"MARCOS ANTONIO ANDRADE NOBRE"

"INTEGRAÇÃO, CONVERSÃO E OTIMIZAÇÃO EM SISTEMA DE REALIDADE VIRTUAL PARA MANUTENÇÃO EM UNIDADES GERADORAS "

São Paulo
2011

"MARCOS ANTONIO ANDRADE NOBRE"

"INTEGRAÇÃO, CONVERSÃO E OTIMIZAÇÃO EM SISTEMA DE REALIDADE VIRTUAL PARA MANUTENÇÃO EM UNIDADES GERADORAS "

Monografia apresentada na disciplina Projeto de Formatura II para a conclusão do curso de Engenharia Elétrica na Escola Politécnica da Universidade de São Paulo

Área de Concentração:
Eng. Elétrica - Sistemas Eletrônicos

Orientador: Mestre Fábio Luis Doreto Rodrigues

São Paulo
2011

DEDICATÓRIA

Dedico este trabalho aos meus pais Antonio e Judite Maria Andrade Nobre.

AGRADECIMENTOS

Ao orientador e amigo Mestre em Engenharia Fabio Luiz Doreto Rodrigues, que com sua experiência e apoio foi fundamental para conclusão desta monografia

Ao Prof. Dr. Marcelo Knörich Zuffo, coordenador dos Meios Eletrônicos Interativos do Laboratório de Sistemas Integráveis da USP, pelo apoio e incentivo neste intenso momento de conclusão de curso.

À minha família e à minha namorada Viviane, que nunca me faltaram nos momentos mais difíceis da minha vida e sempre me deram apoio incondicional.

A equipe da Caverna Digital, em especial a Olavo Belloc, Douglas dos Santos, Mário Nagamura e Luis Fernando Paulucci que trabalharam arduamente para que este projeto tivesse êxito.

"De erro em erro vai se descobrindo toda verdade".

(Sigmund Freud)

RESUMO

Este monografia apresenta o trabalho de integração dos vários módulos dentro de um sistema de visualização em Realidade Virtual, a partir disso, discutem-se as necessidades e vantagens da investigação deste processo.

Este trabalho documenta um projeto do Laboratório de Sistemas Integráveis e apresenta o trabalho de projeto de formatura que ocorreu dentro do escopo deste projeto.

ABSTRACT

This monograph presents the work of integrating the various modules inside a visualization system in Virtual Reality, as it discusses the research needs and advantages of this process.

This work documents an Integrated System's Laboratoy's project and presents an undergraduate dissertation that was developed inside this project scope.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CAD	*Computer-Aided Design*
ISO	*International Organization for Standardization*
LSI	*Laboratório de Sistemas Integráveis*
P&D	*Pesquisa e Desenvolvimento*
PNML	*Petri-Net Markup Language*
RV	*Realidade Virtual*
UG	*Unidade Geradora*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

SUMÁRIO

1 Introdução

O uso da Realidade Virtual (RV) para treinamento, capacitação e simulação é largamente documentado na literatura acadêmica.

Diversos fatores são responsáveis pelo aumento do uso deste tipo de ferramenta, dentre elas podem se destacar: A complexidade de determinadas tarefas, que demandam intenso treinamento, seja pela delicadeza do processo (o caso de determinados procedimentos cirúrgicos), seja pelo risco humano ou materiais associados, bem como limitações temporais, geográficas ou financeiros, apontam para uma demanda crescente da opção de desenvolvimento de simuladores [1] [2].

Ultimamente, os constantes avanços tecnológicos desta área, a popularização de periféricos, e conseqüentemente, a redução dos custos dos equipamentos, têm viabilizado o uso cada vez maior das ferramentas de Realidade Virtual na elaboração de simuladores e aplicações de treinamento [3] [4].

O sistema de visualização discutido neste projeto foi proposto com o objetivo de facilitar o entendimento das atividades e dos procedimentos envolvidos na manutenção de Unidades Geradoras (UG) de uma concessionária de energia elétrica. O software será utilizado por equipes de manutenção com o objetivo principal de disseminar os conhecimentos envolvidos nas suas atividades e analisar procedimentos de trabalho, ocorrências e falhas.

Para possibilitar que os usuários consigam desenvolver com maior facilidade o aprendizado na área de manutenção, todas as peças que compõem a unidade geradora poderão ser visualizadas com as suas características.

Uma ferramenta gráfica de visualização e treinamento em realidade virtual, necessariamente trabalhará, de forma que, a interoperabilidade entre ferramentas de aprendizagem, motores gráficos, estruturas de modelagem de procedimentos, trabalhem adequadamente em paralelo e de maneira otimizada, o que implicará num esforço de integração e otimização para a convergência de todas as informações.

A base do processo de integração e otimização está inserida dentro do contexto do trabalho de gerenciar diferentes formatos de arquivos e tecnologias, portanto o

processo de conversão de arquivos é o cerne do processo de desenvolvimento de ferramentas de visualização para simulação e treinamento.

Para o desenvolvimento deste trabalho foram utilizadas técnicas modernas de engenharia de software, priorizando sempre que possível o uso de ferramentas distribuídas gratuitamente e do código aberto.

2 Objetivos

Este trabalho de projeto de formatura tem como objetivo documentar o trabalho executado dentro do escopo do projeto de pesquisa e desenvolvimento, estabelecido no Laboratório de Sistemas Integráveis (LSI) da Escola Politécnica da Universidade de São Paulo.

Para o desenvolvimento de um sistema de visualização e treinamento utilizando realidade virtual, trabalhamos com diversas camadas de tecnologia que convergem para uma única solução que é o sistema de visualização.

Uma das especificações definidas neste projeto foi à escolha de computadores e periféricos comerciais que implicam imediatamente em determinadas limitações de capacidade computacional.

A utilização de computadores convencionais e suas restrições são o ponto de partida para o trabalho de otimização dos objetos 3D, visto que dentro de um sistema visualização é a etapa onde o processamento computacional de imagens é o de maior custo dentro do processamento total.

Assim, o objetivo técnico do trabalho de conclusão de curso é o desenvolvimento e testes de ferramentas de integração entre softwares de produtividade, algoritmos de modelagem de procedimentos e sistemas de treinamento em realidade virtual, dentre os quais podemos destacar:

- Integração de modelos 3D polígonos entre o INVENTOR e o 3D MAX;
- Integração dos modelos do 3D MAX para adequação das estruturas da Rede de Petri;
- Nomenclatura dos modelos, dentro do software de visualização;

- Definição e montagem das entidades XML.

Estas etapas dentro do projeto de formatura são essenciais para o desenvolvimento de um sistema de visualização que permita a interoperabilidade entre as diversas etapas citadas acima.

3 Escopo do Projeto

O projeto de P&D, no qual esta monografia se insere, é o desenvolvimento de uma aplicação de treinamento que possa ser executada em um sistema de RV, com tecnologia de projeção estereoscópica, usando computadores convencionais, visando reduzir o investimento necessário para a implantação de uma infraestrutura de treinamento e visualização.

Como parte do escopo deste projeto, o trabalho de conclusão de curso visa a integração entre diferentes módulos de software e entre ferramentas de produtividade e softwares de interação e visualização.

O sistema de RV proposto pelo projeto deve permitir o treinamento de manutenção de unidade geradora de forma que os técnicos possam se familiarizar com as diversas peças e as formas de montagem deste equipamento de grandes dimensões.

A Figura 1 apresenta um possível caso de uso da aplicação, que envolve a criação de uma sala de visualização, onde as instruções de manutenção são apresentadas e discutidas pela equipe de técnicos e engenheiros. Neste cenário, todos os presentes podem acompanhar o que está sendo apresentado nos seus próprios terminais.

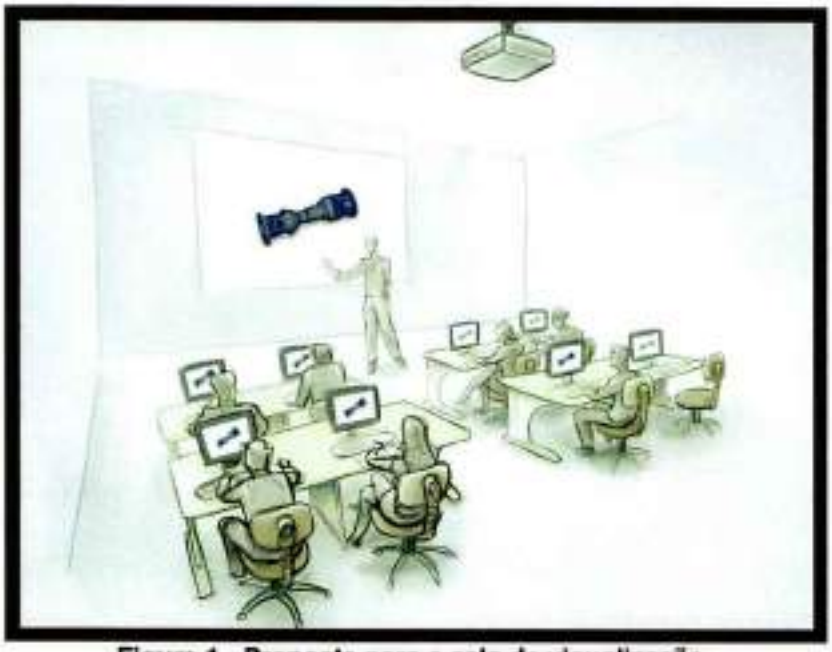

Figura 1 - Proposta para a sala de visualização.

A lista abaixo apresenta os principais requisitos utilizados para o desenvolvimento do software:

- O software deve ilustrar o procedimento de desmontagem/montagem da unidade geradora, dando ênfase na ordem das atividades que precisam ser praticadas pela equipe;
- Quando requisitado pelo usuário, o software deverá disponibilizar informações técnicas sobre as peças da UG;
- Deve rodar em computadores convencionais;
- Pode utilizar mecanismos de interação convencionais, como mouse;
- O software será desenvolvido para o Sistema Operacional Windows da Microsoft;
- O desenvolvimento da ferramenta deverá ser realizado de forma independente do Inventor da Autodesk, permitindo que o mesmo seja executado em máquinas que não tenham o Inventor instalado, evitando a aquisição de licenças deste software.

3.1 Descrição do Sistema

A solução completa de sistema de treinamento é composta de quatro módulos conforme vemos na Figura 2:

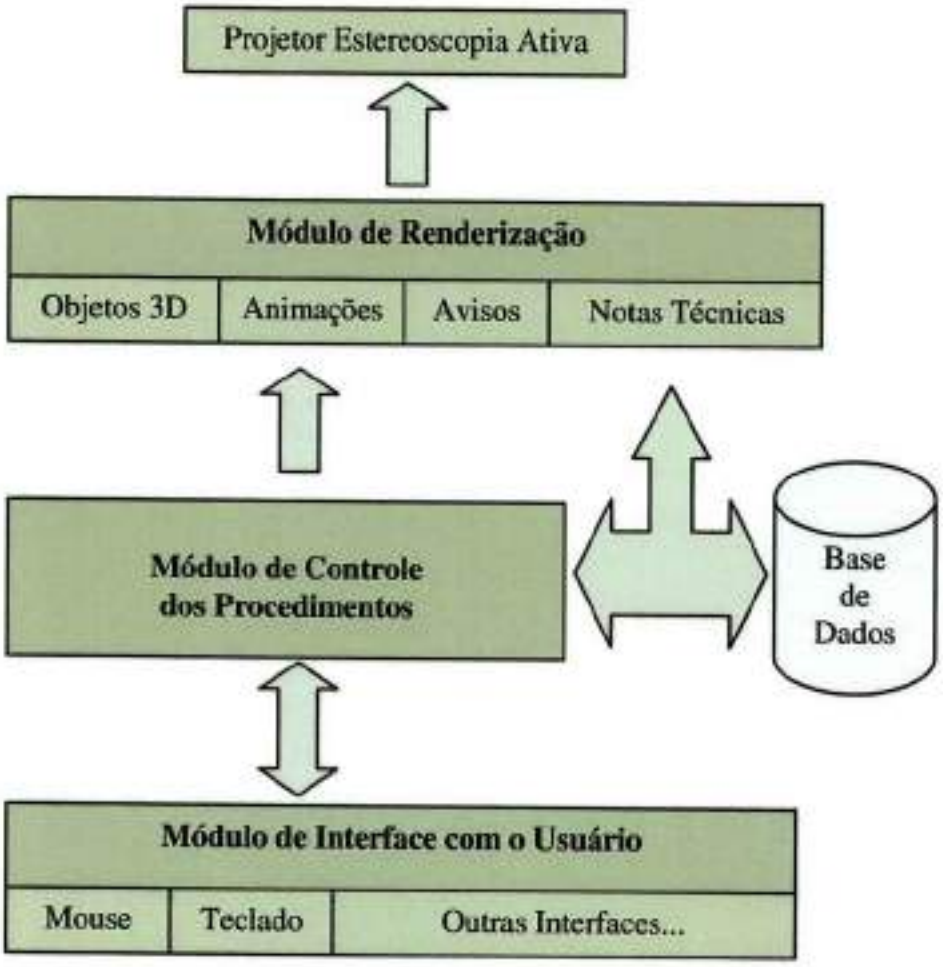

Figura 2 - Diagrama da aplicação

A seguir cada módulo é explicado brevemente.

3.1.1 Módulo de apresentação

O Módulo de apresentação é responsável por disponibilizar todo o conteúdo da ferramenta para o usuário. Este conteúdo pode ser classificado como objetos 3D, imagens, textos e elementos da interface gráfica, como botões, menus, caixas de diálogo, entre outros.

O Módulo de apresentação foi desenvolvido com base na biblioteca de *software* Ogre [10] e utiliza os seus recursos para montar o cenário virtual e renderizar objetos 3D em tempo-real. Além disto, para apresentar a interface gráfica, este módulo utiliza a biblioteca Hikari [11].

O Hikari é uma biblioteca de *software* que possui recursos para interpretar e executar arquivos em Adobe Flash. Além de ser integralmente compatível com o Ogre, a tecnologia Adobe Flash permite que interfaces gráficas sejam criadas de forma simples e fácil, contendo recursos para apresentar imagens e vídeos no formato FLV.

O Módulo de apresentação recebe comandos do Módulo de controle dos procedimentos para alterar as características dos modelos geométricos e outras informações apresentadas pela ferramenta.

3.1.2 Módulo de controle dos procedimentos

O Módulo de controle dos procedimentos é utilizado para auxiliar o usuário da ferramenta a compreender as etapas envolvidas em uma determinada atividade de manutenção.

Sendo assim, este módulo é responsável por coordenar a atividade de manutenção que está sendo visualizada na ferramenta, avaliando as ações executadas pelo usuário no cenário virtual e verificando se estas ações estão de acordo com a descrição do procedimento de manutenção.

As ações do usuário são recebidas pelo módulo de Interface, sendo posteriormente processadas e avaliadas. Estas ações são comparadas com a descrição dos procedimentos e os resultados desta comparação são enviados ao Módulo de apresentação.

Após receber o resultado, o Módulo de apresentação atualiza as informações presentes na tela, informando o usuário se o mesmo agiu corretamente ou cometeu um erro.

Sendo assim, o Módulo de controle deve interagir com os Módulos de apresentação e de interface com o usuário, além de acessar e manipular a descrição dos procedimentos na Base de dados.

Este módulo de controle foi desenvolvido integralmente pelo laboratório e utiliza diagramas de redes de Petri para interpretar as relações de dependência existentes entre as diferentes etapas de um determinado procedimento.

As redes de Petri descrevem como os elementos presentes no cenário virtual devem ser manipulados para executar corretamente as atividades de manutenção previamente estabelecidas.

3.1.3 Módulo de interação com o usuário

O Módulo de interação com o usuário permite com que os objetos presentes na cena virtual sejam facilmente selecionados e manipulados através de botões, menus e caixas de diálogo.

Para facilitar a utilização da ferramenta e reduzir os seus custos de implantação, foi determinado que a mesma utilizasse apenas periféricos convencionais de interação, como teclado e mouse, evitando equipamentos sofisticados como luvas, sensores de posição, etc.

Este módulo é responsável por identificar os objetos que o usuário deseja manipular e enviar as opções selecionadas para o Módulo de controle.

3.1.4 Base de dados

A Base de dados é composta por um conjunto de arquivos no padrão XML e PNML que contêm a descrição de todos os elementos interativos do cenário virtual e a declaração das atividades de manutenção.

Nesta primeira versão da ferramenta, apenas as atividades de montagem e desmontagem da Unidade Geradora foram incluídas.

Esta base de dados é acessada diretamente por dois módulos da aplicação: o Módulo de apresentação e o Módulo de controle dos procedimentos. O Módulo de apresentação consulta apenas as informações geométricas dos modelos e seus respectivos materiais, enquanto que o Módulo de controle obtém todas as informações referentes aos processos de montagem e desmontagem da UG.

3.1.5 Infraestrutura

O software de treinamento foi planejado e desenvolvido para ser executado em um computador comum (PC), ou em um sistema de realidade virtual. Desta forma, a aplicação pode tirar proveito dos recursos de estereoscopia ativa presentes na infraestrutura de um sistema de projeção com projetores de 120 hz e computadores

com placas gráficas que gerem estereoscopia, bem como, também ser executada em uma infraestrutura simples, contendo apenas um monitor ou projetor convencional.

A iniciativa de reduzir os custos de sistemas de Realidade Virtual acompanha o esforço da comunidade científica de computação gráfica, que têm apresentado avanços significativos nesta área [12] [13] [14].

3.1.6 Descrição da UG

A Unidade Geradora (UG) é representada no sistema por um grupo de modelos 3D obtidos a partir do CAD 3D da unidade geradora real. As diversas peças são encaixadas virtualmente da mesma forma na qual são encaixadas na unidade geradora real.

A descrição das peças que compõem a UG são descritas através da estrutura de entidades. Já o procedimento é modelado utilizando redes de Petri.

3.1.7 Entidades

As entidades são estruturas de dados que descrevem os elementos que compõem a aplicação de visualização. Exemplos de entidades são as peças da UG. As entidades são compostas por Propriedades e Ações, ambos descritos em um arquivo XML.

As Propriedades são elementos nomeados que armazenam informações de características e do estado de uma entidade.

As Ações descrevem todas as formas de interação e manipulação da entidade. Desta forma, a interação do usuário com as peças da UG é através das Ações disponíveis nas entidades que as representam.

As ações são descritas através de seqüências de comandos. Quando uma ação é acionada, os comandos são executados. Cada comando executa uma tarefa específica como animações, alteração de propriedades, alterações de geometria, acionamento de caixas de diálogos, entre outras.

3.1.8 Seleção de objetos gráficos

Durante a execução da aplicação de visualização, o usuário interage com as peças da UG selecionando-as e executando ações sobre elas.

A seleção de uma entidade gráfica ocorre quando o usuário posiciona o mouse sobre o objeto 3D que a representa, alterando sua cor para indicar a seleção. Como exemplo, a Figura 3 mostra a seleção do *Mancal Guia e Escora do Gerador* (destacado em verde).

Ao clicar sobre qualquer entidade que esteja selecionada, a aplicação exibirá um menu de contexto, contendo todas as opções e ações disponíveis ao usuário, para manipular e interagir com esta entidade, conforme a descrição da entidade através do arquivo XML.

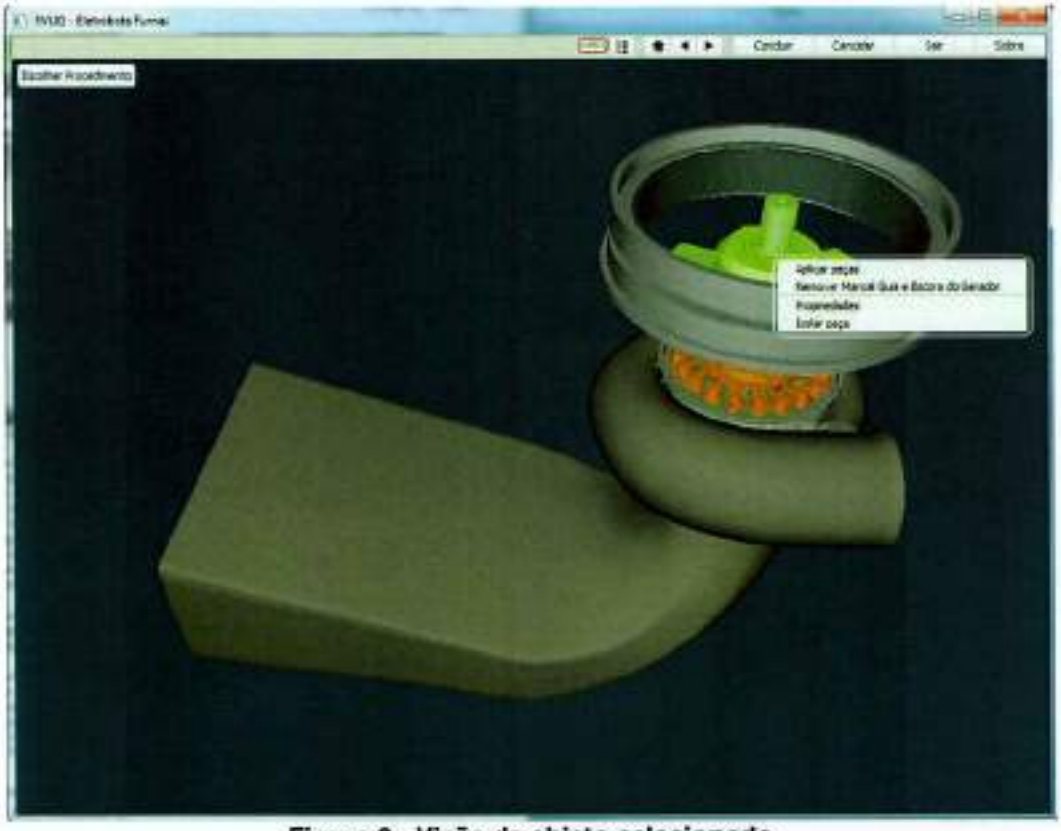

Figura 3 - Visão de objeto selecionado

3.1.9 Arquivo de configuração

Nesta seção, será apresentada a sintaxe XML que contém a definição das entidades. Cada peça da Unidade Geradora é representada por uma entidade gráfica. Cada entidade contém um conjunto de propriedades e um conjunto de ações.

Propriedades: Armazenam informações a respeito das características e do estado interno da entidade;

Ações: Define as possíveis ações que o usuário poderá executar sobre uma determinada entidade;

Para exemplificar a declaração de uma entidade, considere o arquivo anel_escora.xml, apresentado na Tabela .

Neste arquivo, a declaração de uma entidade é dividida em duas seções: A primeira é determinada pela *tag* <properties> e a segunda é determinada pela *tag* <actions>. A *tag* <properties> contém a declaração das propriedades e a *tag* <actions> contém a definição das ações dos objetos ao serem manipulados pelo usuário.

As propriedades são identificadas pelo seu nome, e os valores configurados neste arquivo determinam o estado da peça, se ela está inserida ou não na cena, se ela pode ser selecionada (irá ficar destacada em verde), se possui informações técnicas associadas, entre outras características.

```
<graphical_entity name="turbina">
  <properties>
            <property name="description" value="Turbina"/>
      <property name="icon" value="resources/icons/Turbina.png"/>
            <property name="selectable" value="true"/>
            <property name="attached" value="true"/>
            <property name="state" value="insert"/>
       <property name="information" value="entities/info/info_turbina.xml" />
  </properties>
  <actions>
            <action name="apply" description="Aplicar peças">
                  <availability>
                        <return value="true"/>
                  </availability>
                  <execution>
                        <flash_call_command flash_entity="main_menu"
function="show_selection_menu">
                              <parameter name="action" value="insert" />
                              <parameter name="entities"
value="vedacao_do_mancal_guia, ... " />
                        </flash_call_command>
                  </execution>
            </action>
      :
  </actions>
</graphical_entity>
```

Tabela 1 - Exemplo de descrição de uma Entidade, no formato XML

As ações são definidas através de comandos. Os comandos possuem funções específicas e são utilizados para definir a seqüência de reações de uma interação com o usuário.

Cada comando é representado por uma tag diferente e possui os seus próprios atributos. Estes comandos são utilizados para definir os dois principais elementos de uma ação: a sua disponibilidade e a sua execução, representados pelas tags *availability* e *execution*, respectivamente.

O elemento representado pela *tag* <availability> é usado para identificar se a ação em questão está disponível ou não para o usuário. Para verificar se a ação está disponível, todos os comandos deste elemento serão executados seqüencialmente. Caso o retorno destes comandos for verdadeiro (*true*), a ação estará disponível. Caso contrário, a ação não estará disponível. Normalmente são utilizados comandos de verificação de condições.

Já o elemento representado pela *tag* <execution> é utilizado para definir as reações de cada ação. Estas reações podem incluir a execução de animações, a alteração da visibilidade do objeto, ou a apresentação de elementos da interface gráfica, como menus, caixa de diálogo, entre outros.

Na Tabela , a definição da entidade turbina é apresentada de forma simplificada. Nesta listagem, apenas a ação *apply* é demonstrada. Ambas as *tags* <availability> e <execution> desta ação possuem um único comando.

Para maiores informações sobre o significado das propriedades e dos comandos existentes para a composição das ações, verifique o **Apêndice A** deste documento.

3.1.10 Procedimentos

O procedimento consiste na definição da ordem em que as etapas de uma determinada atividade precisam ser executadas. Esta ordem é definida pela sequência de ações que o usuário precisa executar em cada objeto da cena.

Para definir a sequência de ações de um determinado procedimento, o modelo de rede de Petri foi escolhido. Embora este modelo seja baseado em conceitos simples, o mesmo pode ser utilizado para descrever diversos tipos de relações de dependência.

Uma rede de Petri contém três elementos principais: arcos direcionados (*directed arcs*), posições (*places*) e transições (*transitions*).

Cada transição de uma rede de Petri representa uma ação de um determinado objeto. As posições da rede são utilizadas para especificar as pré-condições e os efeitos da execução de cada transição.

Desta forma, uma ação de um determinado objeto só será considerada correta quando a transição associada com esta ação estiver disponível na rede de Petri. Caso o usuário tente executar uma ação cuja transição não está disponível na rede, a ferramenta de visualização apresentará uma mensagem de erro. Caso contrário, a ação será executada e o estado da rede de Petri será atualizado.

Sendo assim, as conexões existentes entre as transições e as posições de uma determinada rede definem as relações de dependência existentes entre as ações dos objetos.

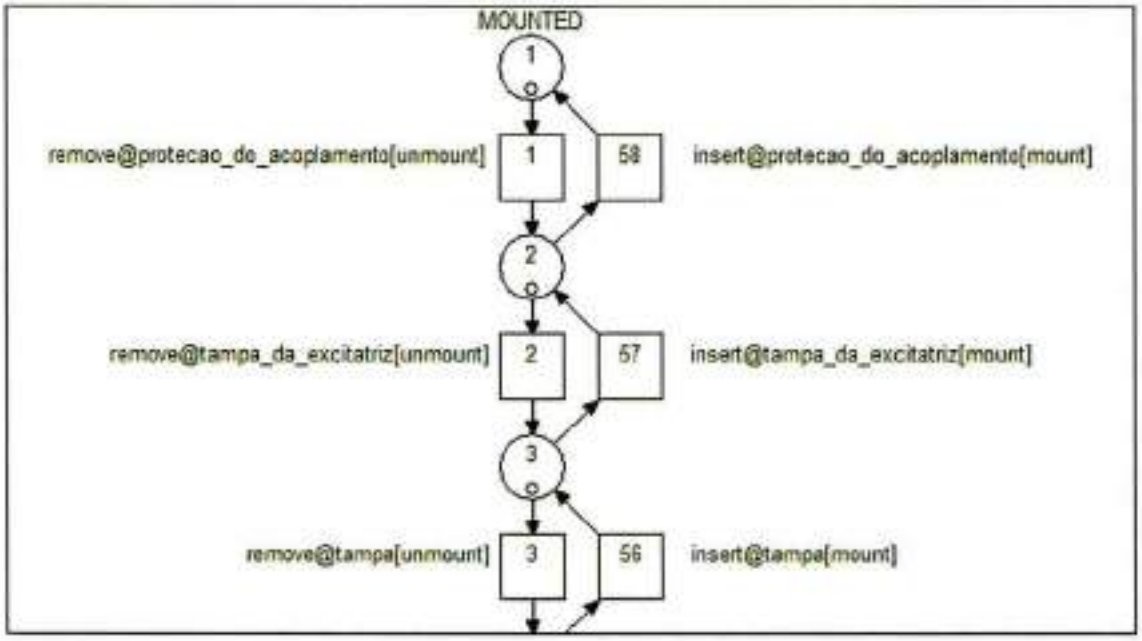

Figura 4 – Exemplo de rede de Petri

A Figura 4 ilustra parcialmente a rede de Petri do procedimento de montagem e desmontagem da Unidade Geradora.

Esta rede de Petri foi elaborada no programa Netlab, distribuído gratuitamente. Este programa possui uma interface gráfica amigável que facilita e simplifica a edição destas redes. Além disto, este programa ainda possui recursos para testar e validar o procedimento elaborado, antes de inseri-lo na ferramenta de visualização.

Para maior compatibilidade com editores gráficos como o Netlab, a aplicação de visualização carrega redes de Petri em arquivos no formato PNML (*Petri-Net Markup Language*). Este formato de arquivo é padrão ISO (International Organization for Standardization) e também possui sintaxe XML, facilitando o entendimento e a manutenção.

Para aplicação de visualização associar o procedimento descrito em redes de Petri com ações em entidades, o nome das transições deve obedecer a seguinte sintaxe: "*nome_da_ação@nome_da_entidade[nome_do_procedimento]*".

O nome da ação e o nome da entidade mencionado na transição da rede devem ser os mesmos nomes mencionados no arquivo de configuração XML da entidade.

No exemplo da Figura 4, caso o usuário escolha realizar o procedimento de desmontagem da Unidade Geradora, o marcador será colocado na posição de nome *MOUNTED*. Neste caso, o marcador identifica que a Unidade Geradora está montada, e o usuário deve selecionar objetos e escolher ações para desmontar a UG.

De acordo com a rede de Petri apresentada, apenas uma única ação ficará disponível neste momento: A ação *remove* do objeto que representa a proteção do acoplamento. Se o usuário escolher esta ação, a mesma será aprovada e, conseqüentemente, executada pela aplicação. Caso contrário, a ferramenta de visualização informará ao usuário de que o mesmo escolheu uma ação que não corresponde com o procedimento.

No próximo passo, após executar a ação correta, o marcador será retirado da posição de nome *MOUNTED*, e será colocado na posição de número 2. Neste momento, duas transições estarão disponíveis para serem executadas.

A primeira transição disponível corresponde à ação *remove* da tampa da excitatriz, e promove o avanço do procedimento, enquanto que a segunda transição corresponde à ação *insert* da proteção do acoplamento, que promove o retrocesso do procedimento.

Para utilizar a rede de Petri, é necessário usar os comandos apropriados ao definir as ações dos objetos. Estes comandos servem para inserir marcadores nas

posições da rede, verificar os estados das transições e disparar as transições. Para consultar os comandos disponíveis, consulte o **Apêndice A** deste documento.

4 Implementação

Neste capítulo será apresentada a implementação das integrações, peça chave do projeto e foco do trabalho de conclusão de curso.

4.1 Tarefas

As tarefas executadas dentro do escopo do projeto de P&D foram:

- **Integração de modelos 3D polígonos entre o INVENTOR e o 3D MAX**: esta tarefa consiste na simplificação do modelo CAD da unidade geradora para compatibilidade com software de visualização
- **Integração dos modelos do 3D MAX para adequação das estruturas da Rede de Petri:** As entidades definidas como posições da rede de Petri estão diretamente relacionadas com os conjunto de geometrias presentes nos modelos do 3D MAX, assim como as entidades definidas como transições na rede de Petri são relacionadas com as animações dos modelos.
- **Nomenclatura dos modelos, dentro do software de visualização:** existe a necessidade de que os nomes referenciados dentro do software de visualização sejam coerentes com as entidades da rede de Petri para o correto funcionamento do procedimento.
- **Definição e montagem das entidades XML:** implementação dos comandos e propriedades que permite a interface entre o usuário e o grafo de cena.

4.1.1 Cronograma

A Tabela 2 apresenta o cronograma de desenvolvimento das tarefas do trabalho de conclusão de curso:

Tabela 2 - Cronograma das Etapas .

Tarefa / Mês	1-2	3-4	5-6	7-8	9-10	11-12
Integração Inventor 3D Max						
Integração 3D Max Rede de Petri						
Nomenclatura de Modelos						
Definição entidades XML						
Documentação do Projeto						

4.2 Integração entre Software de Produtividade e Software de Visualização

Na primeira etapa, foi feita a validação do procedimento de conversão e visualização dos modelos.

Um dos principais trabalhos desta etapa consistiu em converter os modelos de CAD, elaborados no Inventor, para malhas poligonais. Para realizar este trabalho foi utilizado o *software* de modelagem 3D Studio Max, da Autodesk.

Durante o processo de importação e conversão, as principais características geométricas dos modelos originais foram mantidas. Outras características dos modelos de CAD, que não seriam utilizadas na visualização em tempo-real, foram descartadas no processo de conversão.

Após o procedimento de conversão, realizado de forma automatizada pelo *software* 3DS Max, o modelo bruto da UG apresentava dois problemas:

- Modelos poligonais replicados desnecessariamente (sem instâncias);
- Objetos poligonais contendo vértices desnecessários, principalmente na junção entre duas superfícies adjacentes;

Para corrigir estes problemas, foi desenvolvido um *script* a ser utilizado dentro do software de modelagem 3D Studio Max. O *script* foi desenvolvido utilizando a linguagem de programação do próprio software, o MAXScript.

Ao final do processo de conversão, o resultado obtido continha um modelo com aproximadamente quatro milhões de polígonos. A partir deste modelo, foi iniciado o processo de desenvolvimento da primeira versão do visualizador.

Na primeira etapa, o principal objetivo da aplicação era viabilizar a manipulação e a visualização interativa (tempo-real) do modelo convertido. Para isto, diversas bibliotecas de software disponíveis gratuitamente e de código aberto foram consideradas.

4.2.1 Problema do uso de arquivos CAD

O requisito de uma ferramenta utilizando computadores convencionais, tornou extremamente necessário a otimização dos modelos obtidos do Inventor.

Diferente de projetos anteriores, onde o trabalho de desenvolver o simulador envolvia a modelagem total dos objetos, neste projeto, pela complexidade foram fornecidos os arquivos com os modelos da UG desenvolvidos na ferramenta de CAD Autodesk Inventor.

A partir daí, elimina-se o problema de modelar as peças da UG, por outro lado, a finalidade do Inventor como ferramenta de CAD, tem como efeito, modelos com um número muito elevado de detalhes, que são necessários para a finalidade de projeto de uma peça, mas que para a utilização em um ferramenta de visualização acaba por elevando o custo de processamento necessário para as tarefas de renderização o que inviabilizaria a utilização de computadores convencionais.

Desta forma, na primeira etapa do projeto iniciou-se um esforço para eliminar detalhes dos arquivos do Inventor exportados para o 3D Max, que permitiu os primeiros testes do visualizador.

Já na segunda etapa que é o escopo desta monografia, foi feito um trabalho de reconstrução e otimização dos scripts, para que fosse reduzido o número de polígonos gerados e consequentemente diminuírem o custo de processamento.

Também foram desenvolvidos componentes para garantir a coerência dos modelos, uma vez que naturalmente no processo de conversão de arquivos, algumas geometrias que eram únicas no Inventor, se apresentavam separadas no 3D Max.

A seguir vamos descrever um passo a passo da conversão e execução dos scripts, que torna a percepção do trabalho de conversão mais didática.

No Apêndice B, pode-se ver a listagem do script desenvolvido no 3D Max que permitiu a execução destas tarefas e é o código por trás das interfaces gráficas vistas a seguir.

4.2.2 Procedimento de Conversão e Adaptação dos Modelos do Inventor

Este procedimento tem o objetivo de converter os objetos CAD da UG modelados na ferramenta Inventor para o formato nativo da aplicação de visualização. O procedimento de conversão é realizado no software de modelagem 3D Studio Max, da Autodesk. Além desta aplicação, também são utilizados outros dois recursos: Um script elaborado em MAXScript para aprimorar e otimizar os modelos importados do Inventor e o plugin OgreMax, para exportar o modelo do 3D Studio Max para o formato de arquivo nativo da aplicação de visualização.

O procedimento de conversão e adaptação dos modelos contém as seguintes etapas:

1. Instalação do software 3D Studio Max;
2. Instalação do plug-in OgreMax;
3. Importar os modelos do Inventor para o 3D Studio Max;
4. Aperfeiçoar os modelos utilizando o script "Inventor Post Production";
5. Aplicar materiais e animações; e,
6. Exportar os modelos utilizando o plug-in OgreMax.

Cada etapa do procedimento será explicada em detalhe nós próximos tópicos deste documento.

4.2.3 Instalação do software 3D Studio Max.

O primeiro passo consiste em instalar o software 3D Studio Max, da Autodesk. Esta ferramenta possui recursos para criar e manipular objetos 3D representados em

polígonos e, portanto, permite a criação e transformação de conteúdo para visualização em tempo-real.

O software utilizado foi o 3D Studio Max 2009 64-bits, instalado no Windows Vista 64-bits. O procedimento de instalação é simples e as opções pré-selecionadas pelo instalador são suficientes para realizar o procedimento de conversão.

Caso seja necessário converter todo o conjunto da Unidade Geradora (UG), recomenda-se a utilização de um computador de alto desempenho, com memória RAM acima de 4GB e instalação da versão 64-bits do Windows e do 3D Studio Max, para garantir a utilização da memória instalada.

4.2.4 Instalação do plug-in OgreMax.

Após instalar o 3D Studio Max, será necessário instalar o plug-in OgreMax. Este plugin exporta os modelos elaborados ou importados no 3D Studio Max para formato nativo da aplicação de visualização.

O plugin OgreMax pode ser obtido gratuitamente através do site http://www.ogremax.com/downloads e não precisa de licença para ser utilizado. A versão utilizada no projeto foi a 1.6.23. É necessário instalar também o *Visual C++ 2008 Runtime Libraries* e o *DirectX Runtime*, disponíveis na mesma página.

O processo de instalação pode ser feito em apenas um passo. Após executar o arquivo de instalação do OgreMax, abrirá uma tela de opções, como ilustrado na figura abaixo.

Figura 5 – Menu de instalação do OgreMax.

Caso esteja disponível, altere a opção *Ogre 3D Version* para a versão *1.6.x* ou para a versão *1.7.x*. Mantenha todas as outras opções com os valores pré-estabelecidos. Clique no botão "Install" e escolha o local de instalação.

4.2.5 Importar os modelos do Inventor para o 3D Studio Max.

Para importar os modelos do Inventor para o 3D Studio Max, foi utilizado um *importer* disponível automaticamente se uma cópia do software Inventor estiver instalada.

Para Importar os modelos, acesse a opção do menu "*File > Import...*", como ilustrado na figura abaixo:

Figura 6 – Menu de importação do 3D Studio Max.

A Figura 7 mostra a janela de opções apresentada pelo *importer* do 3D Studio Max. Estas opções permitem alterar as seguintes características:

- *Mesh Resolution*: Controla o nível de tecelagem das curvas paramétricas - Quantidade de polígonos que serão utilizados na conversão;
- *Material*: Configura a importação dos materiais do Inventor. Apenas algumas características do material são importadas pelo 3D Studio Max. Esta opção não funcionou de forma adequada, e materiais importados não puderam ser utilizados no 3D Studio Max.
- *Merge/Replace*: Insere os objetos importados na cena atual ou substitui com uma nova cena.

Através do controle *Mesh Resolution* é possível ajustar a quantidade de polígonos que serão utilizados para representar as superfícies paramétricas dos objetos importados.

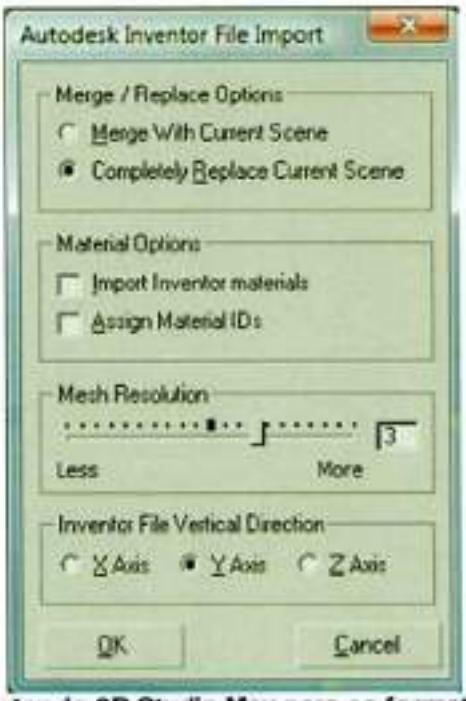

Figura 7 – Importer do 3D Studio Max para os formatos do Inventor.

Quanto maior a resolução, maior será a qualidade da malha gerada, porém maior será a quantidade de polígonos utilizada na tecelagem, podendo resultar em baixa performance durante a visualização em tempo real na aplicação de visualização. Para o caso da UG, valores entre 60% e 70% apresenta boa qualidade, sem comprometer a interatividade com o modelo.

As outras opções podem ser mantidas como apresentadas na Figura 7. Após pressionar o botão *Ok*, os modelos selecionados do Inventor serão importados para o 3D Studio Max.

4.2.6 Aperfeiçoar os modelos utilizando o script "Inventor Post Production"

Devido ao processo de geração de malhas poligonais a partir das superfícies paramétricas durante o processo de importação, os objetos apresentam muitos vértices coincidentes ou muito próximos no espaço 3D, apresentando resultados incorretos nas junções das superfícies e redundância na informação de vértices.

Além desses problemas, é necessário reorganizar a hierarquia de cena para otimizar e preparar o modelo final para a visualização em tempo real na aplicação de visualização.

Para automatizar os processos de correção dos vértices e preparação para a aplicação, foi desenvolvido um script para o 3D Studio Max. Este script deve ser executado após o processo de importação, selecionando a opção de menu "*MaxScript > Run Script > Selecione o arquivo inventor_post-production.ms*", como mostrado na figura abaixo.

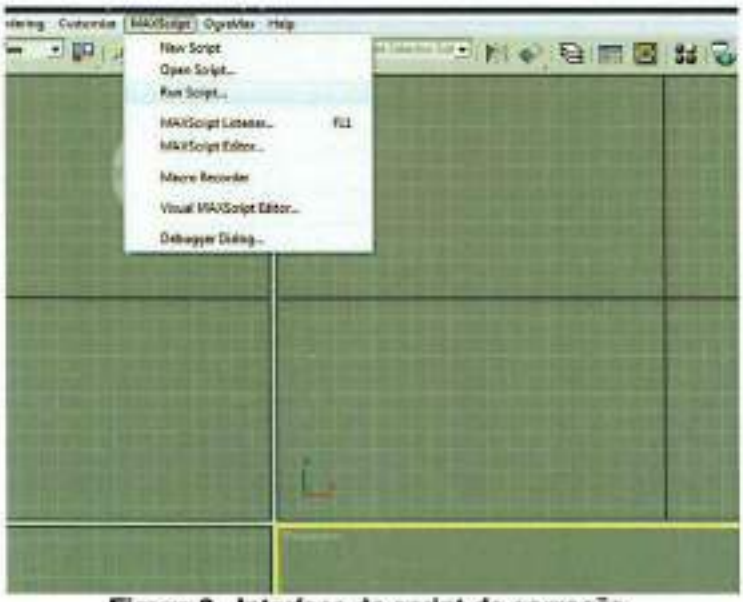

Figura 8 - Interface do script de correção.

A Figura 9 apresenta a interface simples do script de correção. Esta interface apresenta as seguintes seções: *Inventor Post-Production, Ogre Name Validation* e *Options*.

Figura 9 - Interface do script de correção.

Após abrir a janela de opções do Script, execute as seguintes atividades:

- *Inventor Post-Production*: Clique no botão "*Start Post-Production*" e aguarde alguns minutos. Esta seção corrige os problemas apontados anteriormente,

instanciando as peças duplicadas e reduzindo a quantidade de vértices. A opção "*Selected objects only*" permite com que o usuário aplique as correções apenas para os objetos selecionados da cena.

- *Ogre Name Validation*: Clique no botão "*Validate Names*". Esta seção remove os caracteres especiais e os acentos dos nomes dos objetos, para que os mesmos possam ser carregados corretamente pela aplicação de visualização.
- *Options*: A caixa "*Debug Mode*" permite que o Script imprima em uma janela tudo o que está sendo alterado na cena, para fins de diagnóstico e inspeção de seu funcionamento.

Após a execução do script, os objetos importados já foram aperfeiçoados e otimizados para visualização em tempo real e, portanto, já podem ser exportados para a ferramenta de visualização.

4.2.7 Conversão e Adaptação dos Modelos da Unidade Geradora

Para realizar o desenvolvimento deste projeto, os modelos da Unidade Geradora já tinham sido elaborados em CAD, utilizando a ferramenta Autodesk Inventor. Neste caso, o processo de modelagem foi simplificado, sendo necessário apenas converter estes modelos para malhas poligonais e implementar as suas respectivas animações.

O processo de conversão e adaptação destes modelos 3D consiste na transformação dos elementos descritos em CAD (superfícies paramétricas) para elementos descritos através de superfícies poligonais.

Esta conversão viabiliza a visualização destas peças em tempo-real e de forma interativa, permitindo a elaboração de aplicações em Realidade Virtual. Este processo de conversão foi realizado com o auxílio da ferramenta de modelagem Autodesk 3D Studio Max. Esta ferramenta possui recursos importar os modelos criados no Inventor.

Apesar do processo de importação funcionar corretamente para todas as peças da UG, uma inspeção mais detalhada do resultado revelou algumas imperfeições que poderiam ser melhoradas.

Um dos principais problemas encontrados durante o procedimento de importação foi na conversão de cenas contendo objetos em CAD repetidos, como parafusos, lâminas e grandes chapas.

Caso haja repetição de objetos no Inventor, o procedimento de conversão utilizado pelo 3D Studio Max cria, na cena virtual com polígonos, diversos modelos poligonais repetidos. Neste caso, a solução ideal seria criar um único objeto poligonal, e instanciá-lo repetidas vezes no cenário.

O procedimento de criação de instâncias é utilizado em inúmeras aplicações de visualização para otimizar o consumo de memória do computador e, algumas vezes, acelerar o procedimento de renderização. Neste caso, quando um determinado objeto é utilizado duas ou mais vezes no cenário virtual, a criação de instâncias pode ser utilizada para repetir este objeto sem replicá-lo na memória do computador.

No entanto, esta técnica não está sendo utilizada pelo *importer* do 3D Studio Max 2009 e, portanto, todos os objetos que aparecem repetidas vezes no Inventor são multiplicados na cena poligonal de forma indevida.

Além disto, outro problema foi identificado no resultado da conversão realizada pelo *importer* do 3D Studio Max: Os modelos descritos através de superfícies paramétricas, normalmente NURBS, ao serem convertidos para superfícies poligonais, tiveram alguns vértices desnecessários acrescentados à sua superfície.

Estes vértices desnecessários são acrescentados pelo conversor durante o processo de tecelagem, e estão sempre presentes na união entre duas superfícies adjacentes. Nestas ocasiões, estas duas superfícies permanecem separadas no modelo poligonal, podendo causar desconforto durante a sua visualização.

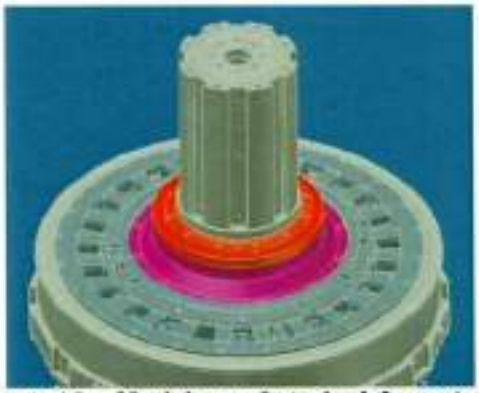

Figura 10 - Modelo no Autodesk Inventor

Para corrigir estes problemas, foi desenvolvido um *script* para ser utilizado juntamente com o software de modelagem 3D Studio Max. O *script* foi implementado utilizando a linguagem de programação do próprio *software*, o MAX Script.

A Figura 11 ilustra os modelos já convertidos dentro o 3DS Max.

Figura 11 - Modelo no 3D Studio Max.

No momento da execução do *script*, a cena importada para o 3DS Max é otimizada para reduzir o seu tamanho e aumentar o desempenho durante a visualização em tempo real.

Os resultados da execução do script podem ser observados na Tabela 3 abaixo:

Tabela 3 - Resultados da execução do Script

	Antes	Depois
Número de objetos únicos	5363	434
Número total de vértices	5.4M	3.1M

Uma vez convertidos e adaptados, os modelos geométricos do 3DS Max podem ser utilizados adequadamente pela aplicação de visualização. No A

4.2.8 Aplicar materiais e animações

Após os aperfeiçoamentos e otimizações dos modelos importados, é necessário ajustar e aplicar as propriedades de materiais (cores, texturas, propriedades de reflexão entre outros), e animar cada um dos objetos importados.

A aplicação de materiais e o uso das ferramentas de animação são recursos específicos da ferramenta de modelagem 3D Studio Max, e estão fora do escopo deste documento.

4.2.9 Exportar os modelos utilizando o plug-in OgreMax

Para exportar o modelo para a aplicação de visualização, é necessário utilizar os recursos disponíveis no plug-in OgreMax. Este plug-in permite que todo o conteúdo elaborado no 3D Studio Max (.max) seja exportado para o formato de arquivo aceito pelo visualizador (.scene).

O OgreMax apresenta um conjunto amplo de opções, sendo possível customizar diversos aspectos da cena e de cada objeto individualmente. Um manual completo sobre o funcionamento destas opções pode ser encontrado na seguinte página: http://www.ogremax.com/Documents/OgreMaxSceneExporter-3DSMax/.

Para exportar e cena com as opções pré-estabelecidas, basta acessar o item de menu "*OgreMax > Export > Export Scene*", ilustrado na Figura 12, e selecionar o local onde o arquivo será salvo.

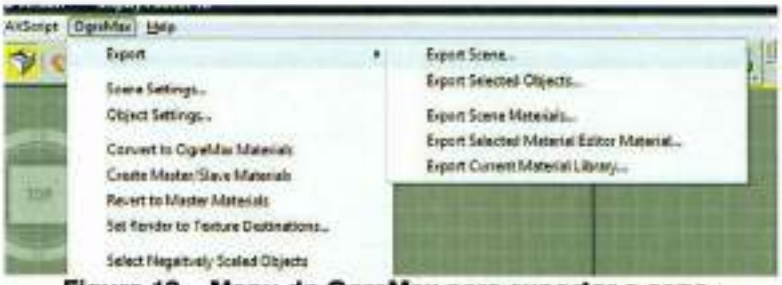

Figura 12 – Menu do OgreMax para exportar a cena.

Caso seja necessário exportar a animação de alguma peça, é preciso configurar de forma individual as animações de cada objeto. Para isso, basta selecionar o objeto desejado e chamar o item de menu "*OgreMax > Object Settings*".

Na janela de opções do objeto, clique na aba "Node Animation" e acrescente as animações desejadas (Figura 13).

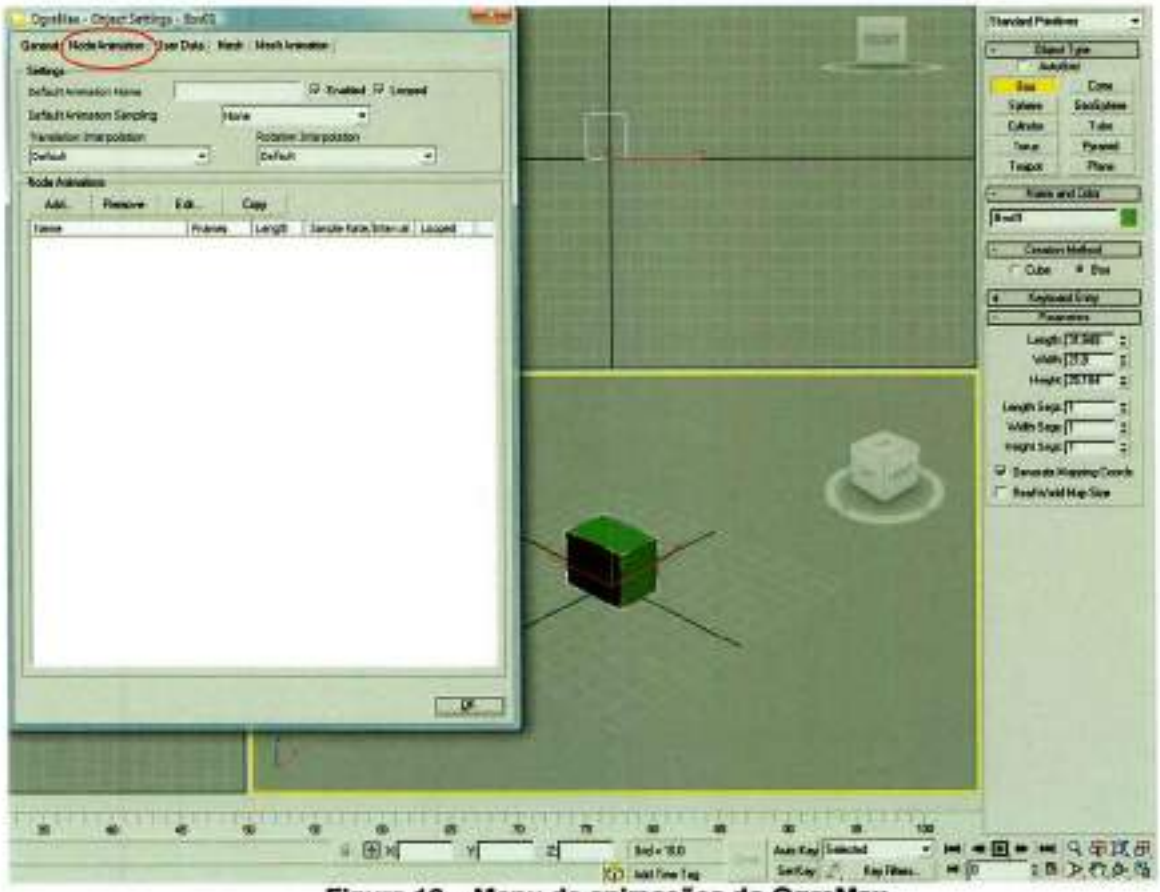

Figura 13 – Menu de animações do OgreMax.

Todas as animações acrescentadas nesta lista serão identificadas pelo visualizador e poderão ser utilizadas na criação de novos procedimentos.

4.3 Integração entre Módulo de Controle e Motor de Visualização

Nesta etapa do projeto os procedimentos de montagem/desmontagem da UG e as interfaces gráficas da aplicação foram integrados.

O Módulo de Controle foi desenvolvido integralmente pelo próprio laboratório e é composto de algoritmos de modelagem dos procedimentos de montagem e desmontagem da unidade geradora, controlando assim a interação do usuário com os objetos virtuais da UG. Este módulo é responsável por verificar se os procedimentos de montagem e desmontagem praticados pelo usuário estão de acordo com a ordem estabelecida pela equipe de manutenção.

O Módulo de Controle utiliza arquivos de configuração no formato XML. Estes arquivos contêm a descrição de todos os componentes da UG e dos recursos disponíveis para interagir e manipular cada um destes elementos.

A descrição dos procedimentos de montagem e desmontagem determina como o usuário deve interagir com os componentes presentes no cenário virtual para atingir os objetivos da atividade.

Este módulo utiliza diagramas de redes de Petri para representar as relações de dependência existentes entre as etapas dos procedimentos. Esta descrição contém, para cada procedimento de montagem/desmontagem, a ordem com que o usuário deve manipular cada peça ou objeto da UG.

Estes diagramas de Petri são elaborados a partir da ferramenta NetLab e exportados para XML em um formato padronizado para este tipo de informação. Este padrão é chamado de PNML – Petri Net Markup Language.

O Módulo de Controle carrega a informação da rede de Petri por meio destes arquivos XML e os transforma em algoritmos procedurais que regem a lógica de montagem e desmontagem da unidade geradora.

Como o software de visualização é desenvolvido em C++, foi necessário um esforço de integração entre as funções do motor de visualização e a lógica de execução dos procedimentos.

4.3.1 Ferramentas utilizadas

Neste tópico, são mencionadas as principais ferramentas utilizadas durante o desenvolvimento do projeto. De forma geral, as ferramentas e tecnologias mencionadas foram escolhidas para facilitar a manutenção e o aprimoramento do software em trabalhos futuros.

As principais ferramentas utilizadas para o desenvolvimento da aplicação de treinamento foram:

Ogre 3D

Biblioteca de renderização distribuída gratuitamente. Esta biblioteca é o núcleo do sistema de visualização. Uma das vantagens do Ogre 3D é a compatibilidade com a

ferramenta OgreMax, que foi essencial para exportar objetos poligonais da aplicação de modelagem 3D Studio Max.

Hikari

O Hikari permite que programas elaborados com a tecnologia Adobe Flash, sejam apresentados ao usuário de forma compatível com Ogre. Este recurso foi utilizado principalmente na composição da interface gráfica com o usuário.

NetLab

Ferramenta gratuita que permite a criação de redes de Petri de forma rápida e fácil através de diversos recursos disponíveis em sua interface gráfica. Além disto, esta aplicação possui suporte ao formato PNML (Petri-Net Markup Language), que é um formato padrão ISO e que facilita a integração das redes geradas nesta aplicação com o Módulo de Controle da ferramenta de visualização.

Adobe Flash

Ferramenta utilizada para criar as interfaces gráficas da aplicação, como botões, menus e caixas de diálogo. É uma ferramenta amplamente utilizada na elaboração de interfaces, pois permite o uso da linguagem de programação ActionScript, que é uma linguagem simples, flexível e de desenvolvimento rápido.

5 Resultados

Os resultados obtidos dentro do contexto desta monografia de Projeto de Formatura estão pontuados abaixo:

- Otimização do script de conversão e ajuste dos polígonos entre o INVENTOR e o 3D MAX
- Desenvolvimento de um script para ajuste dos modelos do 3D MAX para se adequarem a nomenclatura das entidades do XML, bem como, as estruturas da Rede de Petri;
- Desenvolvimento de script para correção da nomenclatura dos modelos, que até então causavam erros no visualizador;
- Definição e montagem dos arquivos de entidades XML.

A otimização dos scripts de conversão dos arquivos do Inventor para o 3D Max, bem como a otimização destes arquivos permitiram reduzir a complexidade da aplicação permitindo se adequar aos requisitos de execução nos sistemas operacionais Windows 7 ou Windows Vista da Microsoft – com suporte tanto para a versão 32 bits, como para a versão 64 bits.

A partir destes pontos podemos dizer que, implementação das entidades por arquivos XML no módulo de visualização permitiu o controle dos objetos de cena que, embora o cenário virtual tenha um número consideravelmente grande de polígonos, o software de visualização realiza otimizações que permitem com que os objetos que não estejam sendo observados pelo usuário sejam removidos da cena virtual, aumentado o desempenho da aplicação.

REFERÊNCIAS

[1] G. Chen, Z. Gan, J. Sheng, X. Lu, "Equipment Simulation Training System Based on Virtual Reality", International Conference on Computer and Electrical Engineering, Dec., 2008

[2] B. Arendarski, W. Termath, P. Mecking, "Maintenance of Complex Machines in Electric Power Systems Using Virtual Reality Techniques", Conference Record of the IEEE International Symposium on Electrical Insulation, 2008

[3] D. M. Oliveira, S. C. Cao, X. F. Hermida, F. M. Rodriguez, "Virtual Reality System for Industrial Training", International Symposium on Industrial Electronics, 2007

[4] E. Blümel, G. Müller, W. Salem, M. Schenk, "Technology enhanced training at workplace: A virtual reality based training system for the technical domain", International Conference on E-Business and E-Learning, 2005

[5] D. F. McAllister, "Stereo Computer Graphics and other True 3D technologies", Princeton U. Press, Princeton, NJ, Oct 1993

[6] D. F. McAllister, "Stereo Computer Graphics", in State of the Art in Computer Graphics – Aspects of Visualization, Eds. Springer-Verlag, Dec. 1993, p. 143-175

[7] D. F. McAllister, "3D Displays", Wiley Encyclopedia on Imaging, Jan., 2002, pp. 1327-1344

[8] M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language", Addison-Wesley Professional, 3rd Edition, Sept. 2003.

[9] E. Gamma, R. Helm, R. Johnsonm, J. M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, Nov. 1994.

[10] G. Junker, "Pro OGRE 3D Programming", Apress, 1st Edition, Sept. 2006.

[11] A. J. Simons, "Hikari: A Flash embedding layer for Ogre3D", [Online]. Disponível: http://code.google.com/p/hikari-library/.

[12] J. M. Zelle and C. Figura, "Simple, low-cost stereographics: VR for everyone", in Proceeding of the 35th SIGCSE technical symposium on Computer Science Education, 2004, p. 348-352

[13] B. Raffin, M. K. Zuffo, H. Kerzmarski, Z. Pan, "Commodity Cluster for Virtual Reality", in Proceedings of the IEEE Virtual Reality, 2003, p. 307

[14] L. P. Soares, B. Raffin, J. A. Jorge, "PC Clusters for Virtual Reality", in Proceedings of the IEEE conference on Virtual Reality, 2006, p. 215-222

APÊNDICE A – ELEMENTOS DE CONFIGURAÇÃO DO XML

Este anexo contém uma listagem dos principais elementos para o arquivo de configuração XML da aplicação.

Propriedades

As propriedades guardam informações a respeito das características e do estado interno das entidades. Elas são definidas no arquivo XML entre as *tags* <properties>.

As propriedades utilizadas pela aplicação de visualização são:

description: Descrição da entidade. Nome usado na composição dos elementos da interface gráfica. Este é o nome que será usado para apresentar a peça ao usuário.

icon: Propriedade que define o caminho do arquivo que contém o ícone da peça. Este ícone também é usado na interface gráfica da aplicação.

Ex: <property name="icon" value="resources/icons/ProtecaodoAcoplamento.png"/>

selectable: Propriedade que define se a peça poderá ser selecionada diretamente pelo usuário. Esta propriedade contém um valor booleano (*true*/*false*), ou seja, definindo a propriedade como '*true*' (verdadeiro), ao passar o mouse sobre a peça ela ficará destacada em verde. Caso contrário, se a propriedade for definida como '*false*' (falso), passando o mouse sobre a peça, ela não será destacada das demais.

attached: Propriedade com valor booleano (*true*/*false*) que define a visibilidade da peça. Peças com a propriedade igual a '*true*' estarão visíveis. Caso contrário, se o valor da propriedade for '*false*', a peça será removida da cena, permanecendo invisível. É importante notar que a visibilidade de um determinado objeto depende da visibilidade das peças acima deste. Ou seja, um determinado objeto só permanecerá visível se a sua propriedade for '*true*' e se a entidade que é pai desta também estiver visível e assim sucessivamente.

state: Propriedade que define o estado da peça. O seu valor determina, por exemplo, se a peça está montada ou não. Normalmente, esta propriedade é usada para controlar o conjunto de ações que estão disponíveis dentro de um determinado objeto.

information: Propriedade que define o caminho do arquivo com informações técnicas sobre a peça.

Comandos

As ações definidas em uma determinada Entidade são descritas através de uma sequência de comandos XML. Estes comandos definem o que o software de visualização precisa fazer quando o usuário executar uma determinada ação.

Os comandos são encontrados no arquivo XML entre as *tags* <availability> e <execution>, na definição de uma ação, realizada na *tag* <action>.

Todos os comandos retornam um determinado valor, que varia de acordo com implementação deste comando. Este retorno é usado principalmente na definição da *tag* <availability>. Quando o valor retornado pelo comando for correspondente a um valor booleano '*true*' (verdadeiro), a ação estará disponível. Caso contrário, a ação não estará disponível para o usuário.

check_conditions: Este comando é utilizado para verificar uma condição, ou um conjunto de condições.

Para verificar estas condições, todos os comandos dentro da tag <conditions> serão executados sequencialmente. Caso o retorno destes comandos seja igual ao valor da propriedade *check_return* desta mesma *tag*, a condição será verdadeira. Caso contrário, a condição será falsa.

Caso a condição seja verdadeira, todos os comandos presentes dentro da *tag* <true> serão executados, caso contrário, os comandos da *tag* <false> serão executados.

O próprio comando *check_conditions* sempre retornará um valor booleano, sendo este valor '*true*' quando as suas condições forem verdadeiras, e '*false*' quando correr o contrário.

A seguir, um exemplo simplificado de uso do comando:

```
<check_conditions>
        <conditions check_return="true" logic="or">
                <check_property property="state" value="remove"/>
        </conditions>
    <true>
```

```
    :
  </true>
  <false>
    :
  </false>
</check_conditions>
```

check_property: Este comando verifica se uma determinada propriedade contém um valor específico. Quando o valor da propriedade é igual ao valor verificado, este comando retorna '*true*', caso contrário retorna '*false*'. Em geral, este comando é utilizado em conjunto com o **check_conditions**.

Ex: <check_property property="state" value="remove"/>

Neste exemplo, verificamos se o valor da propriedade 'state' é igual a 'remove'

flash_call_command: Este comando é utilizado para que o XML possa interagir com os elementos da interface gráfica, elaborados em Flash. Este comando permite que as ações definidas no XML possam apresentar caixas de diálogo ou outros menus.

No caso deste projeto, a única entidade Flash que pode ser usada no arquivo XML é a entidade de nome *main_menu*, definida pelo atributo *flash_entity*. O atributo *function* determina qual interface gráfica o comando irá exibir.

Atualmente, o atributo *function* suporta apenas dois valores: *show_selection_menu* e *show_messege_box*. A opção *show_selection_menu* apresenta ao usuário um menu contendo diversas alternativas de peças. Já a opção *show_message_box* apresenta ao usuário uma caixa de diálogo contendo uma determinada mensagem.

```
<flash_call_command flash_entity="main_menu" function="show_selection_menu">
        <parameter name="action" value="insert" />
        <parameter name="entities" value="alavanca_da_palheta, ..." />
</flash_call_command>
```

Este primeiro exemplo demonstra o uso do comando para apresentar um menu com alternativas de peças. Este é o comando executado ao clicar em 'Aplicar peças' durante a execução do aplicativo, que por sua vez, abre o menu de seleção de peças, com a lista dos elementos que podem ser aplicados.

A função *show_selection_menu* necessita de apenas dois parâmetros: *action* e *entities*. O parâmetro *entities* contém a lista das peças a serem apresentadas no

menu. Já o parâmetro *action* contém a ação que será executada na peça escolhida pelo usuário.

```
<flash_call_command flash_entity="main_menu" function="show_message_box">
        <parameter name="title" value="Operação Inválida" />
        <parameter name="message" value="Não é possível executar este
procedimento neste momento."/>
</flash_call_command>
```

Neste segundo exemplo, o mesmo comando é utilizado para apresentar uma caixa de diálogo, utilizando a função *show_message_box*. Esta função também necessita de apenas dois parâmetros: *title* e *message*.

Neste caso, o parâmetro *title* determina o título da caixa de diálogo que será apresentada ao usuário, e o parâmetro *message* determina a mensagem que será apresentada.

ogre_animate: Este comando executa as animações associadas a uma determinada entidade. No caso da aplicação de visualização, as animações executadas são de inserir ou remover as peças.

Ex: <ogre_animate animation="remove" />

Ao executar o comando do exemplo, a animação de nome *remove* será executada na entidade onde comando está sendo utilizado.

petrinet_check_transition: Este comando verifica se a transição solicitada está disponível para ser executada na rede de Petri. Este comando é utilizado na definição das ações, para verificar se estas ações serão executadas de acordo com a definição do procedimento. Esta verificação é realizada no momento em que o usuário solicitar a sua execução.

Ex: <petrinet_check_transition net="exemplo_rede" transition="remove@excitatriz" />

Neste caso, o comando verificará na rede de Petri de nome 'exemplo_rede', se a transição de nome 'remove@excitatriz' está disponível para execução. Caso a transição esteja disponível, o comando retornará '*true*', caso contrário retornará '*false*'.

petrinet_execute_transition: Este comando executa uma determinada transição na rede de Petri. Esta execução é realizada apenas se a transição estiver disponível.

Ex <petrinet_execute_transition net="exemplo_rede" transition="remove@excitatriz" />

Neste exemplo, o comando executará na rede de Petri de nome 'exemplo_rede', a transição de nome 'remove@excitatriz'. Este comando retornará '*true*' caso a transição seja efetuada sem erros. Caso a transição não esteja disponível, este comando retornará '*false*', indicando que não foi possível efetuá-la.

set_property: Este comando altera o valor de uma determinada propriedade.

Ex: <set_property property="attached" value="false"/>

Neste exemplo, o valor da propriedade '*attached*' é alterado para '*false*'.

APÊNDICE B – LISTAGEM DO SCRIPT DE CONVERSÃO E OTIMIZAÇÃO DE MODELOS.

A seguir listamos o código desenvolvido em conjunto pela equipe da Caverna Digital para conversão e otimização dos modelos do Inventor para o 3D MAX

```
-- NÃO MODIFIQUE ESTE CÓDIGO.

(

--=====================================::PRE
DECLARATIONS::======================

--========================

-- pre declare local functions

local get_original_name, fix_geometry, fix_and_instance, start_post_production,
validate_names, range_replace, remove_accents

local getObjectsByMaterial, setObjectsMaterial, deleteMaterials, convertMaterials,
organize_nodes

-- pre declare local rollouts

local PostProductionHelp, PostProductionOptions, PostProduction, NameValidation,
OrganizeNodes

-- pre declare local variables

local debug, instanceFlag, obj_set, ban_list, ban_mat

local mtScene, mtCount, mtDel, debug

--local children_copy, search_obj, node_organize, num_group

------------------------------------------------------------------------------------------------------------------------
----

--
=====================================================================
===================================

-- ************************************:: FUNCTIONS
::**************************************************
```

```
--
================================================================================
==================================
--------------------------------------------------------------------------------
--                              FUNCTION:   GET ORIGINAL NAME
--                          Retorna o nome original da peça (prefixo)
--------------------------------------------------------------------------------
--      ex:                 get_original_name  parafuso_sextavado:1
--      retorna:            parafuso_sextavado
--
--      ex:                         get_original_name  D23:76-81:645_16:12
--      retorna:            D23:76-81:645_16
-- ------------------------------------------------------------------------------
        fn get_original_name name =
        (
                local name_segments = filterString name ":"

                if name_segments.count > 1 then
                (
                        local suffix_number = name_segments[ name_segments.count
] as Integer

                        if suffix_number != undefined then
                        (
                                local object_name = name_segments[1]

                                for i = 2 to name_segments.count - 1 do
                                        object_name += ( ":" + name_segments[ i ] )
```

```
                return object_name

            )
            else
                return name

        )
        else
            return name
    )

fn fix_geometry obj =
(
    --======  Weld  and  Normal  Fixing  ==========================================
    -- Corrige o objeto com faces separadas, unificando
    -- dois ou mais vértices que estejam próximos. Para peças muito
    -- pequenas, como parafusos pode ser necessário diminuir o Threshold.
    -- 0.03 é um bom valor para a maioria das peças.
    -- ------------------------------------------------------------------------
    convertToPoly( obj )
    select obj
    obj.weldThreshold = 0.03
    polyop.weldVertsByThreshold obj #all
    modPanel.addModToSelection (Edit_Normals ()) ui:on
    subobjectlevel = 1
    nn = obj.Edit_Normals.GetNumNormals()
```

```
        obj.Edit_Normals.Break selection:#{1 .. nn}

        if PostProductionOptions.debug_box.checked do
                format "    Fixed with weld and correct normals.\n" to:debug

        --=======:                Smoothing                Groups                fixing
:=======================================
        -- Aqui é feito a correção de smoothing group das peças.
        -- ------------------------------------------------------------------------------
        convertToPoly( obj )
        modPanel.addModToSelection (Edit_Poly ()) ui:on
        subobjectLevel = 4
        sel = polyop.getFaceSelection obj
        max select all
        --obj.autoSmoothThreshold = 20
        --obj.EditablePoly.autosmooth ()
        obj.modifiers[#Edit_Poly].autoSmoothThreshold = 20
        obj.modifiers[#Edit_Poly].ButtonOp #Autosmooth
        --polyop.setFaceSelection obj sel
        subobjectLevel = 0

        clearSelection()

        if PostProductionOptions.debug_box.checked do
                format "    Smoothing Groups corrected.\n" to:debug
        --
=============================================================================
==
```

```
)

fn fix_and_instance =
(
		if PostProduction.selection_box.checked then
			obj_set = selection as array
		else
			obj_set = objects as array
		-- Faz uma lista com os objetos e armazena em um array. Necessário pois
		-- o passo de Weld e Normal Break exige seleção e deseleção de peças.

		instanceFlag = #()
		-- Instance Flag serve para marcar todas as peças que já foram instanciadas. Deste modo
		-- quando o script identifica uma peça instanciada, ele ignora e pula para a próxima

		for i=1 to obj_set.count do
		-- ------------------------------------------------------------------------------------
		-- LOOP 1: percorre todos os objetos do obj_set.
		-- i = "Object Index"
		-- ------------------------------------------------------------------------------------
		(
			checkFlag = findItem instanceFlag i
			-- Checa se o objeto consta na lista de instâncias ou não, através de procura pelo Object Index (i)
			-- Se encontrar o objeto na lista, pula o código e passa para o objeto seguinte, reiniciando o LOOP 1
```

```
-- Se não encontrar, retorna zero - significa que é um objeto original, e roda o script.
if checkFlag == 0 then
(

    if PostProductionOptions.debug_box.checked do
        format "Found original Object. [index : %] \n"  i to:debug

    obj = obj_set[i]

    fix_geometry(obj)

    search_name = get_original_name obj.name
    search_index = 1
    replace_counter = 1

    if PostProductionOptions.debug_box.checked do
        format "    Searching objects with name % (from %)\n" search_name obj.name to:debug

    for replace_obj in obj_set do
    -- ------------------------------------------------------------------------------------------
    -- LOOP 2: percorre todos os objetos do obj_set, procurando por réplicas do atual objeto (index i).
    -- Caso encontre, transforma a réplica em instancia e padroniza seu nome. Depois inclui o objeto instanciado
    -- na lista de instâncias (instanceFlag)
```

```
                    -- "search_index" é equivalente ao "Object Index" do LOOP 1
                    -- ----------------------------------------------------------------------------
----------------------------------

                    (
                        current_name = get_original_name replace_obj.name
                        if search_name == current_name and search_index != i
do
                        (
                            if areNodesInstances replace_obj obj != true do
                            (
                                instancereplace replace_obj obj
                                if
PostProductionOptions.debug_box.checked do
                                    format "      Created %(%) as
instance of %(%)\n" replace_obj.name search_index obj.name i to:debug

                                --replace material of instanced objects
                                replace_obj.material = obj.material
                            )

                            replace_obj.name =   current_name + ":" +
replace_counter as String --> sufixos para objetos instanciados.
                            replace_counter += 1

                            append instanceFlag search_index --> inclui na
lista de instâncias
                        )
                        search_index += 1
                    )
```

```
                        if replace_counter != 1 do
                            obj.name = search_name --> objeto original !
                    )
                    else --> checkFlag != 0
                    (
                        if PostProductionOptions.debug_box.checked do
                            format "Found instanced Object. [index : %] - move to
next \n" i to:debug
                    )

                    PostProduction.progress_bar.value = (i  * 100) / obj_set.count

                    --i += 1
                )
)

-- ------------------------------------------------------------------------
--                          FUNCTION: POST PRODUCTION
-- ------------------------------------------------------------------------

fn start_post_production =
(
        start = timeStamp()
        PostProduction.generate_btn.enabled = false

        if PostProductionOptions.debug_box.checked do
        (
```

```
		debug = newScript()
		format "Started auto instance replacement\n----------\n"
to:debug
	)

	fix_and_instance()

	end = timeStamp()

	if PostProductionOptions.debug_box.checked do
	(
		totalObjects = obj_set.count - instanceFlag.count
		format "\n----------\n" to:debug
		format "- ORIGINAL SCENE: \n" to:debug
		format "Objects: % \nInstances: 0 \n" obj_set.count
to:debug
		format "----------\n" to:debug
		format "- AFTER SCRIPT: \n" to:debug
		format "Objects: % \nInstances: % \n" totalObjects
instanceFlag.count to:debug
		format "----------\n" to:debug
		format "Finished script in % seconds\n" ((end - start) / 1000.0) to:debug
	)

	PostProduction.progress_bar.value = 0
	PostProduction.generate_btn.enabled = true

	messageBox "Post Production Done!"
```

```
)

-- ------------------------------------------------------------------------

--                                FUNCTION: VALIDATE NAMES

-- ------------------------------------------------------------------------

--  This script replaces special charaters like "ç" and accentuation from all models of your scene

--  Then searches for repeated node names, changing if necessary


-- this is the banned characters list, based on ANSI chart.

-- the first parameter is the replacement letter, the other two parameters are the range of banned characters.

-- eg: "A-À-Å" means: ---> replace all characters from  À  to  Å  with  A.


local ban_list = #("A-À-Å", "E-È-Ê", "I-Ì-Ï", "O-Ò-Õ", "U-Ù-Ü", "a-à-å", "i-ì-ï", "e-è-ê", "o-ò-õ", "u-ù-ü", "C-Ç-Ç", "c-ç-ç", "N-Ñ-Ñ", "n-ñ-ñ","_-€-þ", "_-:-?", "_-/-/", "_-\\-\\", "_-\"-\"", "_-|-|")

local ban_mat = #("_sp_- - ", "_-:-:")


fn range_replace objeto rep rmin rmax =

(

        local int range_min = bit.charAsInt rmin

        local int range_max = bit.charAsInt rmax

        for y = 1 to objeto.count do

        (

                local int n = bit.charAsInt objeto[y]

                if n >= range_min and n <= range_max then

                        objeto[y] = rep
```

```
        )
        return objeto
)

fn remove_accents obj list =
(
        local string mn = obj.name
        for j = 1 to list.count do
        (
                local letter = filterString list[j] "-"
                mn = range_replace mn letter[1] letter[2] letter[3]
        )
        return mn
)

fn validate_names =
(
        objs = objects as array

        -- first loop: remove special characters from Objects
        for u=1 to objs.count do
        (
                objs[u].name = remove_accents objs[u] ban_list
                --u += 1
        )

        -- remove special characters from Materials
        for l=1 to sceneMaterials.count do
```

```
(
	sceneMaterials[l].name = get_original_name sceneMaterials[l].name
	sceneMaterials[l].name = remove_accents sceneMaterials[l] ban_list
	sceneMaterials[l].name = remove_accents sceneMaterials[l] ban_mat
)

-- second loop: search for repeated names in the Objects
for k=1 to objs.count do
(
	check_name = objs[k].name
	check_index = 1
	rename_counter = 1
	for renameObj in objs do
	(
		if check_name == renameObj.name and check_index != k  do
		(
			renameObj.name =  check_name + "$" + rename_counter as
String
			rename_counter += 1
		)
		check_index += 1
	)
	--k += 1
)

-- search for repeated names in the Materials
for p=1 to sceneMaterials.count do
(
```

```
            check_name = sceneMaterials[p].name
            check_index = 1
            rename_counter = 1

            for renameMat in sceneMaterials do
            (
                if check_name == renameMat.name and check_index != p do
                (
                    renameMat.name = check_name + "$" + rename_counter as
String
                    rename_counter +=1
                )
                check_index += 1
            )
        )

        messageBox "Name Validation Done!"

)

-- ------------------------------------------------------------------
--          FUNCTIONS FOR MATERIAL CONVERSION
-- ------------------------------------------------------------------

        fn getObjectsByMaterial mat =
        (
            return (for g in objects where g.material == mat collect g)
        )
```

```
fn setObjectsMaterial objArray mat =
(
	objArray.material = mat
)

fn convertMatList list =
(
	if PostProductionOptions.debug_box.checked do format "*** Class of Process
: [ % ] *** \n" (classof list)  to:debug
	listType = classof list

	for i = 1 to list.count do
	(
		cls = classof list[i]
		if  PostProductionOptions.debug_box.checked  do  format  "===
MATERIAL: % === CLASS = % ==============\n" list[i].name cls to:debug
		if (cls == Architectural) then
		(
			oldMat = list[i]
			if  PostProductionOptions.debug_box.checked  do  format  "***
Architectural material found : % \n" oldMat.name to: debug
			r = Standard()
			if  PostProductionOptions.debug_box.checked  do  format  "--
Copying atributes to new Standard material \n" to: debug
			r.name = oldMat.name
			r.shaderType = 1
			r.opacity = (100 - oldMat.transparency)
--			r.ambient =
```

```
--		r.ambientMap
--		r.ambientMapAmount =
--		r.ambientMapEnable =
		r.diffuse = oldMat.diffuse
		r.diffuseMap = oldMat.diffuseMap
		r.diffuseMapAmount = oldMat.diffuseMapAmount
		r.diffuseMapEnable = oldMat.diffuseMapEnable
--		r.diffuseLevel =
--		r.diffuseLevelMap =
--		r.diffuseLevelMapAmount
--		r.diffuseLevelMapEnable =
--		r.specularMap =
--		r.specularMapAmount =
--		r.specularMapEnable =
--		r.glossiness =
--		r.glossinessMap =
--		r.glossinessMapAmount =
--		r.glossinessMapEnable =
--		r.specular =
--		r.specularLevel =
--		r.specularLevelMap =
--		r.specularLevelMapAmount =
--		r.specularLevelMapEnable =
--		r.selfIllumAmount =
--		r.selfIllumColor =
--		r.selfIllumMap =
--		r.selfIllumMapAmount =
--		r.selfIllumMapEnable =
```

```
--                      r.opacity =
--                      r.opacityFallOff =
--                      r.opacityFallOffType =
--                      r.opacityMap =
--                      r.opacityMapAmount =
--                      r.opacityMapEnable =
--                      r.opacityType =
--                      r.filter =
--                      r.filterColor =
                        r.filterMap = oldMat.filterMap
--                      r.filterMapAmount =
--                      r.filterMapEnable =
                        r.bumpMap = oldMat.bumpMap
                        r.bumpMapAmount = oldMat.bumpMapAmount
                        r.bumpMapEnable = oldMat.bumpMapEnable
--                      r.reflectionLevel =
--                      r.reflectionMap =
--                      r.reflectionMapAmount =
--                      r.reflectionMapEnable =
                        r.displacementMap = oldMat.displacementMap
                        r.displacementMapAmount = oldMat.displacementMapAmount
                        r.displacementMapEnable = oldMat.displacementMapEnable

                        if (listType == MaterialLibrary) then
                        (
                                if PostProductionOptions.debug_box.checked do format
"--- Aplying new material to objects in scene \n" to:debug
```

```
                    setObjectsMaterial (getObjectsByMaterial oldMat) r

                    if PostProductionOptions.debug_box.checked do format
"---- Replacing [ % ] with new Standard material \n" oldMat.name to: debug

                    append sceneMaterials r
                )
                else if (listType == Multimaterial) then
                (
                    list[i] = r
                )
            )
            else if (cls == Multimaterial) then
            (
                if PostProductionOptions.debug_box.checked do
                (
                    format "==== Multi-Material found ==== \n"  to:debug
                    format "Processing Multi-material [ % ] \n" list[i].name
to:debug

                )
                convertMatList list[i]
            )
            if PostProductionOptions.debug_box.checked do
            (
                format "----- index : % \n" i to:debug
                format "-- processed material [ % ] is [ % ] type \n" list[i].name
(classof list[i]) to: debug
```

```
                format "---------- \n" to:debug
            )
        )
    )

-- ----------------------------------------------------------------------
--                  FUNCTION: CONVERT MATERIALS
-- ----------------------------------------------------------------------
-- Convert Architectural materials to Standard materials type

    fn convertMaterials =
    (
        if PostProductionOptions.debug_box.checked do
        (
            start = timeStamp()
            debug = newScript()
        )

        mtScene = sceneMaterials

        if PostProductionOptions.debug_box.checked do format "Scene Materials
Count: % \n" mtCount to:debug

        convertMatList sceneMaterials

        if PostProductionOptions.debug_box.checked do
        (
            format "---------- \n" to:debug
```

```
                format "number of materials in scene :: % \n" sceneMaterials.count
to:debug

                end = timeStamp()
                format "Finished script in % seconds\n" ((end - start) / 1000.0) to:debug
            )
            messageBox "Material Conversion Done!"
        )

-- ------------------------------------------------------------------------
--                  FUNCTION: ORGANIZE NODES
-- ------------------------------------------------------------------------
-- Look in hyerarchical node tree and format node names to create internal groups by node
name
------------------------------------------------------------------------
-- Author :  Marcos Nobre
-- Version:  3ds max 2009
--
*****************************************************************************************************
***************
-- Rev: 1.0
-- last update:

-- To Do's: Avaliar novo método para registrar a variável num_group, uma vez que variáveis
globais podem trazer problemas futuros.
--
*****************************************************************************************************
***************
        /*fn xml_entity_writer group_name_final obj_name =
```

```
(

	--xml_file = getOpenFileName caption:"Open Graphical Entity XML File:" \
	--filename:"C:/Users/marcos/Documents/3dsmax/Conversion Script/" \
	--types: "Xml Files(*.xml)|*.xml|All Files (*.*)|*.*|"



	--createFile "Scene.xml"
	edit_xml_file = openfile(xml_file) mode:"a+"
	entity_name = substitutestring group_name_final "Group:" ""
	--format "<?xml version=\"1.0\" encoding=\"iso-8859-1\"?>\n" to:xml_file
	format "\n<graphical_entity name=\"%\">\n" entity_name to:xml_file
	format "	<properties>\n" to:xml_file
format "	<property name=\"description\" value=\"%\" />\n" obj_name  to:xml_file
format "	<property name=\"icon\" value=\"\" />\n" to:xml_file
format "	<property name=\"state\" value=\"\" />\n" to:xml_file
format "	<property name=\"selectable\" value=\"true\" />\n" to:xml_file
format "	<property name=\"visible\" value=\"true\" />\n" to:xml_file
	format "	</properties>\n" to:xml_file
	format "	<actions>\n" to:xml_file
format "	</actions>\n" to:xml_file
	format "</graphical_entity>\n" to:xml_file
)*/
--global path_xml_file
fn generate_entities node_selected xml_file=
(
	local x=1
	local num_entities = 0
```

```
local children_copy = #()
-- Monta array com os nós filhos
while x<= node_selected.children.count do
(
	join children_copy #(node_selected.children[x])
	x=x+1
)
while children_copy.count !=0 and children_copy != undefined do
(
	--Pega o primeiro item do array, apaga ele e compara com os demais itens
	local search_obj =  children_copy[1]
	deleteItem  children_copy 1

	--Se o nome do nó for entity ou group, ele entra no grupo para organizar os nós dentro dele, caso contrário continua a organização dos nós no nível em que está
	--if findstring search_obj.name "entity:"  == 1 or (isgrouphead search_obj == true and findstring search_obj.name "Group" == 1) then
	if isgrouphead search_obj == true then
	(

		--createFile "Scene1.xml"
		--if findstring search_obj.name "entity:"  == 1 do entity_name = substitutestring search_obj.name "entity:" ""
		if findstring search_obj.name "Group:"  == 1 do
		(
```

```
                    local entity_name = substitutestring search_obj.name
"Group:" ""
                    local strlen = entity_name.count
                    local strpoint = findstring entity_name "@"
                    local description = substring entity_name 1 (strpoint-1)
                    --format "<?xml version=\"1.0\" encoding=\"iso-8859-
1\"?>\n" to:xml_file
                    format "\n<graphical_entity name=\"%\">\n" entity_name
to: xml_file
                    format "     <properties>\n" to: xml_file
                    format "          <property name=\"description\"
value=\"%\" />\n" description to: xml_file
                    format "          <property name=\"icon\" value=\"\"
/>\n" to: xml_file
                    format "          <property name=\"state\"
value=\"\" />\n" to: xml_file
                    format "          <property name=\"selectable\"
value=\"true\" />\n" to: xml_file
                    format "          <property name=\"visible\"
value=\"true\" />\n" to: xml_file
                    format "     </properties>\n" to: xml_file
                    format "     <actions>\n" to: xml_file
                    format "     </actions>\n" to: xml_file
                    format "</graphical_entity>\n" to: xml_file
                    num_entities = num_entities + 1
                )
                local n = generate_entities search_obj xml_file
                num_entities = num_entities + n
            )
        )
```

```
        return num_entities
    )
    --Define a variável global num_group para registrar o número de grupos criados pelo organize_nodes
    --global num_group = 0
    fn organize_nodes node_organize =
    (

        --Verifica se o nó selecionado é um group_head e se ele está aberto. É necessário que o group esteja aberto para que seja manipulado
        if node_organize != rootnode do
        (
            if isOpenGroupHead (node_organize) == false and isGroupHead (node_organize) ==true do
            (
                setGroupOpen (node_organize) true
            )
        )
        local num_group = 0
        local x=1
        local children_copy = #()
        -- Monta array com os nós filhos
        while x<= node_organize.children.count do
        (
            join children_copy #(node_organize.children[x])
            x=x+1
        )
```

```
while children_copy.count !=0 and children_copy != undefined do
(
	--Pega o primeiro ítem do array, apaga ele e compara com os demais
ítens
	local search_obj =  children_copy[1]
	deleteItem  children_copy 1

	--Se o nome do nó for entity ou group, ele entra no grupo para
organizar os nós dentro dele, caso contrário continua a organização dos nós no nível em
que está
	--if findstring search_obj.name "entity:"  == 1 or (isgrouphead
search_obj == true and findstring search_obj.name "Group:" != 1) then
	if isgrouphead search_obj == true then
	(
		local n = organize_nodes search_obj
		num_group = num_group + n
	)
	else
	(
		if findstring search_obj.name "Group:" != 1 do
		(
			local group_node_arr = #()
			local i=1
			local  comp_obj  =  get_original_name  search_obj.name

			join group_node_arr #(search_obj)
			while i <=children_copy.count do
			(
```

```
                local obj = get_original_name children_copy[i].name

                if stricmp obj comp_obj ==0 then
                (
                    join group_node_arr #(children_copy[i])
                    deleteItem children_copy i
                )
                else
                (
                    i =i+1
                )
            )
            --Só cria um novo grupo se o array tiver mais do que um nó.
            if group_node_arr.count > 1 do
            (
                makeUniqueArray group_node_arr -- Aparentemente desnecessário, mas para evitar a criação de algum grupo com nós duplicados

                --Define o Group_head_name, ou seja, o nome do grupo após o "@"
                if search_obj.parent !=undefined then
                (
                    local group_head_name = get_original_name search_obj.parent.name
                )
                else
```

```
(
    local group_head_name = "SceneRoot"
)
-- Quando um grupo pai tem "entity" no nome o original_name devolve o nome do grupo com "entity" que é desnecessário para essa nomenclatura
if findstring group_head_name "entity:" == 1 do
(
    group_head_name = substitutestring group_head_name "entity:" ""
)

--Monta a nomenclatura final do nome
local obj_name = get_original_name search_obj.name
local group_name_final = "Group" + ":" + obj_name + "@" + group_head_name
-- Agrupa todos os nós no array com o nome definido no group_name_final
group group_node_arr name: group_name_final

--if isOpenGroupHead(getnodebyname group_name_final) == false do setgroupopen(getnodebyname group_name_final) true
setgroupopen(getnodebyname group_name_final) true
num_group = num_group + 1

--xml_entity_writer group_name_final obj_name
    )
)
```

```
                    )
                )
                return num_group
            )
            --organize_nodes(rootnode)
--
================================================================================
==================================
--      ************************************::                          INTERFACE
::**********************************************
--
================================================================================
==================================

--==================== INTERFACE : HELP ============================
rollout PostProductionHelp ":: Help ::" width: 290 height:245
(
    label lblh1 "1 . Import the model from inventor" pos:[20,47] width:243 height:23
    label lblh2 "2 . Press 'Start Post-Production' button" pos:[20,67] width:243 height:23
    label lblh3 "Wait the script finish. It may took some minutes" pos:[40,87] width:243
height:23
    label lblh4 "3. Press 'Convert Materials' button and wait" pos:[20,117] width:243
height:23
    label lblh5 "4. Press 'Validate Names' button and wait" pos:[20,147] width:243
height:23
    label lblh6 "Now your model is ready to be exported. Please refeer to the manual for
more detailed instructions on exporting" pos:[20,177] width:243 height:47
    label lblh7 "Using the Post-Production Tool" pos:[12,19] width:243 height:23
)
--==================== INTERFACE : OPTIONS ======================
```

```
rollout PostProductionOptions "Options" width:255 Height:60
(
	checkbox debug_box "Debug mode" pos:[25,15] width:101 height:20
	button help_btn "Help" pos:[180,10] width:60 height:25

	local help_floater

	on help_btn pressed  do
	(
		if help_floater != undefined do
			closerolloutfloater help_floater

		help_floater = newRolloutFloater ":: Help ::" 300 260
		addRollout PostProductionHelp help_floater
	)
)

--====================	INTERFACE	:	POST	PRODUCTION
====================================

rollout PostProduction "Inventor Post-Production" width:255 height:340
(
	button generate_btn "Start Post-Production" pos:[22,20] width:200 height:32
toolTip:"import model from inventor then click here"
	progressBar progress_bar "" color:[255,104,3] pos:[25,105] width:200 height:10
	label timeElapsed "Progress" offset:[0,0] pos:[25,90]
	checkbox selection_box "Selected objects only" pos:[25,60] width:128 height:18
```

```
	on generate_btn pressed do
	(
		start_post_production()
	)
)

--==================== INTERFACE : OGRE VALIDATION ====================================
rollout NameValidation "Ogre Export Validation" width:255 height:200
(
	button convert_mt_btn "Convert Materials" pos:[22,15] width:200 height:25 toolTip:"validate materials before export to Ogre"
	button validate_btn "Validate Names" pos:[22,45] width:200 height:25 toolTip:"validate names before export to Ogre"
	label ruler_1 "..." --this is gambi design

	on convert_mt_btn pressed do convertMaterials()
	on validate_btn pressed do validate_names()

)

rollout OrganizeNodes "Organize Nodes" width:255 height:200
(
	button organize_node_btn "Organize Nodes" pos:[22,15] width:200 height:25 toolTip:"organize all nodes in groups by node names"
	button generate_entities_btn "Generate Entities" pos:[22,45] width:200 height:25 toolTip:"generate xml file from schematic view model"
	on organize_node_btn pressed do
	(
```

```
        local num_group
        if selection.count ==0 then
        (
                num_group = organize_nodes rootnode
        )
        else
        (
                local k=1
                while k<=selection.count do
                (
                        node_organize =selection[k]
                        num_group = organize_nodes node_organize
                        k=k+1
                )
        )

        if PostProductionOptions.debug_box.checked do
        (
                debug = newScript()
                format "**** Grouped Objects : % \n" num_group to: debug
        )
        schematicviews.open "Schematic View 1"
)

on generate_entities_btn      pressed do
(
        if PostProductionOptions.debug_box.checked do
```

```
		local num_entities
		local path_xml_file = getOpenFileName caption:"Open Graphical Entity XML
File:" \
		filename:"C:/Users/marcos/Documents/3dsmax/Conversion Script/" \
		types: "Xml Files(*.xml)|*.xml|All Files (*.*)|*.*|"
		local xml_file = openfile(path_xml_file) mode:"wt"
		if selection.count ==0 then
		(
			num_entities = generate_entities rootnode xml_file
		)
		else
		(
			local k=1
			while k<=selection.count do
			(
				node_selected =selection[k]
				num_entities = generate_entities node_selected xml_file
				k=k+1
			)
		)
		close xml_file
		if PostProductionOptions.debug_box.checked do
		(
			debug = newScript()
			format "**** Entities Generated : % \n" num_entities to: debug
		)
		messagebox "Entities has been successfully generated!"
	)
```

```
)


--
================================================================================
==================================
--      INTERFACE FINALIZATION:
--
-- create the rollout window and add the  rollout
--
================================================================================
==================================
if post_production_floater != undefined do
        closerolloutfloater post_production_floater

post_production_floater = newRolloutFloater ":: Inventor Post Production ::" 260 430

addRollout PostProduction post_production_floater
addRollout NameValidation post_production_floater
addRollout OrganizeNodes post_production_floater
addRollout PostProductionOptions post_production_floater --rolledup:true
--
================================================================================
==================================
)
```