é apresentada uma técnica de deformação de malha eficiente para grandes superfícies e volumes de malhas. Nesta técnica a formulação geral é reduzida uma um sistema Laplaciano, e apenas é utilizado o algoritmo de multigrade para solucionar as equações relevantes. Assim a técnica é aplicada em toda a malha em um tempo hábil e com pouco custo de memória. Na Figura 7 é apresentada a seqüência da animação de um tatu fazendo passos de balé.



Fonte: Shi et al. (2006).

Figura 7 – Seqüência da animação de um Tatu fazendo passos de balé

2.4.2 Free Form Deformation

O Free Form Deformation é um programa criado para exemplificar uma implementação do algoritmo FFD em modelos geométricos (GILSINAN IV; PENNER, 2002). Neste programa é lido um arquivo binário com a estrutura de vértices do modelo geométrico, e é gerado um paralelepípedo envolta desse modelo. A partir da movimentação dos pontos do paralelepípedo o modelo é deformado. Nesse programa é possível criar animações a partir de *frames* guardados da movimentação dos pontos do cubo. Este programa é feito na linguagem C++, e possui código fonte aberto. Na Figura 8 é mostrado um exemplo do funcionamento desse programa, com um modelo de um coelho antes da deformação e após a movimentação de dois pontos do paralelepípedo.

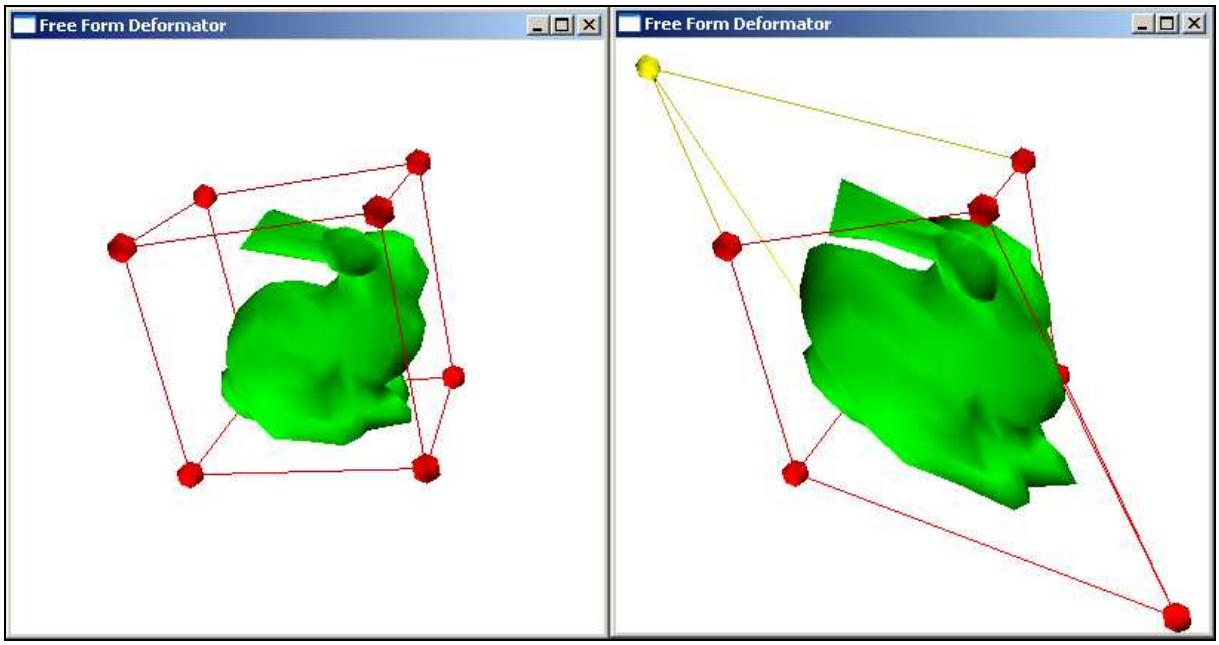
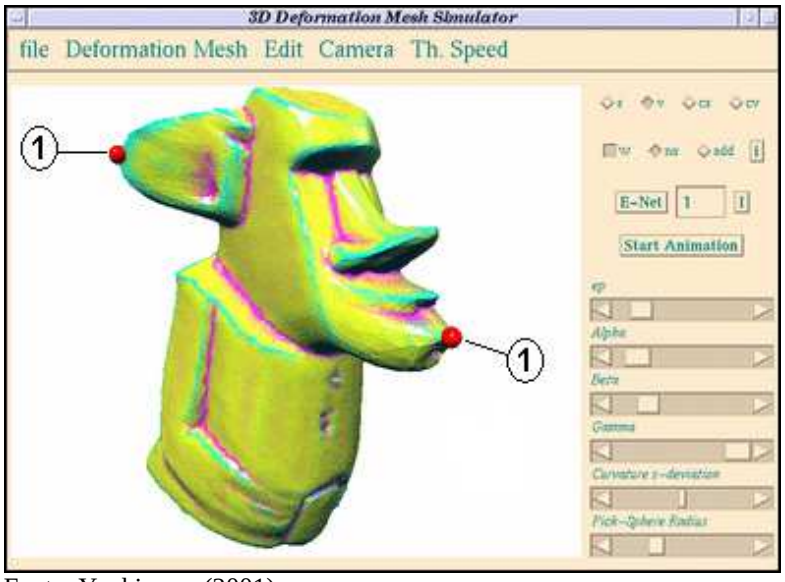


Figura 8 – Exemplo do Free Form Deformation

2.4.3 3D Mesh Simulator

O 3D Mesh Simulator é um simulador de deformação de malhas em objetos 3D. Este simulador foi desenvolvido na linguagem Java (YOSHIZAWA, 2001). Neste simulador o usuário carrega modelos 3D no formato PLY2[2]. A partir dessa imagem o usuário adiciona pontos de controle e com a movimentação desses pontos o simulador altera a imagem original. Na Figura 9 é mostrada a interface do simulador com seus pontos de controle (1) em uma imagem deformada. Este simulador foi desenvolvido apenas para a representação de como ocorre a deformação da malha em objetos. Não existe disponível os códigos fontes deste simulador e nem explicações mais aprofundadas de como ele funciona.

[2] PLY2 é um formato de arquivo que armazena malhas de triângulos 3D que se aproximam do formato real da imagem.



Fonte: Yoshizawa (2001).

Figura 9 - Simulador 3D Mesh Simulator

3 DESENVOLVIMENTO

Neste capítulo são abordados os requisitos da ferramenta e sua especificação. Além disso, é explicada a implementação do algoritmo de deformação de malhas no *framework* V-Art detalhando as ferramentas utilizadas em seu desenvolvimento. Por fim são apresentados os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Nesta implementação deverá ser adicionado ao *framework* V-Art:

a) adicionar pontos de controle no humanóide V-Art que será utilizado pelo algoritmo de deformação de malhas;
b) mostrar os pontos dos vetores de uma malha para que o usuário deforme manualmente a mesma;
c) realizar a deformação da malha no humanóide V-Art no momento das animações;
d) implementar rotinas que permitam que a malha seja deformada sem a interação do usuário após uma movimentação da articulação;
e) implementar utilizando o ambiente Microsoft Visual C++ 8.0.

3.2 ESPECIFICAÇÃO

A seguir é mostrada a especificação do *framework* V-Art com as devidas alterações. São utilizados para apresentar a modelagem o diagrama de classes e para mostrar o funcionamento o diagrama de seqüência.

3.2.1 Diagrama de Classes

Nesta seção apresenta-se o diagrama de classes do *framework* V-Art com as classes adicionadas para a execução do algoritmo de deformação de malhas, este diagrama se encontra na Figura 10.

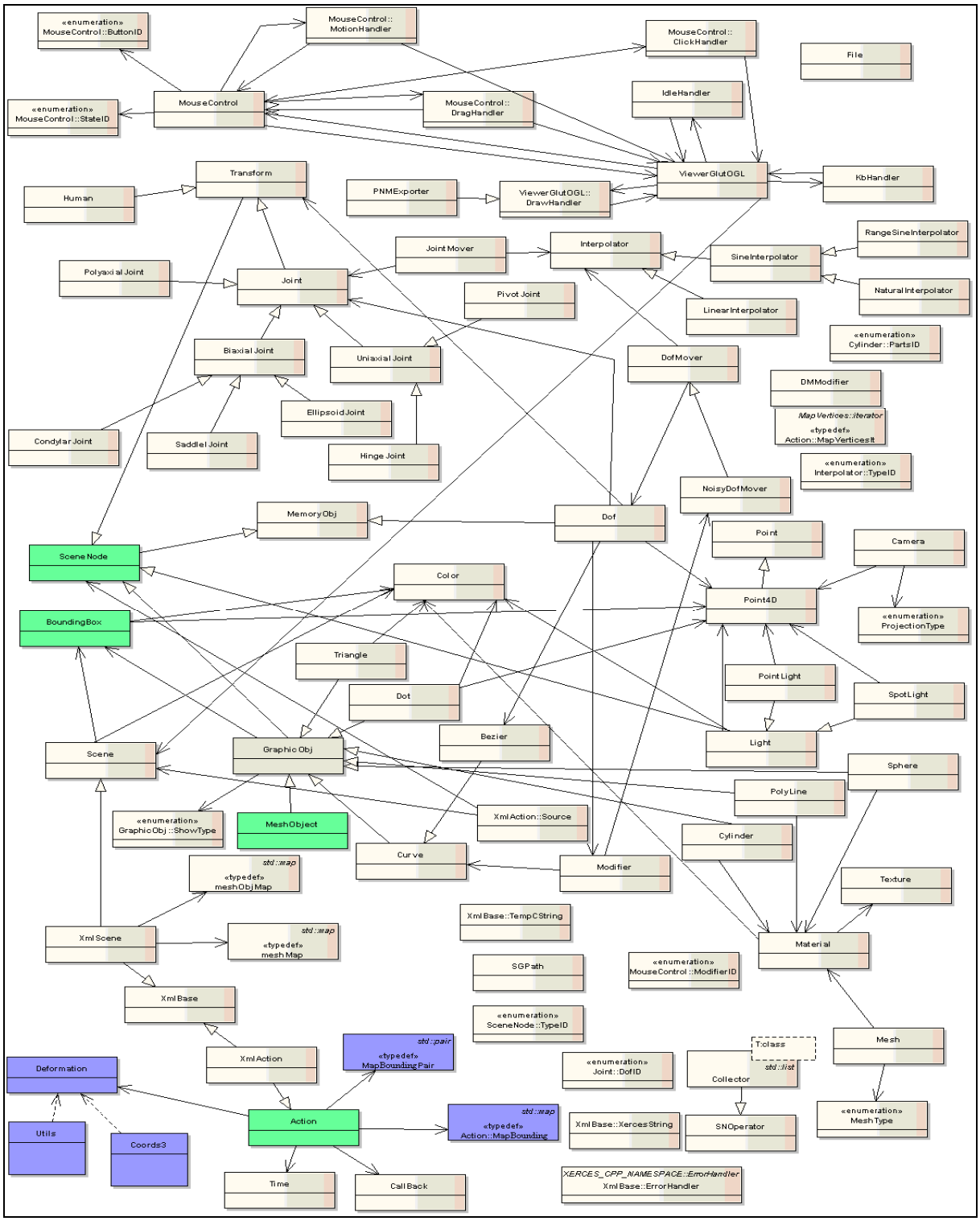


Figura 10 – Diagrama de classes do V-Art

Na Figura 10 mostra-se o diagrama de classes completo do *framework* V-Art. As classes que se encontram com a cor bege são classes que já existiam no *framework.* As classes em verde são classe que já existiam no *framework,* mas foram modificadas, as classe em azul foram classes adicionadas para o funcionamento da deformação de malha. A seguir será explicado cada uma dessas classes, começando pelas classes que foram alteradas.

A classe `Action`, como descrito anteriormente, é a classe responsável pela animação do humanóide. Para o funcionamento da deformação de malha torna-se necessário a inserção de chamadas para a deformação de malha durante o processo de movimentação do humanóide.

Na classe `MeshObject`, também descrita anteriormente, é a classe que dentre outros dados também guarda a estrutura de vértices da malha. Nesta classe é necessário adicionar a funcionalidade de ser possível informar uma nova lista com os vértices rearranjados pelo algoritmo de deformação de malha. Na classe `SceneNode`, verificou-se que existe uma estrutura de dados necessária para a realização da deformação de malhas, mas não existe acesso a essa estrutura, para tanto é necessário adicionar acesso a esta estrutura. Estas duas classes (`MeshObject` e `SceneNode`) possuem um forte acoplamento entre si, pois `MeshObject` é herdado por `GraphicObject` e `GraphicObject` é herdado por `SceneNode`. A seguir será explicado qual será a função de cada uma das classes criadas.

Dentre os dados criados existem as definições de tipos `MapBounding` e `MapBoundingPair`. `MapBounding` é uma estrutura de dados com o formato de um `map`[3]. Nela é guardada como chave o nome da junta e como o objeto a *boudingbox* da junta, isto é necessário para não ser realizado duas vezes deformação de malha em uma mesma junta a cada passo. `MapBoundingPair` é uma estrutura auxiliar para fazer busca dentro de uma estrutura de dados, facilitando a busca dentro da map `MapBounding`.

A classe `Deformation` é a classe responsável pela deformação de malha de uma junta. Ela é chamada pela classe `Action` durante o processo de movimentação do humanóide, para executar o algoritmo. A classe `Deformation` possui dependências com as classes `Utils` e `Coords3`. A classe `Coords3` possui a estrutura de um ponto, assim como as operações necessárias a ser realizadas em um ponto durante a deformação de malha. A classe `Utils` possui uma estrutura para o cálculo da fatoração necessária no algoritmo.

[3] Map é uma estrutura de dados existente na biblioteca STL.

3.2.2 Diagrama de Seqüência

Esta seção apresenta os diagramas de seqüência para a execução do algoritmo de deformação de malha. O diagrama da Figura 11 apresenta a seqüência necessária para a ativação da movimentação do humanóide. Para melhor visualizar esta seqüência foi utilizada a iteração necessária para executar a animação do V-Art utilizando-se do ActionBuilder.

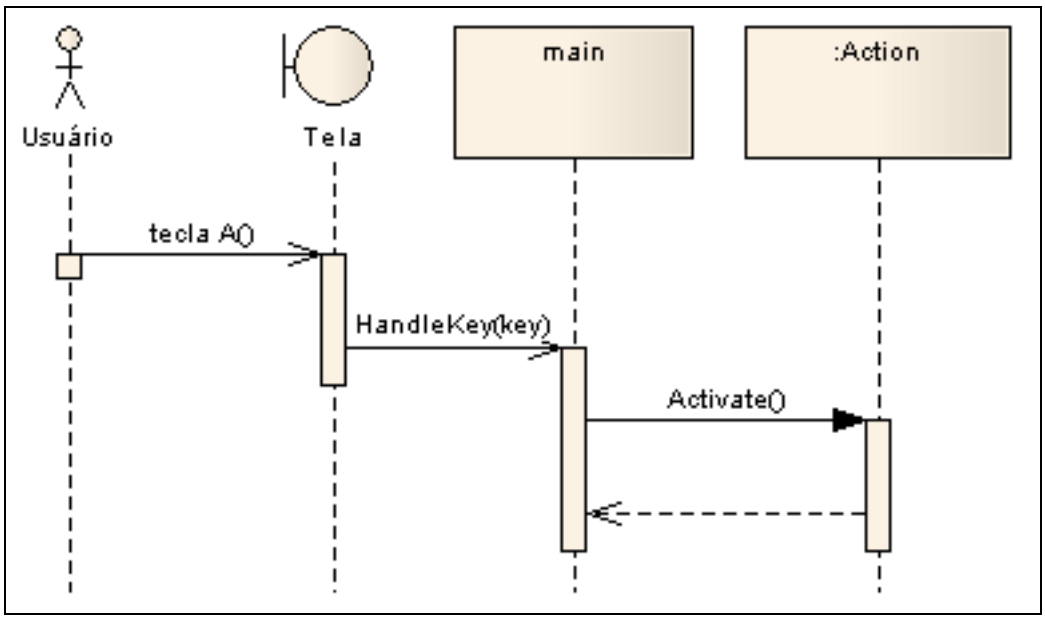


Figura 11 – Diagrama de seqüência da ativação da animação

A seqüência começa com o usuário pressionando a tecla “a”. Com a interrupção gerada pelo pressionamento da tecla “a”, o método `HandleKey` da classe `main` (classe do ActionBuilder) é chamado. Neste método é verificado se o humanóide está se movimentando ou não, se estiver movimentando ele chama o método `Deactivate` para parar a movimentação, senão ele chama o método `Activate`, da classe `Action`, com isso a animação do humanóide é ativada.

Na Figura 12 a seqüência da animação começa com um objeto `Action` chamando o método `Move` de todos os objetos `JointMover` que existirem em sua lista. Nos objetos `JointMover` ele chama o método `Move` de todos os `DofMover` que existirem. O `DofMover` por sua vez chama o método `MoveTo` passando como parâmetros a nova posição que deve ser alcançada. O objeto `Dof` chama o método `MakeRotation` para a classe `Transform`, que faz os cálculos necessários para realocar a junta em sua nova posição.

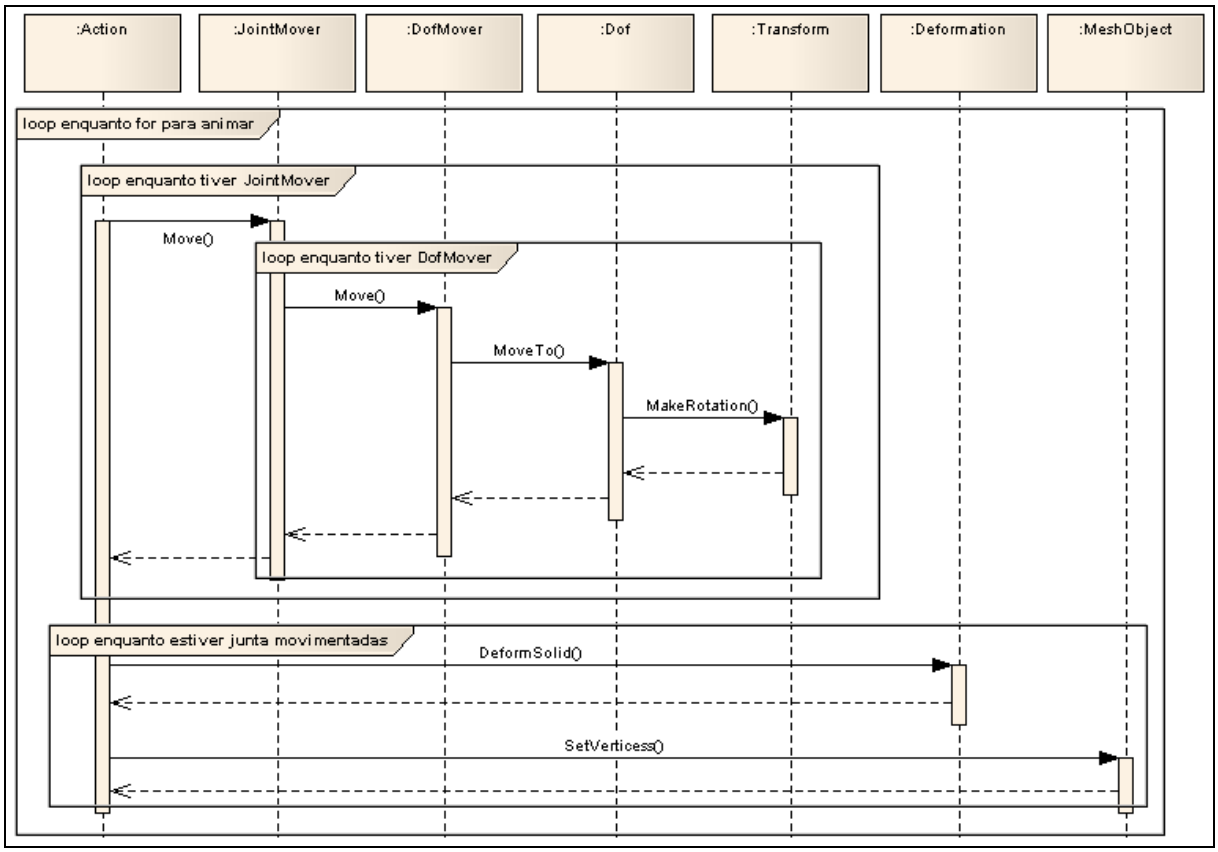


Figura 12 – Diagrama de seqüência da animação

Após a movimentação a classe `Action` chama o método `DeformSolid` da classe `Deformation`, que executa o algoritmo de deformação de malhas. Este método retorna a nova lista de vértices de uma junta, que é coloca na junta pelo método `SetVerticess` da classe `MeshObject`.

3.3 IMPLEMENTAÇÃO

Esta seção apresenta primeiramente as técnicas e ferramentas utilizadas na programação da aplicação. Em seguida é mostrado o modo como o usuário pode interagir com a ferramenta.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação do algoritmo de deformação de malha no *framework* V-Art foi

utilizado o ambiente de programação Microsoft Visual Studio 2005, utilizando-se da linguagem de programação C++. Para a criação dos diagramas foi utilizado o Enterprise Architect 7.0.

Para a criação da documentação do *framework* foi utilizado o Doxygen. O Doxygen é um programa que gera documentação de várias linguagens de programação, entre elas para C++ (HEESCH, 2008). Ele funciona a partir de comentários no código, e a partir de *tags* próprias adicionadas ao comentário.

Na criação do código fonte foi respeitada a formatação de escrita e documentação para código fonte no V-Art criadas por SCHNEIDER e VILLAMIL (2006). Também foi utilizada a biblioteca STL. STL é uma biblioteca genérica que provê uma estrutura básica para criação, busca e edição de estruturas de dados, assim como algoritmos básicos para utilização das estruturas (STANDARD TEMPLATE LIBRARY PROGRAMMER'S GUIDE, 2006).

3.3.2 Implementação do algoritmo de deformação de malha

Como dito anteriormente, para uma deformação acontecer é necessário iniciar uma animação, e com isto o algoritmo de deformação de malha FFD é executado.

Para que a animação não ficasse excessivamente lenta, por causa da estrutura como é feita a animação, onde é dado um pouco do tempo da animação para cada `JointMover`, com isso o algoritmo seria executado excessivas vezes para fazer uma deformação mínima com um custo alto (pela execução do algoritmo ser lenta). Assim o algoritmo de deformação de malha apenas é executado a cada 10 vezes que a classe `Action` chama a execução do método `Move` de todos os `JointMover` existentes. Antes que o algoritmo de deformação de malha seja executado, é necessário encontrar os interpolantes trilineares (Quadro 2 que se encontra na seção 2.1.2). Pois é necessário que seja feito o mapeamento dos vértices na *bounding box* inicial antes da execução do algoritmo, para se ter o mapeamento dos vértices no principio e assim usar este valor durante todo o algoritmo, para que a deformação funcione. O código que faz este mapeamento inicial se encontra no Quadro 8.

```
1   void VART::Deformation::InitialSTU(std::vector<double>& vertices, const
2                                      std::string name, VART::BoundingBox box){
3       mapSTUPair.first = mapSTU.find(name);
4       if (mapSTUPair.first == mapSTU.end()){
5           std::vector<double> verticesDef;
6           for(int w = 0; w < vertices.size() ; w += 3){
7               Coords3 X;     // the cartesian coords of the vertex
8               // Get this vertex in 3-space
9               X.x = vertices[w];
10              X.y = vertices[w+1];
11              X.z = vertices[w+2];
12              Coords3 max = VART::Deformation::MaxPosition(box);
13              Coords3 min = VART::Deformation::MinPosition(box);
14              Coords3 X0 = Coords3(min.getX(), min.getY(), min.getZ());
15              Coords3 S = Coords3(max.getX(), min.getY(), min.getZ()) - X0;
16              Coords3 T = Coords3(min.getX(), max.getY(), min.getZ()) - X0;
17              Coords3 U = Coords3(min.getX(), min.getY(), max.getZ()) - X0;
18              verticesDef.push_back(((T ^ U) % (X - X0)) / ((T ^ U) % S));
19              verticesDef.push_back(((U ^ S) % (X - X0)) / ((U ^ S) % T));
20              verticesDef.push_back(((S ^ T) % (X - X0)) / ((S ^ T) % U));
21          }
22      mapSTU.insert(MapSTU::value_type(name, verticesDef));
23      }
24  }
```

Quadro 8 – Cálculo dos interpoladores na implementação

O método `VART::Deformation::InitialSTU` recebe por parâmetro a lista de vértices de uma junta (`std::vector<double>& vertices`), o nome da junta (`std::string name`) e a *bounding box* da junta (`VART::BoundingBox box`).

Nas linhas 3 e 4 é onde se encontram a verificação se já foi realizado o cálculo dos interpoladores da junta. Na linha 5 é feita a declaração da lista que guardará os valores de `s`, `t` e `u`. Após é feito um *loop* por todos os vértices, da linha 7 a 11 é feita a declaração da variável que possuirá os valores das coordenadas de um vértice. Em seguida da linha 12 a 17 é realizada a criação dos vetores ortogonais (Quadro 1 da seção 2.1.2). Da linha 18 a 20 são realizados os cálculos para a criação do interpoladores, e eles são guardados na lista. Ao final na linha 22 a lista com os valores de `s`, `t` e `u` são armazenados em um `map`, possuindo como "chave" o nome da junta.

Nos cálculos realizados acima e em outros subseqüentes é utilizada a subscrição de operadores, onde é possível atribuir um operador a um tipo de objeto e ao aparecer um calculo onde o operador é utilizado, o código contido dentro da subscrição é executado. No Quadro 9 existe a implementação dos métodos subscritos nos cálculos acima. É observado que existe uma subscrição para o mesmo operador (*, multiplicação), mas a diferença entre eles é que a ordem em que os parâmetros aparecem são diferentes, algo que para a subscrição de operadores, representam dois métodos diferentes.

```
1   Coords3 operator- (Coords3& a, Coords3& b) { // Coord Subtract
2       Coords3 res;
3       res.x = a.x - b.x;
4       res.y = a.y - b.y;
5       res.z = a.z - b.z;
6       return res;
7   }
8
9   double operator% (Coords3& a, Coords3& b) { // Dot Prod
10      return(a.x * b.x +
11             a.y * b.y +
12             a.z * b.z);
13  }
14
15
16  Coords3 operator^ (Coords3& a, Coords3& b) { // Cross Prod
17      Coords3 ret;
18      ret.x = a.y*b.z - a.z*b.y;
19      ret.y = a.z*b.x - a.x*b.z;
20      ret.z = a.x*b.y - a.y*b.x;
21      return ret;
22  }
23
24  Coords3 operator* (double a, Coords3& b) { // Scalar Multiply
25      Coords3 res;
26      res.x = a * b.x;
27      res.y = a * b.y;
28      res.z = a * b.z;
29      return res;
30  }
31
32  Coords3 operator* (Coords3& b, double a) { // Scalar Multiply
33         Coords3 res;
34         res.x = a * b.x;
35         res.y = a * b.y;
36         res.z = a * b.z;
37         return res;
38  }
```

Quadro 9 – Subscrição de operadores

Tendo os valores dos interpoladores é possível realizar a deformação de malha no humanóide. Após a realização da movimentação das juntas 10 vezes, o algoritmo de deformação de malha é realizado. No Quadro 10 existe o código que realiza a retirada dos dados necessários para a realização da deformação.

```
1   MapBounding auxBB;
2   MapBoundingPair auxBBPair;
3   for (iter = jointMoverList.begin(); iter != jointMoverList.end(); ++iter)
4   {
5       const Joint* jo = (*iter)->GetAttachedJoint();
6       const VART::SceneNode* cene = dynamic_cast<const VART::SceneNode*>(jo);
7       std::list<SceneNode*> lista = cene->childList;
8       list<VART::SceneNode*>::const_iterator iteracao;
9       iteracao = lista.begin();
10      VART::MeshObject* mesh = dynamic_cast<VART::MeshObject*>(*iteracao);
11      const VART::BoundingBox box = mesh->GetBoundingBox();
12      auxBBPair.first = auxBB.find(mesh->GetDescription());
13      if (auxBBPair.first == auxBB.end()){
14          VART::BoundingBox auxBox;
15          auxBB.insert(MapBounding::value_type(mesh->GetDescription(), box));
16          VART::BoundingBox* box2 = new VART::BoundingBox(box.GetSmallerX(),
17                                                          box.GetSmallerY(),
18                                                          box.GetSmallerZ(),
19                                                          box.GetGreaterX(),
20                                                          box.GetGreaterY(),
21                                                          box.GetGreaterZ());
22          const VART::Transform* trans=dynamic_cast<const VART::Transform*>(jo);
23          box2->ApplyTransform((*trans));
24          auxBox.SetGreaterX(MinMax(box.GetGreaterX(), box2->GetGreaterX(), 1));
25          auxBox.SetGreaterY(MinMax(box.GetGreaterY(), box2->GetGreaterY(), 1));
26          auxBox.SetGreaterZ(MinMax(box.GetGreaterZ(), box2->GetGreaterZ(), 1));
27          auxBox.SetSmallerX(MinMax(box.GetSmallerX(), box2->GetSmallerX(), 0));
27          auxBox.SetSmallerY(MinMax(box.GetSmallerY(), box2->GetSmallerY(), 0));
28          auxBox.SetSmallerZ(MinMax(box.GetSmallerZ(), box2->GetSmallerZ(), 0));
29          mesh->SetVerticess(deform->DeformSolid(mesh->GetVerticesCoordinates(),
30                                                 auxBox));
31      }
32  }
33
```

Quadro 10 – Código que retira os dados para a deformação

Primeiramente as variáveis `auxBB` e `auxBBPair` são declaradas (linhas 1 e 2 respectivamente), e utilizadas para que não seja realizada a deformação de malha na mesma junta mais de uma vez, pois a lista de `JointMover` (`jointMoverList`) pode ter mais de um movimento na mesma junta. Da linha 5 a linha 11 são feitos as transformações e buscas para conseguir ter os valores necessários para a deformação da malha.

Na linha 3 é onde se encontra o *loop* para todos os `JointMover` que existem, na linha 5 é atribuído a variável `jo` a junta desse movimento. Na linha 6 é feita uma conversão da variável `jo` de `Joint` para `SceneNode` (como estava no diagrama de classes um `Joint` estende de `Transform` que estende de `SceneNode`). Com a variável `cene` na linha 7 é retirada a variável `childList`, que possui em sua primeira posição o `MeshObject` da junta e nas outras posições os filhos dessa junta (por exemplo a cintura tem como filhos a perna esquerda e a perna direita).

Após isso é feito um *iterator* (linha 8, `iteração`) para retirar o primeiro valor da variável lista (linha 9, método `begin`), e é realizada uma conversão para um `MeshObject` (linha 10). Com o `MeshObject` na linha 11 é retirada a *bounding box* da junta e colocada na variável `box`. Nas linhas 12 e 13 é feita a busca para ver se já foi feita a deformação da malha

nesta junta (`auxBB.find(mesh->GetDescription())`), se o valor que estiver em `auxBBPair.first` for igual a `auxBB.end`, então nessa junta ainda não foi realizada a deformação de malha.

Entre as linhas 14 a 28 é realizada uma seqüência de ações para realocar a *bounding box* da junta, pois a mesma é guardada utilizando-se das coordenadas locais. Mas para se fazer a deformação de malha, é necessário que se possua a diferença entre a *bounding box* com as coordenadas locais e a *bounding box* com as coordenadas globais. Na Figura 13 é mostrado o que é gerado nestes cálculos, onde a caixa vermelha é a *bounding box* com as coordenadas locais (`box`), o retângulo preto é a mesma *bounding box* mas após a transformação (`box2`), e o retângulo verde é a *bounding box* gerada a partir das duas (`boxAux`).

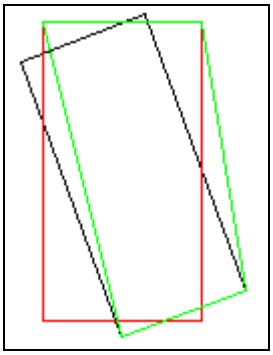


Figura 13 – Demonstração da operação realizadas com a *bounding box*

Na linha 14 é declarada uma variável do tipo `VART::BoundingBox` (`auxBox`), e após é inserida em `auxBB` esta junta, para que neste *loop* não entre mais nesta estrutura (linha 15). Na linha 16 é criada uma variável idêntica a `box` (`box2`), e na linha seguinte é realizada uma conversão da junta (variável `jo`) para um `Transform`, e esta transformação é aplicada encima da `box2`.

Da linha 24 a linha 28 são atribuídos os valores máximos e mínimos (método `MinMax`, se o terceiro valor for 1 este método trará o valor máximo, se for 0 trará o mínimo) para a criação da nova *bounding box* que será utilizada pelo algoritmo. E por último é chamado o método `DeformSolid` da classe `Deformation` (cujo objeto é a variável `deform`), enviando a lista de vértices da junta, bem como `auxBox`, já que este método retorna a nova lista de vértices da junta, já é adicionada esta lista ao `MeshObject` utilizando o método `SetVerticess`.

No Quadro 11 é descrito o início do algoritmo de deformação de malha.

```
1 | l = m = n = 1;
2 | Coords3 max = VART::Deformation::MaxPosition(box);
3 | Coords3 min = VART::Deformation::MinPosition(box);
4 | Coords3 X0 = Coords3(min.getX(), min.getY(), min.getZ());
5 | Coords3 S = Coords3(max.getX(), min.getY(), min.getZ()) - X0;
6 | Coords3 T = Coords3(min.getX(), max.getY(), min.getZ()) - X0;
7 | Coords3 U = Coords3(min.getX(), min.getY(), max.getZ()) - X0;
```

Quadro 11 – Início do algoritmo de deformação de malha

Inicialmente são inicializados os valores de `l`, `m` e `n`, eles sempre terão o valor um porque sempre haverá apenas um ponto de controle em cada direção nesta implementação. Após isso, das linhas 2 a linha 7 são feitos os cálculos dos vetores ortogonais (`X0, S, T` e `U`) com os valores da nova *bounding box.*

Tendo os novos valores dos vetores, é feita a declaração das variáveis que serão utilizadas para a finalização do algoritmo, esta declaração se encontra no Quadro 12.

```
double s, t, u;  /* the (s, t, u) bounding box space coords */

Coords3 Xnew;
Coords3 Xnew2;
Coords3 Xnew3;

std::vector<double> verticesDef;
```

Quadro 12 – Declaração das variáveis para o algoritmo

As variáveis `s`, `t` e `u`, são as mesmas que se encontram no Quadro 2 (seção 2.1.2). As variáveis `Xnew`, `Xnew2` e `Xnew3`; serão utilizadas para conseguir o novo valor dos vértices, e a variável `verticesDef` é a que possuirá a nova lista de vértices da junta. No Quadro 13 existem os cálculos que terminarão com a execução do algoritmo de deformação de malha.

Inicialmente é feito um *loop* para cada um dos vértices (eles estão salvos em uma lista onde cada posição é um valor no espaço cartesiano). Com isso, das linhas 4 a 6 são pegos os valores no espaço cartesiano de um vértice. Das linhas 8 a 10 são retirados os valores de `s`, `t` e `u` do vértice (valores que foram calculados anteriormente). Por último das linhas12 a linha 31 é feito uma seqüência de *loops* para realocar o vértice na sua nova posição, estes *loops* representam a fórmula que se encontra no Quadro 5 (seção 2.1.2), onde o método `Utils::combination` faz a fatoração entre dois números e o método `pow` faz com que o primeiro número seja elevado ao segundo número. Na linha 20 é realizado o cálculo para localização do ponto de controle, que se encontra no Quadro 4 (seção 2.1.2). Por último são adicionados a lista `verticesDef` os novos valores de `x`, `y` e `z` do vértice. E ao final do algoritmo, na linha 36, a nova lista de vértices é retornada.

```
1   for(int w = 0; w < vertices.size() ; w += 3){
2       Coords3 X;      // the cartesian coords of the vertex
3       // Get this vertex in 3-space
4       X.x = vertices[w];
5       X.y = vertices[w+1];
6       X.z = vertices[w+2];
7
8       s = stu[w];
9       t = stu[w+1];
10      u = stu[w+2];
11      Xnew.setCoords(0,0,0);
12      for (int i = 0; i <= l; i++) {
13          Xnew2.setCoords(0,0,0);
14          for (int j = 0; j <= m; j++) {
15              Xnew3.setCoords(0,0,0);
16              for (int k = 0; k <= n; k++) {
17                  Xnew3 += Utils::combination(n, k) *
18                           pow((1 - u), (n - k)) *
19                           pow(u, k) *
20                          (X0+(i/(double)l)*S+(j/(double)m)*T+(k/(double)n)*U);
21              }
22              Xnew2 += Utils::combination(m, j) *
23                       pow((1 - t),(m - j)) *
24                       pow(t,j) *
25                       Xnew3;
26          }
27          Xnew += Utils::combination(l, i) *
28                  pow((1 - s),(l - i)) *
29                  pow(s,i) *
30                  Xnew2;
31      }
32      verticesDef.push_back(Xnew.x);
33      verticesDef.push_back(Xnew.y);
34      verticesDef.push_back(Xnew.z);
35  }
36  return verticesDef;
```

Quadro 13 – Final algoritmo de deformação de malha

3.3.3 Operacionalidade da implementação

Para executar o algoritmo de deformação de malhas, é necessário ter o executável do ActionBuilder, assim como os arquivos `xerces-c_2_6D.dll`, `MSVCR71D.dll`, `DevIL.dll`. Esses arquivos devem estar no mesmo diretório que o executável do ActionBuilder.

O ActionBuilder tem que ser executado via linha de comando, pois é necessário passar dois parâmetros para o mesmo. Estes dois parâmetros são respectivamente o arquivo XML com a estrutura do humanóide e o arquivo XML com a animação do humanóide. O arquivo com a estrutura do humanóide deve possuir a extensão (.xml) após o nome do arquivo, já o arquivo da animação do humanóide não pode possuir esta extensão, caso qualquer um dos dois parâmetros não estejam com estas configurações, o ActionBuilder irá disparar um erro e não executará. No Quadro 14 existe um exemplo de como deve estar a linha de comando para executar o ActionBuilder.

```
$>ActionBuilder_d.exe policial_01_arm.xml RPC\caminha
```

Quadro 14 – Exemplo para execução do ActionBuilder

Ao ser executado o ActionBuilder é exibida uma janela com o humanóide centralizado. É possível selecionar uma junta clicando sobre ela, ao se fazer isso apresenta-se na linha de comando os dados sobre a junta selecionada. Para iniciar/parar a animação apenas é necessário pressionar a tecla “a”. Para ativar/parar a deformação de malha no humanóide basta pressionar a tecla “v”, caso ele esteja aplicando o algoritmo de deformação de malha o programa irá parar, senão ele irá começar a deformar a malha. Caso seja necessário aumentar ou diminuir a quantidade de *loops* necessários para gerar a deformação, basta pressionar 9 ou 0, respectivamente. Para utilizar-se das outras funções do ActionBuilder, apenas é necessário pressionar “h”, que com isso irá aparecer no console uma lista com todos os comandos assim como um resumo do que cada comando faz.

3.4 RESULTADOS E DISCUSSÃO

Nesta seção serão apresentados os resultados alcançados na implementação do algoritmo de deformação de malha no humanóide do V-Art, os quais foram os seguintes:

a) o algoritmo de deformação de malha funcionou no humanóide, isso pode ser verificado na Figura 14, onde aparece o humanóide deformado;
b) o algoritmo selecionado (FFD) mostrou-se rápido no momento da execução, apesar de que no momento em que ocorre à deformação, os *Frames Per Second* (FPS) do aplicativo caem, ocasionado pelo alto processamento realizado em um instante apenas, fazendo com que a animação fique menos real;
c) a utilização do algoritmo FFD não se mostrou como a melhor escolha para se realizar a deformação, pois por este algoritmo deformar toda a malha durante a deformação, utilizando-se de uma *bounding box*, ele acabou deformando excessivamente o humanóide fazendo com que o mesmo aumentasse de tamanho, isto também é visível na Figura 14;
d) o humanóide do V-Art demonstrou não estar preparado para receber a deformação de malha, pois ele foi construído pensando na não existência de um algoritmo de deformação de malha no V-Art, por isso as articulações das juntas se encontram “dentro” da malha de outra junta e nas juntas onde mais aparece uma deformação

(joelho e cotovelo), existe um enchimento, fazendo com que a deformação não fique muito visível. Isso é demonstrado pela Figura 15, onde se percebe que no cotovelo do humanóide existe uma esfera, para não mostrar a união entre as juntas.

Figura 14 – Humanóide deformado

Figura 15 – Demonstração da esfera no cotovelo do humanóide

Com relação aos trabalhos correlatos, Fast Multigrid Algorithm for Mesh Deformation, 3D Mesh Deformation e Free Form Deformation, a principal diferença em relação a todos os três é que em ambos sempre se trabalha apenas com uma malha, mas no V-Art é necessário trabalhar com um conjunto de malhas que o humanóide possui ao mesmo tempo e é necessário manter a ligação que existe entre as malhas. Em comparação com o 3D Mesh Deformation, outra diferença é a presença da animação, pois nele a deformação é gerada em uma imagem estática com a interferência do usuário, mas no V-Art ela é gerada durante uma animação, sem interferência externas para deformar.

Os testes realizados foram feitos utilizando-se do humanóide do V-Art (Police Man) que possui 12 juntas e é composto por 3.667 polígonos, para a animação foi utilizado um exemplo de uma caminhada.

Durante os testes foram feitas verificações visuais de quantas vezes era necessário ser chamado o método `Move` da classe `Action`, para que ocorresse uma modificação significativa na malha para ser executado o algoritmo de deformação de malha, para assim melhorar o desempenho da animação. Com a observação da movimentação chegou-se a conclusão que após 10 *loops* o humanóide movimentava o suficiente para assim o deformar. Na Figura 16 aparece a diferença que ocorre no humanóide após 10 *loops.*

Figura 16 – Mudança do humanóide após 10 *loops*

Durante os testes verificou-se que o algoritmo, apesar de deformar não corretamente os membros, funcionava perfeitamente, pois ao término de um movimento completo a malha retorna a sua posição inicial. Na Figura 17 é apresentada uma seqüência da animação em quatro quadros para demonstrar o processo de deformação da malha, as imagens possuem uma ordem representada pelas letras do alfabeto.

Figura 17 – Animação do humanóide

Esta seqüência, que representa um ciclo de caminhada, começa (letra A), com o

humanóide parado. Após vem a seqüência de uma caminha normal, com a alternância de posição entre os braços e as pernas, indo um pra frente e outro para trás (letras B e C), e por último o humanóide voltando para a posição inicial para assim inverter as posições dos braços e das pernas (os membros que estavam na frente vão para trás e vice-versa).

4 CONCLUSÕES

Conclui-se que o algoritmo de deformação de malhas FFD, funciona no humanóide V-Art. Foi possível atribuir os pontos de controle em cima das malhas do humanóide, utilizando-se da *bounding box* existente no V-Art. Conseguiu-se fazer com que houvesse deformação no humanóide durante sua animação, mas verificou-se que utilizar a *bounding box* do V-Art não se mostrou eficiente no aspecto visual.

A utilização do Microsoft Visual Studio 2005 auxiliou bastante na criação do código e o Enterprise Architect, se mostrou bastante útil por ser possível gerar os diagramas do V-Art a partir de engenharia reversa (pelo mesmo não possuir um diagrama de classes), pois apesar do V-Art possuir uma documentação de diagramas, gerados pelo DoxyGen, estes diagramas não se encontram em total conformidade com a diagramação UML.

Apesar da deformação não ficar visualmente aceitável para representação de um ser humano, mostrou-se que é possível implementar um algoritmo de deformação de malha no V-Art, mas é necessário que haja uma melhora no algoritmo, principalmente na área de quando se deve deformar o humanóide, para assim aumentar a taxa de FPS. Pois a atual implementação, apenas leva em conta a quantidade de vezes que um método é chamado, mas dependendo do computador em que é executado o V-Art esta quantidade pode não ser suficiente para fazer o humanóide se mexer.

Observou-se que o modelo do humanóide utilizado no V-Art não se encontra preparado para a implementação de um algoritmo de deformação de malha. Primeiramente porque o mesmo foi criado de modo a não aparecer à ligação das juntas, fazendo com que visualmente a deformação não apareça. Também o método de movimentação segue no sentido oposto aos algoritmos, pois nos algoritmos a movimentação da *bounding box* é que gera a mudança na malha, mas no V-Art é a modificação da malha que altera a *bounding box* e após isso é executado o algoritmo para rearranjar os vértices.

4.1 EXTENSÕES

As extensões encontradas para este trabalho foram:

a) implementar outro algoritmo de deformação de malha, para fins de comparação

com o atual, podendo inclusive ser um algoritmo de deformação de representações poligonais, mas levando-se em conta que este tipo de algoritmo é mais lento;

b) adicionar a possibilidade de ocorrer a deformação manual da malha do humanóide, já levando-se em conta que essa modificação acarretaria em ter que modificar a estrutura atual do arquivo XML da animação;
c) criar um novo humanóide onde as juntas fiquem visíveis, para realização de novos testes;
d) criar uma maior quantidade de animações, para testar a eficiência do algoritmo em casos de pequenas movimentações e movimentações mais sutis, como por exemplo a movimentação de um dedo.

REFERÊNCIAS BIBLIOGRÁFICAS

ARMSTRONG, J. **Macromedia – director tutorial**: performing mesh deformation. [S.l.], 2001. Disponível em: <http://www.adobe.com/support/director/3d_lingo/mesh_deformation/>. Acesso em: 18 set. 2007.

FREITAS, C. M. D. S.; NEDEL, L. P. **VPAT**: visualização e interação com pacientes virtuais. [Porto Alegre], [2002]. Disponível em: <http://www.inf.ufrgs.br/cg/vpat/vpatDescription.html>. Acesso em: 18 set. 2007.

GILSINAN IV, J.; PENNER, S. **Keyframe animation of free form mesh deformations**. [S.l.], 2002. Disponível em: <http://www.gilsinan.com/cs276final/>. Acesso em: 2 abr. 2008.

H | ANIM humanoid animation. [S.l.], [2004]. Disponível em: <http://www.h-anim.org/Specifications/H-Anim200x/ISO_IEC_FCD_19774/concepts.html#Manipulation>. Acesso em: 18 set. 2007.

HEESCH, D. V. **Doxygen.** [S.l.], 2008. Disponível em: <http://www.stack.nl/~dimitri/doxygen/>. Acesso em: 24 abr. 2008.

NEDEL, L. P.; FREITAS, C. D. S. **V-ART**: virtual articulations for virtual reality. [S.l.], 2007. Disponível em: <http://www.codeplex.com/vart>. Acesso em: 6 set. 2007.

PARENT, R. **Computer animation**: algorithms and techniques. Berkeley: Morgan Kaufmann, 2002.

SCHNEIDER, B.; VILLAMIL, M. **Recomendações V-Art**. [S.l.], [2006?]. Disponível em: <http://www.inf.ufrgs.br/~boschneider/v-art/recomendacoes.html>. Acesso em: 6 set. 2007.

SHI, L. et al. A fast multigrid algorithm for mesh deformation. **ACM Transactions on Graphics**, New York, v. 25, n. 3, p. 1108-1117, jul. 2006.

STANDARD TEMPLATE LIBRARY PROGRAMMER'S GUIDE. [S.l.], 2006. Disponível em: <http://www.sgi.com/tech/stl/index.html>. Acesso em: 20 abr. 2008.

WATT, A.; WATT, M. **Advanced animation and rendering techniques**: theory and practice. Nova Iorque: ACM Press, 1992.

YOSHIZAWA, S. **A simple approach to interactive free-form shape deformations**. [S.l.], 2001. Disponível em: <http://www.mpi-inf.mpg.de/~shin/Research/DeformMesh/DefMesh.html>. Acesso em: 5 set. 2007.

ANEXO A – DTD completa do arquivo XML das juntas

No Quadro 15 encontra-se a DTD completa do arquivo XML das juntas do V-Art.

```
<!-- DTD for a Vpat scene -->
<!-- Author: Andreia Schneider, Feb 2006 -->
<!ELEMENT scene (camera*, node+)>
<!ELEMENT camera (position,target,up_vector)>
<!ELEMENT position EMPTY>
<!ELEMENT target EMPTY>
<!ELEMENT up_vector EMPTY>
<!ELEMENT node ((bezier|sphere|cylinder|meshobject|directionallight|
spotlight|pointlight|transform|joint), node*)>
<!ELEMENT bezier (ponto1, ponto2, ponto3, ponto4)>
<!ELEMENT ponto1 EMPTY>
<!ELEMENT ponto2 EMPTY>
<!ELEMENT ponto3 EMPTY>
<!ELEMENT ponto4 EMPTY>
<!ELEMENT sphere (radius, material)>
<!ELEMENT cylinder (radius, height, material)>
<!ELEMENT height EMPTY>
<!ELEMENT radius EMPTY>
<!ELEMENT material EMPTY>
<!ELEMENT meshobject (material?)>
<!ELEMENT directionallight (intensity, ambientIntensity, color, enabled,
position)>
<!ELEMENT pointlight (intensity, ambientIntensity, color, enabled,
position, attenuation)>
<!ELEMENT spotlight (intensity, ambientIntensity, color, enabled,
position, attenuation, beam_Width, cut_Off_Angle)>
<!ELEMENT intensity EMPTY>
<!ELEMENT ambientIntensity EMPTY>
<!ELEMENT color EMPTY>
<!ELEMENT enabled EMPTY>
<!ELEMENT attenuation EMPTY>
<!ELEMENT beam_Width EMPTY>
<!ELEMENT cut_Off_Angle EMPTY>
<!ELEMENT transform (translation|scale|rotation|matrix)>
<!ELEMENT translation EMPTY>
<!ELEMENT scale EMPTY>
<!ELEMENT rotation EMPTY>
<!ELEMENT matrix EMPTY>
<!ELEMENT joint (dof+)>
<!ELEMENT dof ( position, axis, range )>
<!ELEMENT axis EMPTY>
<!ELEMENT range EMPTY>
<!ATTLIST scene description CDATA #REQUIRED>
<!ATTLIST camera description CDATA #REQUIRED
                            type CDATA #REQUIRED>
<!ATTLIST joint description CDATA #REQUIRED
                type (biaxial|polyaxial|uniaxial) #REQUIRED >
<!ATTLIST dof description CDATA #REQUIRED >
<!ATTLIST position  x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >
<!ATTLIST target    x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >
```

```
<!ATTLIST up_vector  x CDATA #REQUIRED
                     y CDATA #REQUIRED
                     z CDATA #REQUIRED >
<!ATTLIST translation x CDATA #REQUIRED
                      y CDATA #REQUIRED
                      z CDATA #REQUIRED >
<!ATTLIST scale x CDATA #REQUIRED
                y CDATA #REQUIRED
                z CDATA #REQUIRED >
<!ATTLIST rotation axis CDATA #REQUIRED
                    radians CDATA #REQUIRED>
<!ATTLIST matrix m00 CDATA #REQUIRED
                 m01 CDATA #REQUIRED
                 m02 CDATA #REQUIRED
                 m03 CDATA #REQUIRED
                 m10 CDATA #REQUIRED
                 m11 CDATA #REQUIRED
                 m12 CDATA #REQUIRED
                 m13 CDATA #REQUIRED
                 m20 CDATA #REQUIRED
                 m21 CDATA #REQUIRED
                 m22 CDATA #REQUIRED
                 m23 CDATA #REQUIRED
                 m30 CDATA #REQUIRED
                 m31 CDATA #REQUIRED
                 m32 CDATA #REQUIRED
                 m33 CDATA #REQUIRED>
<!ATTLIST bezier description CDATA #REQUIRED >
<!ATTLIST ponto1    x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >

<!ATTLIST ponto2    x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >
<!ATTLIST ponto3    x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >
<!ATTLIST ponto4    x CDATA #REQUIRED
                    y CDATA #REQUIRED
                    z CDATA #REQUIRED >
<!ATTLIST sphere description CDATA #REQUIRED >
<!ATTLIST cylinder description CDATA #REQUIRED >
<!ATTLIST radius value CDATA #REQUIRED >
<!ATTLIST height value CDATA #REQUIRED >
<!ATTLIST material  r CDATA #REQUIRED
                    g CDATA #REQUIRED
                    b CDATA #REQUIRED >
<!ATTLIST meshobject filename CDATA #REQUIRED
                      type CDATA #REQUIRED
                      description CDATA #REQUIRED>
<!ATTLIST transform description CDATA #REQUIRED>
<!ATTLIST axis  x CDATA #REQUIRED
                y CDATA #REQUIRED
                z CDATA #REQUIRED >
<!ATTLIST range min CDATA #REQUIRED
                max CDATA #REQUIRED
                rest CDATA #REQUIRED>
<!ATTLIST directionallight  description CDATA #REQUIRED>
<!ATTLIST spotlight  description CDATA #REQUIRED>
<!ATTLIST pointlight  description CDATA #REQUIRED>
```

```
<!ATTLIST intensity value CDATA #REQUIRED>
<!ATTLIST ambientIntensity value CDATA #REQUIRED>
<!ATTLIST color red CDATA #REQUIRED
                green CDATA #REQUIRED
                blue CDATA #REQUIRED
                alpha CDATA #REQUIRED>
<!ATTLIST enabled value CDATA #REQUIRED>
<!ATTLIST attenuation x CDATA #REQUIRED
                      y CDATA #REQUIRED
                      z CDATA #REQUIRED>
<!ATTLIST beam_Width value CDATA #REQUIRED>
<!ATTLIST cut_Off_Angle value CDATA #REQUIRED>
```

Quadro 15 – DTD do arquivo XML das juntas

ANEXO B – DTD completa do arquivo XML das animações

No Quadro 16 encontra-se a DTD completa do arquivo XML das animações do V-Art.

```
<!ELEMENT action ( joint_movement+ ) >
<!ATTLIST action action_name  CDATA #REQUIRED
                 speed        CDATA #REQUIRED
                 cycle        CDATA #REQUIRED>
<!ELEMENT joint_movement (interpolation, dof_movement+)>
<!ATTLIST joint_movement joint_name     CDATA #REQUIRED
                         duration       CDATA #REQUIRED>
<!ELEMENT interpolation (noise?, persistency?, overshoot?, offset?)>
<!ATTLIST interpolation type  CDATA #REQUIRED>
<!ELEMENT noise EMPTY>
<!ATTLIST noise value  CDATA #REQUIRED>
<!ELEMENT persistency EMPTY>
<!ATTLIST persistency value  CDATA #REQUIRED>
<!ELEMENT overshoot EMPTY>
<!ATTLIST overshoot value  CDATA #REQUIRED>
<!ELEMENT offset EMPTY>
<!ATTLIST offset value  CDATA #REQUIRED>
<!ELEMENT dof_movement EMPTY>
<!ATTLIST dof_movement dofID            CDATA #REQUIRED
                       initialTime      CDATA #REQUIRED
                       finalTime        CDATA #REQUIRED
                       finalPosition    CDATA #REQUIRED>
```

Quadro 16 – DTD do arquivo XML das animações