UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
SETOR DE ENGENHARIA MECÂNICA

DAVID CUSTÓDIO DE SENA

# Desenvolvimento de um Núcleo de Simulador de Eventos Discretos para Sistemas de Manufatura com visualização 3D

São Carlos 2010

DAVID CUSTÓDIO DE SENA

# Desenvolvimento de um Núcleo de Simulador de Eventos Discretos para Sistemas de Manufatura com visualização 3D

Dissertação apresentada à Escola de Engenharia de São Carlos da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Mecânica

Área de Concentração: Manufatura

Orientador: Prof. Dr. Arthur José Vieira Porto

São Carlos

2009

Ficha catalográfica

Nome: SENA, David Custódio de

Título: Desenvolvimento de um Núcleo de Simulador de Eventos Discretos para Sistemas de Manufatura com visualização 3D

Dissertação apresentada à Faculdade de Engenharia Mecânica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia Mecânica

Aprovado em:

Banca examinadora:

Prof. Dr. _____ Instituição: _____
Julgamento: _____ Assinatura: _____

Prof. Dr. _____ Instituição: _____
Julgamento: _____ Assinatura: _____

Prof. Dr. _____ Instituição: _____
Julgamento: _____ Assinatura: _____

DEDICATÓRIA

A minha mãe Acácia Lopes Custódio (*in memmorian*) por ter me dado a vida e servir como exemplo de um ser humano especial na vida de tanta gente.

# AGRADECIMENTOS

A minha mãe Acácia Lopes Custódio (*in memorian*) e a meu pai Lucarine Alves de Sena por me darem a vida e os ensinamentos do dia-a-dia e a minha irmã Nara Juliana Custódio de Sena por mostrarem que a família é a base de tudo.

A meus tios e tias em especial a Ires, Francinete, Diana, Hígia, Lunardo, Fátima, Lívia, Jefferson, Elmar, Ceres e todos os outros, meus primos e primas e demais parentes, a todos o meu muito obrigado pelo na minha jornada.

A minha noiva Sarah Maria Veras Bezerra por ser a companhia por demais especial nesses agradáveis 6 anos e 6 meses. Muito obrigado por tudo.

A meus amigos, em especial Gustavo, André, Saulo, Jorge, Abraão, Anselmo, Camila, Nara, Rafaela, Robson, Mário, Tia, Patrícia, Mita, Samia, Aline, Débora, Rúbens, Hilano, Rafael, Shin, Beth, Heráclito, Breno, Marcelo, Flávio, Yuri, Felipe, João, Professora Ana, Professor Belo, Professor Maxwell e tantos outros não mencionados mas que possuem um lugar especial pra mim. Esses representam muitos outros que também participaram.

Ao professor doutor Arthur José Vieira Porto, pela sua dedicação e orientação que me guiou nessa trilha do conhecimento e aos outros professores da USP São Carlos que contribuíram no seus papéis pedagógicos.

E a todo mundo que participou direta e indiretamente da minha vida, um muito obrigado.

RESUMO

SENA, D. C. **Desenvolvimento de um Núcleo de Simulador de Eventos Discretos para Sistemas de Manufatura com visualização 3D.** 2009. 172 p. Dissertação (Mestrado) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2009.

É crescente a necessidade de se conhecer e controlar o ambiente fabril. Ao longo do século passado e início deste, várias ferramentas e soluções foram desenvolvidas para suprir essa necessidade. Dentre elas, a simulação desempenha um suporte para o apoio da decisão amplamente utilizada principalmente na indústria manufatureira. A realidade virtual pode ser utilizada no ambiente de simulação como um canal de visualização e interação do usuário com o meio simulado. O objetivo deste trabalho é modelar uma biblioteca do núcleo de simulador de eventos discretos para sistemas de manufatura, com visualização tridimensional, que funcione em ambientes imersivos e não-imersivos. Para tal, as abordagens de três fases e orientada a objetos foram utilizadas com algumas alterações. Para a validação desse desenvolvimento, foram feitas duas simulações de um aplicativo que utiliza os elementos básicos de manufatura e foi feita a coleta de seus resultados que possibilitaram a verificação dos objetivos pretendidos. Por fim, foi feita a análise dos resultados e são apresentadas propostas de trabalhos futuros nesta área.

Palavras-chave: Gerência da Produção. Realidade Virtual. Diagrama de Três Fases. Simulação Orientada a Objetos.

ABSTRACT

SENA, D. C. **Development of a Discrete Event Simulation Kernel for Manufacturing Systems with 3D visualization.** 2009. 172 p. Dissertação (Mestrado) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2009.

The need of knowledge and control of the manufacturing environment is continuously growing. Over the last century and the beginning of this, several tools and procedures were designed in order meet those necessities. Among them, simulation is a decision support tool widely used, mainly in the manufacturing industry. Virtual reality can be used in those simulations for user visualization and interaction with the simulated environment. The aim of this research was to model a library for a discrete event simulator core of a manufacturing system, with 3D visualization, that can be used in immersive and non-immersive environments. Two different approaches have been used: the three phases and the object-oriented one. To validate the software development, two simulations were carried out for an application that uses basic elements of manufacturing and production. Data was collected and analyzed in order to check the accomplishment of the research objectives. Finally, a conclusion about the results is presented along with some proposals for future work in this area.

Keywords: Production Planning. Virtual Reality. Three-phases Diagram. Object Oriented Simulation.

LISTA DE FIGURAS

Figura 1 – Sistema de fila de simples servidor ..... 32
Figura 2 - Controle de fluxo abordagem de agendamento de eventos / algoritmo de avanço no tempo (Law & Kelton, 1999, p. 10) ..... 34
Figura 3 - B's e C's em estados ativados ..... 36
Figura 4 - Algoritmo de execução da abordagem de três fases(Pidd, 2004) ..... 37
Figura 5 - Estrutura conceitual de simulações orientadas a objetos (Joines & Roberts, 1996) ..... 38
Figura 6 - Etapas que compõem um sistema de manufatura ..... 42
Figura 7 - Organização hierárquica de um sistema de manufatura ..... 46
Figura 8 - Lógica global da simulação ..... 52
Figura 9 - Gerador de agendamento otimizado (Chien & Chen, 2007, p. 1767) ..... 54
Figura 10 – Rastreador ..... 57
Figura 11 - Mouse 3D ..... 57
Figura 12 - Luva ..... 58
Figura 13 - Óculos de visão 3D estéreo ativo ..... 58
Figura 14 - Monitor CRT ..... 59
Figura 15 - Head Mounted Display ..... 59
Figura 16 - Binocular Omni-Orientation Monitor ..... 60
Figura 17 - Tela panorâmica ..... 60
Figura 18 - Mesa Virtual ..... 61
Figura 19 - CAVE ..... 63
Figura 20 - Operação conceitual das duas chamadas da AMVE(Jones *et al.*, 1993, p. 884) ..... 67
Figura 21 - Definição de classe ..... 71
Figura 22 - Exemplo de diagrama de classe ..... 72
Figura 23 - Exemplo de um diagrama de sequência ..... 73
Figura 24 - Aplicação e Interface PV (Bierbaum *et al.*, 2001) ..... 75
Figura 25 - Exemplo de grafo de cena ..... 77
Figura 26 - Fluxograma da lógica de funcionamento do sistema de manufatura ..... 79
Figura 27 - Dimensões das telas de multi projeção ..... 82
Figura 28 - Unidades básicas do simulador ..... 83
Figura 29 - Camadas do simulador ..... 85

Figura 30 - Diagrama de classes....86
Figura 31 - Diagrama de sequência da lógica....89
Figura 32 - Diagrama de sequência da geração de objetos gráficos....91
Figura 33 - Protótipo do núcleo do simulador....92
Figura 34 - Arranjo físico global....95
Figura 35 - Processo de manufatura do produto Product1....95
Figura 36 - Processo de manufatura do produto Product2....95
Figura 37 - Processo de manufatura do produto Product3....96
Figura 38 - Tempo de processamento do produto Product1....104
Figura 39 - Tempo de processamento do produto Product2....104
Figura 40 - Tempo de processamento do produto Product3....105
Figura 41 - Visualização do aplicativo do simulador em Desktop....106
Figura 42 - Visualização do aplicativo do simulador em CAVE....106

LISTA DE TABELAS

Tabela 1 Características gerais de tipos de layout ..... 49
Tabela 2 – Campos de Imersão ..... 61
Tabela 3 - Campos de Visualização ..... 62
Tabela 4 - Valores dos elementos da lógica da manufatura ..... 80
Tabela 5 - Equipamentos da aplicação ..... 94
Tabela 6 - Equipamento Conv1 ..... 97
Tabela 7 - Equipamento Conv2 ..... 97
Tabela 8 - Equipamento Conv3 ..... 98
Tabela 9 - Equipamento Conv4 ..... 98
Tabela 10 - Equipamento Buf1 ..... 99
Tabela 11 - Equipamento Buf2 ..... 100
Tabela 12 - Equipamento Turning ..... 101
Tabela 13 - Equipamento Milling ..... 101
Tabela 14 - Equipamento Insp1 ..... 102
Tabela 15 - Equipamento Insp2 ..... 103

# LISTA DE SIGLAS

| | |
|---|---|
| CAVE | Caverna digital |
| CSL | *Control and Simulation Language* |
| CAD | *Computer Aided Design* |
| FMS | *Flexible Manufacturing System* |
| CNC | *Computer numerically controlled* |
| AGV | *Automated guided vehicle* |
| OptSG | *Optimization-based Schedule Generator* |
| RV | Realidade Virtual |
| HMD | *Head Mounted Display* |
| BOOM | *Binocular Omni-Orientation Monitor* |
| CRT | *Cathode Ray Tube* |
| VEOS | *Virtual Environment Operating System* |
| AMVE | *AutoMod Virtual Environment* |
| KAMVR | *Knowledge Acquisition and Management using Virtual Reality* |
| UML | *Unified Modeling Language* |
| PV | Plataforma Virtual |
| API | *Application Programming Interface* |
| LMVI | Laboratório Multiusuário de Visualização Imersiva |

SUMÁRIO

1. INTRODUÇÃO ..... 27
1.1 Objetivos ..... 28
1.1.1 Objetivo geral ..... 28
1.1.2 Objetivos específicos ..... 28
1.2 Estrutura ..... 29
2. REVISÃO BIBLIOGRÁFICA ..... 30
2.1 Simulação ..... 30
2.1.1 Tipos de Simulação ..... 32
2.1.2 Paradigmas de Simulação Computacional ..... 34
2.1.3 Linguagem de simulação ..... 39
2.2 Manufatura ..... 42
2.2.1 Sistema de Manufatura ..... 43
2.2.2 Tipos de Manufatura ..... 45
2.2.3 Sistema de Manufatura Flexível ..... 49
2.3 Simulação de Sistemas e Manufatura ..... 50
2.3.1 Modelagem e simulação de uma peça e controle de ferramenta em FMS ..... 50
2.3.2 Modelo baseado em otimização de gerador de agendamento para um sistema de agendamento de produção genérico ..... 53
2.4 Realidade Virtual ..... 55
2.4.1 Dispositivos de Realidade Virtual ..... 56
2.4.2 CAVE ..... 63
2.5 Simulação de sistemas, manufatura e realidade virtual ..... 65
2.5.1 Realidade virtual para sistemas de manufatura ..... 65
2.5.2 Construção de um ambiente virtual para simulação de manufatura ..... 68
2.5.3 Implementação de simulação de eventos em um ambiente virtual ..... 69
3. DESENVOLVIMENTO DE UM NÚCLEO DE SIMULADOR DE EVENTOS DISCRETOS PARA SISTEMAS DE MANUFATURA COM VISUALIZAÇÃO 3D 70
3.1 Ferramentas ..... 70
3.1.1 Linguagem de modelagem unificada ..... 70
3.1.2 VR Juggler ..... 74
3.1.3 OpenSG ..... 76

3.2 Definições de sistema de manufatura estudado ..... 77
3.3 Definições do núcleo do simulador de sistema de manufatura ..... 80
3.4 Definições de ambiente de realidade virtual ..... 81
3.4.1 Elementos do ambiente de realidade virtual ..... 81
3.4.2 Laboratório multiusuário de visualização imersiva ..... 82
3.5 Modelagem computacional ..... 83
3.5.1 Diagrama de Classe ..... 85
3.5.2 Diagramas de Sequência ..... 88
3.6 Prototipação ..... 91
4. TESTE DE APLICAÇÃO DA PROPOSTA ..... 94
4.1 Composição ..... 94
4.2 Resultados ..... 96
5. CONCLUSÃO ..... 108
5.1 Proposta para trabalhos futuros ..... 109
6. REFERÊNCIAS ..... 110
APÊNDICE A – CLASSES ..... 113

# 1. INTRODUÇÃO

A necessidade de se conhecer e controlar o ambiente fabril é cada vez maior. A concorrência a nível mundial, a necessidade de produtos com maior qualidade e diversificados e o controle e diminuição das perdas são algumas causas desse aumento da necessidade de gestão interna.

Nesse sentido, muitas ferramentas e soluções estão sendo desenvolvidas para suprir essa carência, desde a conformação da linha de montagem, feita por Ford, passando por algumas mais recentes desenvolvidas pelos japoneses, como o *Just-in-time* e a produção enxuta. Cada qual com suas especificidades.

Ainda nesse contexto, a simulação é uma ferramenta de apoio a decisão amplamente utilizada na indústria manufatureira. Ela auxilia desde o projeto de novas aquisições de equipamentos e novos produtos até a adequação da capacidade produtiva para atender à demanda existente.

Dentro do cenário da simulação, a realidade virtual é utilizada como um canal de visualização e interação do usuário com o meio simulado. A visualização pode ser de dois tipos: não-imersivo e imersivo, diferindo entre si pela capacidade de capturar a atenção dos sentidos do espectador, geralmente audição ou tato, além do tradicional visão.

Algumas soluções que utilizaram tanto simulação quanto realidade virtual podem ser encontradas, incluindo algumas comerciais, como o Automod. Entretanto, a maioria delas trabalha a realidade virtual separadamente, segregando-a da simulação. A junção das duas pode trazer várias vantagens, dentre elas: a capacidade de comunicação de ambientes virtuais mais complexos, com interação e com a simulação; a sensação de proximidade do ambiente fabril virtual da realidade; dentre outros.

## 1.1 Objetivos

### 1.1.1 Objetivo geral

O objetivo geral deste trabalho é modelar um núcleo de simulador de eventos discretos para sistemas de manufatura com visualização tridimensional que funcione em ambientes imersivos e não-imersivos.

O núcleo do simulador será caracterizado pelo conjunto de classes que possuam as operações básicas que um simulador de eventos discretos precisa executar, centralizado em sua lógica e a visualização do resultado a partir de ambientes tridimensionais imersivos e não imersivos. Outras operações como a composição de relatórios, a geração de números aleatórios, a interface gráfica entre o usuário e o programa, dentre outras que formam um simulador convencional não serão abordadas nesse texto.

### 1.1.2 Objetivos específicos

- ➔ Elaborar uma lógica de simulador de eventos discretos para sistemas de manufatura;
- ➔ Desenvolver uma estrutura de interação dos elementos virtuais para visualização em ambiente 3D;
- ➔ Implementar os objetivos anteriores em linguagem C++;
- ➔ Testar a aplicação em um ambiente de manufatura que utilize os elementos básicos e comentar seus resultados.

## 1.2 Estrutura

Esta dissertação está dividida em cinco capítulos: (1) Introdução; (2) Revisão bibliográfica; (3) Desenvolvimento de um núcleo de simulador de eventos discretos para sistemas de manufatura com visualização 3D; (4) Teste de aplicação da proposta; e (5) Conclusão.

Na revisão bibliográfica far-se-à vistas sobre o que já foi escrito nos assuntos relacionados e explora-se alguns trabalhos similares à presente dissertação.

O desenvolvimento de um núcleo de simulador de eventos discretos para sistemas de manufatura com visualização 3D apresentará as ferramentas, requisitos, modelagem computacional e prototipação.

No teste de aplicação da proposta será desenvolvido um ambiente virtual de sistema de manufatura que utilize todos os elementos básicos e serão apresentados os resultados de duas simulações distintas, a primeira para a análise dos equipamentos e a segunda para análise dos processos de manufatura de cada produto.

A conclusão conterá informações dos resultados, o posicionamento na investigação e recomendações para trabalhos futuros.

# 2. REVISÃO BIBLIOGRÁFICA

Neste capítulo, as bibliografias básicas utilizadas nesse trabalho serão apresentadas. Além dessa revisão dos temas fundamentais, uma revisão sobre trabalhos correlatos será feita.

Três temas serão tratados na sequência. O primeiro tema a ser abordado será os aspectos teóricos da simulação de sistemas, acerca dos conceitos fundamentais dos seus elementos constituintes, dos tipos, dos paradigmas e das linguagens de simulação. O segundo tema refere-se à definição e à classificação de manufatura e mais particularmente aquilo que se refere ao sistema de manufatura flexível. O terceiro tema é a teoria e a prática da realidade virtual com seus conceitos, dispositivos e a caverna digital *(Cave Automatic Virtual Environment* – CAVE).

Por fim, trabalhos que utilizaram os três temas supracitados serão apresentados.

## 2.1 Simulação

Simulação é uma forma interativa de espelhar os fatos reais, representando o que já existe, e que utiliza modelos lógicos manipuláveis matemática e computacionalmente. Suas características possibilitam a construção de um modelo que é usado para se tentar aprender o comportamento do sistema real correspondente (Law & Kelton, 1999).

Segundo Banks *et al* (1996), simulação é a imitação da operação de um processo de mundo real que geralmente ocorre pelo tempo. Por intermédio da simulação, soluções para uma diversidade de problemas podem ser encontradas sem intervir diretamente no objeto estudado. Ainda segundo tais autores, o modelo representa o comportamento do sistema através de expressões matemáticas, lógicas e simbólicas entre as entidades que o compõe.

Um sistema é definido como um conjunto de partes e entidades que, interagindo entre si, tentam atingir um determinado objetivo comum (Law & Kelton, 1999). Para Banks *et al* (1996), a diferença entre simulação e otimização é que este é resolvido e retorna um resultado exato, ao contrário daquele, que é executado e seu resultado precisa ser interpretado pelo usuário.

Os elementos básicos presentes na teoria da simulação são (Law & Kelton, 1999) :

- O modelo: representação do sistema em estudo, servindo para correlacionar os fatos mais importantes referentes ao sistema em estudo. Também é utilizado para avaliar os efeitos de diferentes políticas que podem ser aplicadas no último;

- Entidades: representações de objetos bem definidos do mundo real. Dessas, têm-se ainda as entidades permanentes e as entidades temporárias. As primeiras são criadas no começo da simulação e subsistem enquanto essa estiver executando. As segundas podem tanto ser criadas no começo quanto ao longo da simulação e ser destruídas antes do término da mesma;

- Recursos: elementos pertencentes ao sistema, mas que geralmente não possuem características individuais estudados no modelo. Ao invés disso, eles são tratados como itens contáveis em que os seu procedimentos não são rastreados pelo programa de computador. Muitas vezes, as entidades temporárias se caracterizam como recursos;

- Classes: grupos permanentes de entidades idênticas e similares;

- Conjuntos: grupos temporários de entidades diferentes que são frequentemente usados para representar estado de entidades e filas;

- Atributo: propriedade de uma entidade. O conjunto dos atributos que possuem valores definidos e distintos faz a identificação da entidade no sistema;

- Variável: propriedade da entidade que pode receber qualquer valor dentro do intervalo especificado;

- Atividade: período de comprimento especificado, também chamado de espera não condicional. Dessas, as atividades endógenas são partes integrantes do sistema, enquanto as atividades exógenas correspondem aos eventos de ambiente que podem influenciá-lo.

- Espera: período que não possue comprimento de tempo especificado, também chamado de espera condicional. É necessária a confirmação de uma condição para que ocorra o fim da espera. Por exemplo, um cliente em uma fila de banco só será atendido se um atendente estiver livre.

- Estado de um sistema: coleção de variáveis necessárias para descrever um sistema em um momento do tempo específico. Por exemplo, para um sistema de espera de atendimento em um banco, onde se caracteriza uma fila, o estado do sistema pode ser definido como o número de atendentes e a quantidade de pessoas na fila.

- Evento: ocorrência instantânea que muda o estado de um sistema. Geralmente, uma atividade é limitada por dois eventos: um para seu início e um para seu final.

- Lista: coleção (permanente ou temporária) de entidades associadas ordenada logicamente.

- *Clock*: variável que representa um tempo global na simulação.

Um exemplo que ilustra alguns elementos principais relacionados acima é quando há o sistema de um simples servidor para o atendimento de clientes que esperam em uma fila. Nele, clientes e servidor ilustram entidades, aqueles, entidades temporárias, ou recursos, e esses, entidade permanente. Os atributos podem ser o nome, a idade e qualquer outra informação de cada cliente. A fila dos clientes esperando para serem atendidos caracteriza uma lista. A figura 1 ilustra este exemplo de sistema de fila de simples servidor.

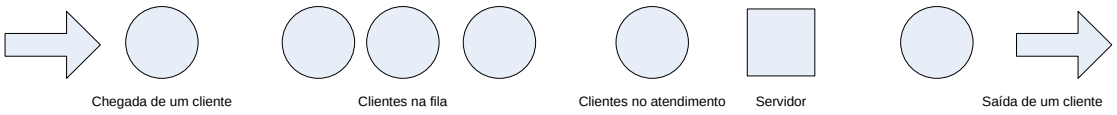


Figura 1 – Sistema de fila de simples servidor

No exemplo da figura 1, os eventos são: a chegada de um cliente, o atendimento de um cliente e a saída de um cliente.

O estado do sistema da figura é composto por: um cliente entrando no sistema, três clientes esperando para serem atendidos, um cliente sendo atendido pelo únivo servidor e um cliente saindo do sistema. A variável quantidade de clientes na fila irá influenciar o tempo que o próximo cliente terá que esperar para ser atendido.

### 2.1.1 Tipos de Simulação

A classificação dos tipos de simulação não é uma atividade precisa, pois geralmente os comportamentos dos sistemas são combinados por diferentes tipos, isso devido ser raro o

encontro de características “puras” pertencentes a um único tipo. Assim, a simulação de sistemas é classificada basicamente em três grupos distintos (Law & Kelton, 1999):

a) Simulação de sistemas discretos x simulação de sistemas contínuos

A simulação de sistemas discretos é aquela em que a(s) variável(is) de estado mudam apenas em conjuntos de pontos discretos de tempo; para fins de analogia, diz-se que ocorrem “saltos” no tempo, que, na verdade, correspondem aos avanços nos eventos seguintes.

Já a simulação de sistemas contínuos é aquela em que a(s) variável(is) de estado podem assumir quaisquer valores dentro de um intervalo real ao longo do tempo da simulação. A simulação de sistemas contínuos comumente vale-se de equações diferenciais.

b) Simulação de sistemas estáticos x simulação de sistemas dinâmicos

Um modelo estático, também conhecido como método de Monte-Carlo, não leva em consideração o tempo quando for executado. Para Pidd (2004) as simulações de Monte-Carlo trabalham com situações de riscos, onde não havendo certeza do que irá acontecer, pode-se construir uma distribuição de probabilidade, e situações que envolvem incertezas, as quais não apresentam formas objetivas de construir uma distribuição de probabilidade de sua ocorrência.

Simulador de sistemas dinâmicos representa modelos com mudanças de estado por onde passam os elementos constituintes do sistema, em que o tempo é considerado como um fator essencial.

c) Simulação de sistemas determinísticos x simulação de sistemas probabilísticos

Na simulação de sistemas determinísticos, os valores das variáveis são constantes. Quando ocorrem replicações dos modelos de simulação desses sistemas, os resultados serão os mesmos.

Já na simulação de sistemas probabilísticos as variáveis podem assumir quaisquer valores dentro de intervalos definidos por distribuições de probabilidade.

Uma vez que os sistemas de manufatura possuem características predominantes de sistemas discretos, com a ocorrência de filas e esperas entre intervalos de tempo determinados pelos processos que os compõem, dinâmicos, pois suas entidades têm influência do tempo em suas atividades, e probabilísticos, pois a incidência de sistemas determinísticos só pode ocorrer em sistemas muito controlados e que não sofra variabilidade, este trabalho faz uso da simulação computacional a eventos discretos, dinâmicos e probabilísticos.

### 2.1.2 Paradigmas de Simulação Computacional

Os paradigmas de simulação computacional servem para orientar conceitualmente o modelador ao longo do projeto da simulação. Em geral, a lógica da simulação segue as seguintes abordagens:

a) Abordagem de agendamento de evento / algoritmo de avanço no tempo:

A sequência das futuras ações é explicitamente codificada no simulador e colocada numa lista de eventos futuros, onde aquelas vão sendo executadas no exato momento de ocorrência dos eventos pelo *clock*. A figura 2 apresenta a lógica do controle de fluxo de agendamento de eventos.

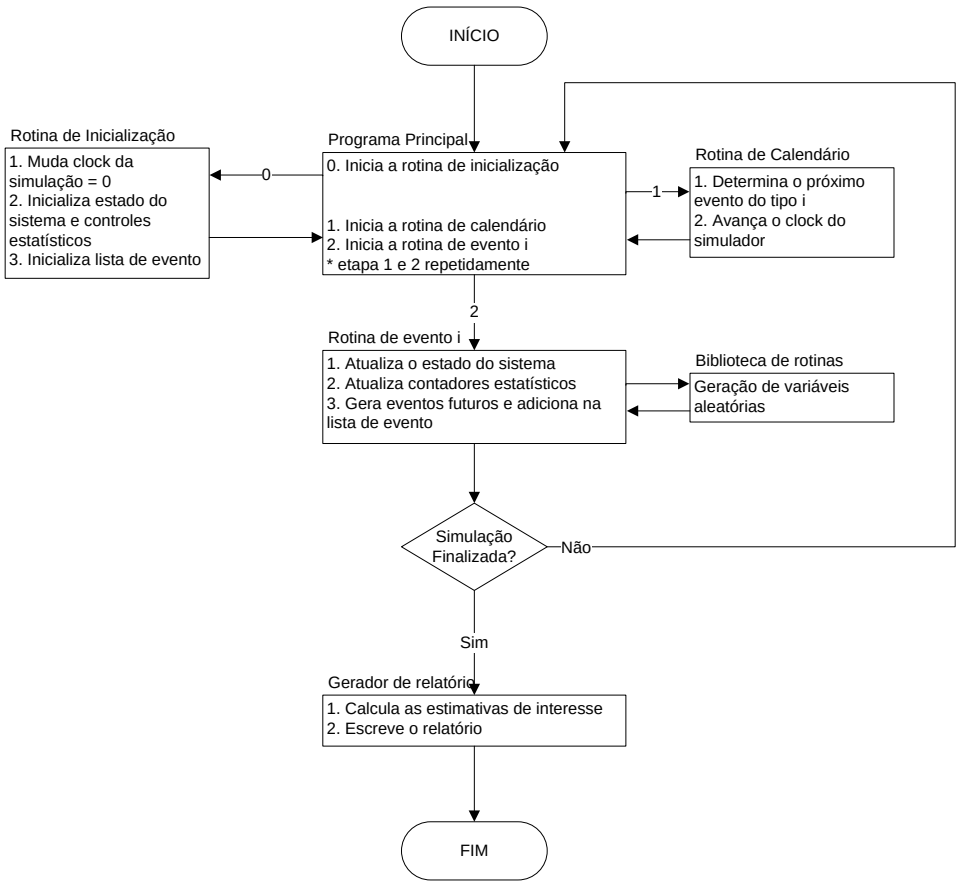


Figura 2 - Controle de fluxo abordagem de agendamento de eventos / algoritmo de avanço no tempo (Law & Kelton, 1999, p. 10)

O fluxograma descrito na figura 2 mostra os passos que devem ser executados antes do início, durante e na finalização da simulação.

O tamanho da lista de eventos futuros é constantemente alterado com o progresso da simulação. O gerenciamento da lista é chamado de processamento da lista. As suas principais operações consistem em remover um evento eminente, adicionar um novo evento à lista, e a remoção de algum evento futuro que não seja mais necessário.

b) Abordagem de procura de atividade

Utiliza um incremento de tempo fixo e uma abordagem baseada em regras para decidir se qualquer atividade pode começar em qualquer ponto do tempo simulado. Um modelador concentra nas atividades de um modelo e suas condições, simples ou complexas, que possibilita uma atividade começar. Em todo avanço do *clock*, as condições para cada atividade são checadas e se forem verdadeiras, então a atividade correspondente inicia. Não é recomendada para a utilização em grandes modelos, por tornar o sistema lento e pesado. Já seu uso em pequenos modelos pode ser utilizado como ferramenta de ensino.

c) Abordagem de processo

O analista define o modelo de simulação em termos do fluxo do ciclo de vida de entidade através do sistema. Um processo é uma sequência temporal da lista de eventos, atividades e esperas, incluindo demanda e espera de recursos.

A simulação é vista em termos de entidades individuais envolvidas, e o modelo é construído observando como cada uma atravessa o sistema.

Por proceder a modelagem de todo o sistema de uma só vez, esta abordagem apresenta-se inflexível, pois qualquer alteração do sistema exige uma mudança na lógica do modelo.

d) Simulação paralela e distribuída

Nessa abordagem, segundo Law & Kelton (1999), é possível distribuir diferentes partes de uma atividade computacional entre processadores individuais num mesmo tempo, ou em paralelo. Para os autores, a melhor forma de trabalhar com esse paradigma é alocar as funções de suporte (como geração de números e variáveis aleatórios, manuseio de lista de eventos, manipulação de lista e fila e coleção estatística) para diferentes processadores.

Outra maneira de distribuir uma simulação entre processadores é decompor o modelo em submodelos distintos, que depois são atribuídos para diferentes processadores.

Sua aplicação torna-se complexa devido a exigência de uma supervisão no controle, para que não haja conflito na mesma informação que podem ser trabalhadas por unidades distintas e também devido a limitação física da utilização de mais de um processador.

e) Abordagem das três fases

Segundo Pidd (2004), trata-se de uma abordagem semelhante à abordagem de procura de atividade, porém com algumas modificações que permitem que ela seja aplicada em modelos maiores. As atividades são divididas em duas: as atividades C's são condicionais, uma vez que precisam de checagens de condições de estado para poderem ser executadas; e as atividades B's são incondicionais, elas são manipuladas por um controlador, pois seu tamanho é conhecido. A figura 3 ilustra as ocorrências das atividades B's e C's.

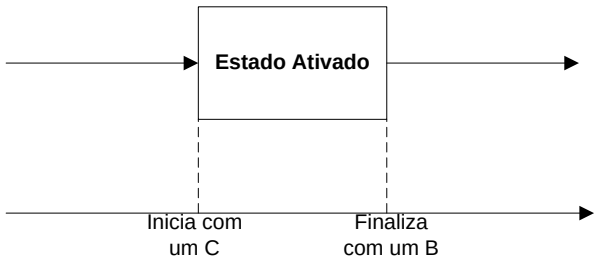


Figura 3 - B's e C's em estados ativados

Tal abordagem é assim nomeada devido ao seu funcionamento baseado em três fases distintas. A fase "A" é a atualização do tempo. Tal fase checa a lista para identificar quando será o próximo evento. A fase "B" executa as atividades B's mais imediatas. Finalmente, a fase "C" verifica em todas as atividades C's quais delas tiveram seus testes satisfeitos. A figura 4 apresenta o algoritmo da sequência de execução da abordagem de três fases.

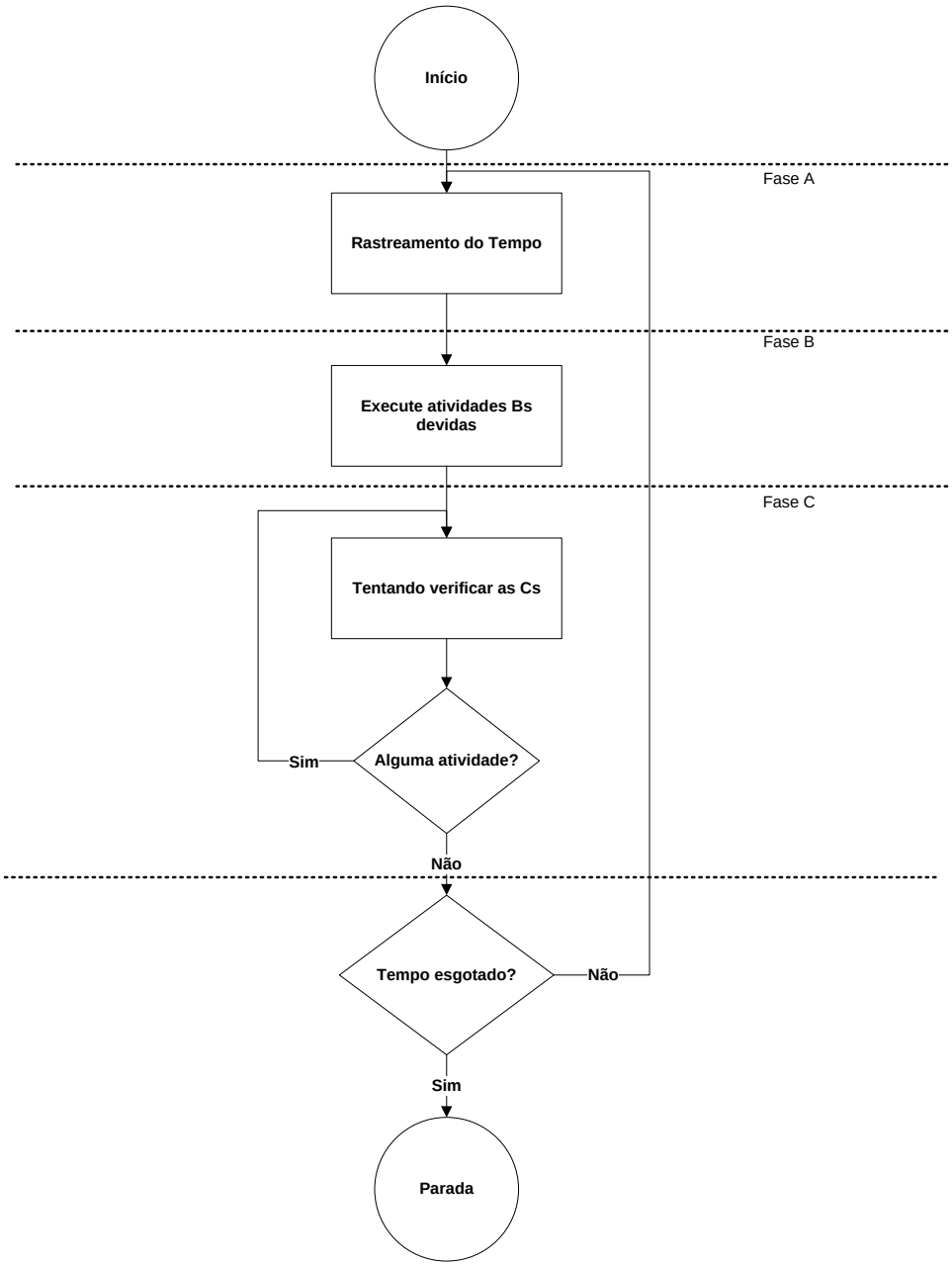


Figura 4 - Algoritmo de execução da abordagem de três fases(Pidd, 2004, p.91)

f) Abordagem orientada a objetos

Paradigma baseado na programação orientada a objetos em que o projetista constrói o modelo observando os elementos abstratos e concretos que compõem o sistema. Devido a essa característica, ele possui um forte apelo intuitivo no desenvolvimento de modelos. (Lomow & Baezner, 1990; Bischak & Roberts, 1991).

Os elementos do simulador possuem seus correspondentes nesta abordagem, quais sejam: as entidades são objetos físicos e ativos compostos por dados, que definem suas características, e funções, ou operações, e seus funcionamentos são independentes e concorrentes; o evento é qualquer mudança que acontece em um objeto e ele é usado tanto para sincronizar as ações de duas entidades como para passar informações de uma entidade para outra; o tempo de simulação sincroniza a execução de todos os eventos que ocorrem nas entidades. A figura 5 apresenta a estrutura conceitual de simulações orientadas a objetos.

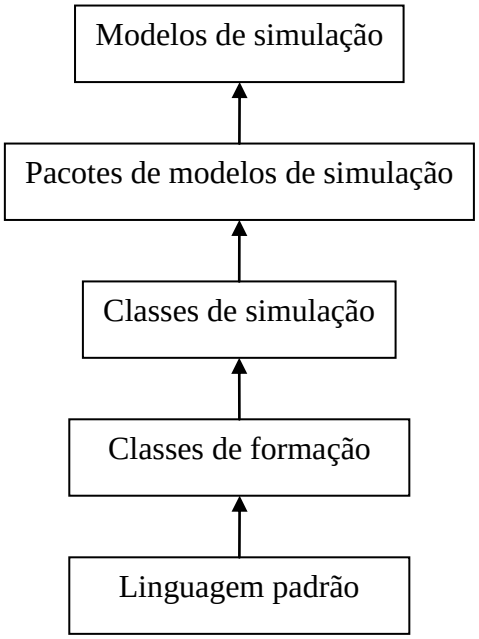


Figura 5 - Estrutura conceitual de simulações orientadas a objetos (Joines & Roberts, 1996)

Na figura 5, a linguagem padrão representa a linguagem de uso geral e está localizada no nível mais baixo. As classes de formação e as classes de simulação representam a abstração dos objetos, estas mais específicas para o ambiente de simulação e aquelas mais genéricas. Os pacotes de modelos de simulação, comerciais ou não, são desenvolvidos para facilitar a utilização por parte do usuário final. Finalmente, os modelos de simulação fazem a interface do simulador (Joines & Roberts, 1996). Nesta estrutura, quanto mais baixo for o nível, maiores serão o controle e a dificuldade para o modelador, ocorrendo o contrário para níveis mais altos.

Para (Bischak & Roberts, 1991, p. 194) “é possível escrever vários tipos de simulação, como por exemplo, a orientada a eventos, usando linguagem de simulação orientada a objetos, nada força o modelador a trabalhar de um modo”. Ademais, há muitas linguagens orientadas a objetos, das quais o modelador pode escolher a que lhe for mais adequada.

Uma vantagem na utilização da simulação orientada a objetos é a possibilidade de paralelismo em sua execução. Ela permite, inclusive, a separação da sua implementação em mais de um processador.

### 2.1.3 Linguagem de simulação

As linguagens de simulação compõem os recursos computacionais pelos quais o projetista constrói os modelos de simulação no computador digital. Nesta seção, algumas das linguagens mais conhecidas e as suas características principais serão abordadas.

a) Linguagens de propósito geral

As linguagens de propósito geral são aquelas que servem para fazer aplicações gerais, não sendo restritos apenas às de simulação. Geralmente servem para desenvolver modelos orientados a eventos. Nelas, o projetista do simulador deve implementar todos os detalhes de programação de evento/ algoritmo avanço de tempo, a capacidade de coletar dados estatísticos, a geração de exemplos de uma distribuição de probabilidade específica e o gerador de relatório. Como exemplos, as seguintes linguagens de propósito geral podem ser citadas: Fortran, Visual Basic, C, C++, Java, dentre outras.

b) GPSS

Linguagem de simulação de propósito especial, altamente estruturada, utilizada na abordagem de interação de processo e orientada por um sistema de fila, foi desenvolvida inicialmente pela IBM em 1961. Uma maneira conveniente de descrever um sistema é utilizando a técnica dos diagramas de blocos. As entidades chamadas de transações podem ser vistas como fluindo através do diagrama de bloco. O diagrama de blocos é preparado numa

forma de reconhecimento do computador durante declarações e a simulação é realizada pelo processador.

c) SIMULA

Linguagem desenvolvida na década de 1960. Possui duas versões, a I e a 67. Para Nance (1996), a SIMULA foi desenvolvida para atender as seguintes características: (1) conceito de rede de evento discreto, (2) baseado em linguagem ALGOL, (3) modificação e extensão do compilador UnivacALGOL 60 e a introdução do "conceito de processo", e (4) a implementação do compilador SIMULA I.

d) SIMAN (*SImulation Modeling and ANalysis*)

Linguagem de simulação que tem a capacidade de trabalhar com modelos baseados em interação de processos, agendamento de eventos e simulação contínua, ou uma combinação dos últimos. A plataforma de modelagem do SIMAN é composta de quadros de modelo e experimentação. Ela apresenta as seguintes características: 1. Características especiais para modelar sistemas de manufatura; 2. Ferramentas que habilitam a modelagem de sistemas de manuseio de materiais; 3. Um controle de execução interativa que permite execuções de procedimentos de controle; 4. Menus, procedimentos apontar-e-clicar e outras características que facilitam a interação com o usuário; 5. Portabilidade.

Segundo Pegden & Ham (1982), o SIMAN tem uma divisão fundamental entre o modelo de sistema, que define as características estáticas e dinâmicas, e o frame experimental, que define as condições de experimentação sob as quais o modelo está submetido.

e) SIMSCRIPT II.5

Linguagem de simulação que permite a construção de modelos que podem ser orientados a processo, orientados a evento ou uma combinação de ambos. Ele é utilizado para produzir tanto apresentações dinâmicas e estáticas, como gráficas. A versão para microcomputador e a versão para estação de trabalho possuem um pacote gráfico e de animação denominado SIMGRAPHIC. O modelo gráfico de interação permite que um conjunto de modificações no modelo possa ser feito sem programação, tornando-o mais intuitivo e amigável, podendo ser utilizado mesmo por aqueles que não estão familiarizados com as técnicas de programação.

f) CSL (*control and simulation language* - CSL)

Linguagem de controle e de simulação. A proposta dessa linguagem é auxiliar a solução de problemas complexos de lógica e de decisão complexas.

g) SLAM II

Comercializado por *Pritsker Corporation*, trata-se de uma linguagem de simulação de alto nível, com versões em FORTRAN e C. Ele permite a utilização de abordagens de agendamento de eventos, interação de processos, ou uma combinação dos dois.

h) MODSIM III

Linguagem de simulação orientada a objetos e de propósito geral. Ele é uma linguagem compilada altamente portável.

i) AweSim

Desenvolvida por O'reilly & Lilegdon (1999), AweSim é um aplicativo que suporta um conjunto de ferramentas que executam um projeto de simulador. Possui como característica principal a arquitetura aberta que o permite comunicar-se com outros programas. Foi desenvolvido em Visual Basic e C/C++.

Ademais, o AweSim utiliza em seu funcionamento um ou mais cenários que são alternativas de um sistema particular sendo compostos por componentes. O executivo do AweSim inclui uma rede, uma sub-rede, um controle e componentes de utilização de usuário.

O AweSim utiliza a metodologia de modelagem do Visual Slam. Seu componente básico é a rede ou diagrama de fluxo, que retrata o fluxo das entidades no sistema. Ainda pode utilizar sub-redes para fazer a reutilização de objetos rede.

Por fim, AweSim pode comparar os dados de saída tanto em formato texto quanto em formato gráfico, e essas informações podem ser exportadas para vários tipos de programas.

A partir da animação na linguagem, pode-se desenvolver e exibir múltiplas animações para um simples cenário.

## 2.2 Manufatura

Manufatura é o estudo das etapas necessárias para fazer peças e fabricá-las utilizando máquinas e mecanismos (Danilevsky, 1973). Um processo produtivo é o procedimento necessário para converter os produtos da natureza em artigos de uso para a humanidade (Balakshin, 1971).

Para Porto(1990), os objetivos para se produzir um produto são: definição e preparação da matéria-prima, agregação de valor a essa matéria-prima e comprovação de qualidade do produto e sua liberação para venda. Para que tais objetivos sejam alcançados, faz-se necessário construir um sistema de manufatura. A figura 6 exibe as etapas descritas anteriormente.


Figura 6 - Etapas que compõem um sistema de manufatura

Na figura 6, a definição fornecerá os dados de entrada para o sistema, como os produtos devem ser processados, utilizando-se de vários componentes auxiliares do sistema, tais como: ferramentas de corte, dispositivos de fixação, sensores, etc.

A etapa de transformação cuidará das mudanças geométricas e de forma à peça, agregando valor perceptível aos clientes, e por último há a aprovação da qualidade do produto final e sua liberação para venda.

O processo de manufatura é a parte do processo produtivo diretamente interessado na modificação da forma, da dimensão e das propriedades físicas e químicas do produto e de seus estados qualitativos. Este processo é realizado numa sequência definida, denominada processo de manufatura.

Para Balakshin (1971), o processo de manufatura é subdividido em diversas partes por dois motivos: um econômico e um físico. O motivo físico é devido à impossibilidade de uma

máquina executar todas as operações de um processo de manufatura. O motivo econômico é devido ao alto custo de se projetar máquinas que atendam a muitas especificações, sendo mais vantajosa a utilização de mais de uma máquina para aumentar a abrangência de fabricação de produtos diferentes. As diversas partes do processo de manufatura são as seguintes:

- Operação: porção completa do processo de manufatura de tratamento físico da peça numa estação de trabalho simples, e é acompanhada por um ou mais operadores continuamente. Trata-se do elemento básico no agendamento e controle da produção de um produto. Cada operação necessita de uma porção de tempo denominado ciclo;
- Processo de montagem: parte do processo produtivo diretamente envolvido com uniões consecutivas de partes finalizadas em unidades de montagens e máquinas completas, com uma qualidade que atenda às especificações de manufatura;
- Ajuste da máquina: parte da operação realizada sem o contato direto com o produto. Ela deve ser evitada, pois não agrega valor perceptível ao cliente no produto final;
- Operação de manipulação elementar: cada fase distinta realizada pelo operador requerida para acompanhar uma operação;
- Um posto ou posição de trabalho: parte da operação realizada sem alterar a posição da peça em relação à ferramenta. Um espaço que se necessita para executar certo trabalho por uma máquina e/ou operador.

### 2.2.1 Sistema de Manufatura

Entende-se o termo "sistema de manufatura" como o conjunto das atividades relacionadas que compõem todo o processo produtivo, desde o projeto do produto até a sua produção física. Ele é dividido em cinco funções principais (Askin & Standridge, 1993):

a) Projeto do produto

Responsável por capturar os dados de entrada advindos do mercado e por realizar a descrição do produto para satisfazer as necessidades dos clientes. O CAD é a principal ferramenta computacional utilizada para tal. A descrição do produto pode ser tanto textual como por protótipos, físicos ou virtuais, bidimensionais ou tridimensionais.

b) Planejamento de processo

Tem a ver com a especificação da sequência de operações requeridas para transformar matérias-primas em peças e então montar as peças em produtos. A sequência é denominada de processo de fabricação.

O planejamento de processos pode ser feito de várias maneiras, mas as mais conhecidas são a interativa e a generativa (Lin & Bedworth, 1988).

c) Operações

Já foram definidas anteriormente e são geralmente de natureza de fabricação, ou seja, alteração na forma da peça, e a montagem, que faz a união de várias peças para gerar o produto.

d) Arranjo físico de fábrica

Relacionado com o arranjo físico e espacial de elementos de processo relacionados, como equipamentos, operadores e transportadores. Ele influencia e é influenciado pelo tipo de produto, mix, volume, etc.

e) Planejamento, programação e controle da produção

Constitui um importante componente do sistema de manufatura. O planejamento da produção é responsável pela informação da demanda de mercado, da capacidade produtiva e do nível de estoques para determinar os níveis de produção planejada para médio e longo prazo. Esse plano é desagregado para obter o agendamento das atividades de curto prazo. O controle irá fiscalizar se há divergências entre o planejado e o que está sendo executado e fará os ajustes necessários.

Já o fluxo de informações funciona como uma espécie de guarda-chuva que abrange todas as cinco funções, supervisionado as suas coordenações e a sincronização com os objetivos coorporativos.

A qualidade é uma parte do trabalho integrada em cada função. Ela determina, em comparação com dados de engenharia pré-definidos, se a peça está em conformidade, podendo ser aceita e continuar no processo produtivo, ser realocada em alguma operação específica para retrabalho ou mesmo então ser descartada.

### 2.2.2 Tipos de Manufatura

Os sistemas de manufatura possuem classificação em quatro áreas distintas: de acordo com a sua natureza, com a sua hierarquia, com a sua escala produtiva e com o seu arranjo físico do chão de fábrica.

Askin & Standridge (1993) classificam a manufatura em:

a) Peças discretas: caracterizadas como peças individuais que são claramente distinguíveis, ocorrendo na maioria dos produtos. Cada produto recebe atenção individualmente no chão de fábrica.
b) Processamento contínuo: os produtos apresentam-se num fluxo contínuo. Por exemplo, os produtos petroquímicos.

Ainda segundo Askin & Standridge (1993), os sistemas de manufatura podem ser naturalmente divididos hierarquicamente. O topo é composto por departamentos. Por sua vez, os departamentos contêm centros de trabalho, que são compostos por uma ou mais máquinas que são tratadas como entidade simples. No nível mais baixo, há a peça individual do equipamento, como ferramenta de máquina, controle ou robô. A figura 7 ilustra a referida organização hierárquica de um sistema de manufatura.

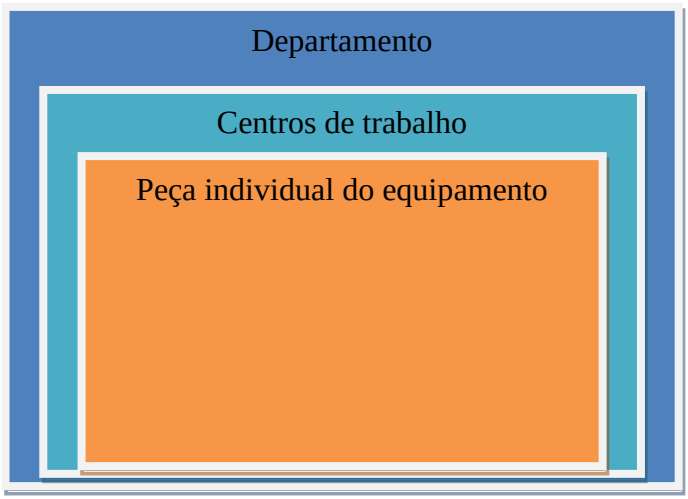


Figura 7 - Organização hierárquica de um sistema de manufatura

Para Danilevsky (1973), a manufatura pode ser por escala de produção e é divida em tipos que dependem de características do produto a ser produzido, tais como:

a) Peça, lote de trabalho ou produção

É caracterizado por pequena quantidade de peças produzidas e cada máquina pode realizar uma grande variedade de operações que não são periodicamente repetidas.

b) Produção em lote

É caracterizada por manufaturas de peças em lotes repetidos. O tamanho do lote é influenciado pelo tipo de peça e a quantidade de trabalho necessário.

c) Produção em Massa

É caracterizada principalmente pela produção de peças pré-definidas e com demandas estáveis. Ademais, tem como propriedades a produção de peças em larga escala e operações repetitivas com linha de produção.

d) Produção de Fluxo Contínuo

Possui como principal característica o tempo requerido para cada operação da linha de produção ser igual ou múltiplo do conjunto de tempo total de operações. A produção contínua geralmente se encaixa nessa categoria.

Por fim, há ainda a classificação dos sistemas de manufatura baseada no arranjo físico de fábrica (Askin & Standridge, 1993; Benjaafar *et al.*, 2002; Drira *et al.*, 2007):

a) Arranjo físico de posição fixa

Utilizado com produtos de dimensões elevadas como barcos, residências e aeronaves, uma vez que é impraticável a movimentação desses no chão de fábrica.

b) Arranjo físico de produto ou linha de produção

Os equipamentos de produção são alinhados de acordo com o processo de manufatura necessário para produzir o produto. Trata-se do tipo de arranjo físico mais eficiente e eficaz quando justificado por um pequeno mix e por um volume elevado de produtos.

c) Arranjo físico baseado em processo

Os departamentos são compostos por equipamentos com capacidades similares que executam funções similares. Em geral, exige-se que os operadores tenham um nível de habilidade mais elevada que o layout por produto, pois o mix de produção geralmente é elevado, sendo muitas vezes necessário o processamento de produtos diferentes no mesmo departamento.

d) Manufatura celular ou tecnologia de grupo

Pode ser usado para converter arranjo físico baseado em processo num arranjo físico de produto, no qual, no entanto, peças similares são agrupadas juntas em quantidade suficiente para justificar seus próprios equipamentos.

A similaridade das peças é baseada em famílias de produtos. Geralmente ela abrange formas parecidas, operações em comum, dentre outros.

e) Manufatura contratual

Esse tipo de arranjo físico permite que fornecedores externos possam ter acesso à linha de produção da empresa. Ele emprega uma estrutura parecida com uma espinha dorsal. Ademais, possibilita que exista uma alta eficácia na estrutura interna e alta eficiência nas empresas fornecedoras.

f) Diferenciação atrasada do produto

Nesse tipo de arranjo físico, a empresa atrasa ao máximo a diferenciação dos produtos. As empresas que fazem isso geralmente mantêm em estoque um número de peças-bases para que, quando a demanda existir, terminar o produto de acordo com o que fora solicitado.

g) Manufatura de multicanais

Nesse tipo de arranjo físico, a empresa emprega várias linhas de produção que podem manufaturar vários tipos de produtos, tornando o caminho bastante flexível e evitando que a

demanda fique desatendida. Além disso, duplicando os departamentos, aumenta-se a probabilidade de se encontrar um caminho ótimo para cada trabalho. Outro exemplo que pode ser considerado é o arranjo físico fractal, em que células são multiplicadas e trabalhos podem ser alocados dinamicamente a elas.

h) Máquinas portáveis

Esse sistema de manufatura é empregado quando há a facilidade de se modificar a posição dos equipamentos em diferentes áreas da fábrica.

i) Arranjo físico distribuído

No arranjo físico distribuído, desagregam-se grandes departamentos funcionais em subdepartamentos distribuídos pelo chão-de-fábrica. A duplicação desses subdepartamentos permite que sejam absorvidas mudanças de flutuações da demanda.

j) Arranjo físico ágil

Esse tipo de arranjo físico é projetado para maximizar o desempenho da operação, ao invés de minimizar o custo de tratamento de material.

k) Arranjo físico modular

Trata-se de um arranjo físico hibrido que combina outros tipos de arranjos físico, tais como o funcional, o linha ou o celular, sendo apropriado para produtos complexos. Algumas das novas tendências de produção industrial valem-se do arranjo físico modular. Os módulos são utilizados conforme a demanda necessite, podendo ser incluído ou excluído ao seu tempo. Assim a fabricação pode ser determinada de forma a aumentar ou a diminuir rapidamente.

Na tabela 1, as características e os quatro primeiros, e mais utilizados, tipos de arranjos físicos de fábrica são relacionados.

| Característica | Produto | Processo | Célula | Posição Fixa |
|---|---|---|---|---|
| Tempo de processamento | Baixo | Alto | Baixo | Médio |
| Trabalho em processo | Baixo | Alto | Baixo | Médio |
| Nível de habilidade | Dependente | Alto | Médio-alto | Variável |
| Flexibilidade do produto | Baixo | Alto | Médio-alto | Alto |
| Flexibilidade da demanda | Médio | Alto | Médio | Baixo |
| Utilização de equipamentos | Alto | Médio-baixo | Médio-alto | Médio |
| Utilização de operadores | Alto | Alto | Alto | Médio |
| Custo de produção unitário | Baixo | Alto | Baixo | Alto |

Tabela 1 Características gerais de tipos de layout

### 2.2.3 Sistema de Manufatura Flexível

Para De Toni & Tonchia (1998, p. 1591), a flexibilidade pode ser entendida como "a habilidade de mudar ou reagir, com pequenas perdas em esforços, custo ou desempenho". Para Porto (1990) é definido como a capacidade de um sistema em responder a perturbações, sem necessitar de alterações em sua estrutura funcional. O conceito de complexidade de sistema é relacionado com duas dimensões: incerteza e tempo.

O termo "flexibilidade" também possui como definição a coleção de estados possíveis de um sistema e o tempo necessário para mover-se entre um estado e outro, que afeta diretamente o custo envolvido na mudança.

Os sistemas de manufatura flexível (*flexible manufacturing system* – FMS), referem-se a um conjunto de máquinas de controle numérico computadorizadas (*computer numerically controlled* – CNC), as ferramentas e a estação de trabalho que são conectados por um manipulador automático de material e controlados por uma central de computadores (Askin & Standridge, 1993).

Além dos mesmos elementos encontrados nos sistemas de manufatura convencionais, os elementos-chaves que compõem um FMS são os que seguem:

a) Máquinas automáticas reprogramáveis;
b) Ferramenta de entrega e mudança automática;
c) Manipulador automático de material tanto para transferências de peças entre máquinas como para o carregamento/descarregamento de peças nas máquinas.
d) Controle coordenado.

Para Suresh Kumar & Sridharan (2009), um FMS pode simultaneamente processar volumes de tamanho médio de uma variedade de tipos de produto. Ele consegue aliar flexibilidade de produção com alta produtividade.

## 2.3 Simulação de Sistemas e Manufatura

Nessa seção serão abordados alguns trabalhos desenvolvidos recentemente e que utilizaram simulação de sistemas de manufaturas.

### 2.3.1 Modelagem e simulação de uma peça e controle de ferramenta em FMS

O trabalho de Suresh Kumar & Sridharan (2009) buscou solucionar problemas relacionados com o controle do fluxo de peças e ferramentas em uma operação de FMS em um ambiente de compartilhamento de ferramenta. Em ambientes como esse, a utilização de uma ferramenta não é definida no começo do planejamento, sendo utilizadas de acordo com a necessidade dos equipamentos.

O estudo de simulação do trabalho supracitado objetivava investigar os efeitos de regras de agendamento que controlam o lançamento de peças e a requisição de seleção de ferramentas de uma operação. Para tal, dois cenários foram definidos, onde o primeiro está livre de falhas, enquanto o segundo apresenta a ocorrência de falhas; o tempo médio entre falhas e o tempo para reparo seguem uma distribuição exponencial. As medidas de desempenho avaliadas são o tempo de fluxo, tempo de atraso médio, tempo médio de espera, tempo médio de espera para ferramenta e percentual de peças atrasadas. Ademais, pressupõe-

se que a chegada de produtos no FMS é aleatória, sendo executada continuamente. Cada chegada pode pertencer a qualquer peça dos vinte tipos encontrados. Os pressupostos feitos a respeito da operação são os seguintes:

a) A sequência de operações para cada tipo de peça é fixa;
b) Cada máquina pode operar no máximo uma operação por tempo;
c) O tempo de setup em cada operação está incluído no tempo de operação;
d) Uma cópia de cada tipo de ferramenta é disponibilizada;
e) AGV só transporta apenas uma peça por vez;
f) Transporte de ferramenta só transporta uma ferramenta por vez;
g) O tempo de transporte para AGV e transportador de ferramenta é proporcional à distância envolvida;
h) Depois de completar uma tarefa, o AGV pode continuar próximo à máquina ou à estação de carregamento/descarregamento, se for o caso;
i) Depois de completar qualquer transferência de material, AGV tem que atender às chamadas pendentes. As chamadas para o AGV podem ser de dois tipos: (i) proporcionando uma parte de entrada para uma máquina e (ii) removendo uma peça finalizada de um buffer de saída de uma máquina.

Para fins de programação da simulação, os autores utilizaram a linguagem de propósito geral C. O algoritmo está representado pelo fluxograma da figura 8.

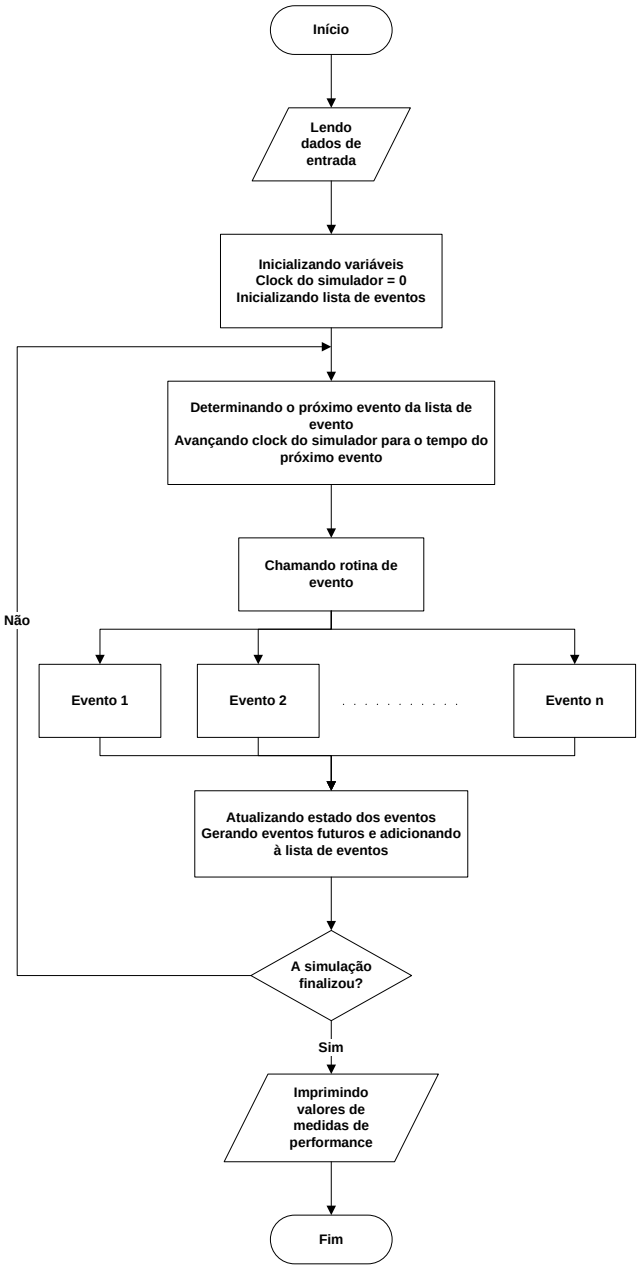


Figura 8 - Lógica global da simulação

O resultado das simulações em Suresh Kumar & Sridharan (2009) resultou em oitenta e quatro experimentos para cada cenário. Cada experimento de simulação foi replicado dez vezes. Por fim, verificou-se que, para o FMS em estudo, regras de agendamento para a decisão do lançamento de peças têm um efeito significativo no desempenho do sistema.

### 2.3.2 Modelo baseado em otimização de gerador de agendamento para um sistema de agendamento de produção genérico

Desenvolvido por Chien & Chen (2007) para o contexto de indústrias de alta tecnologia, como as de semicondutores. O trabalho foca nos problemas generalizados de agendamento de máquinas paralelas independentes. Essas apresentam as seguintes características: ambiente de máquinas paralelas independentes; entrada de tarefa dinâmica; não há uma preferência inicial; tempo de setup próprio dependente de sequência, requisitos de múltiplos recursos, restrição de procedência geral e recirculação de trabalhos. O artigo de Chien & Chen (2007) propõe um gerador de agendamento baseado em otimização (*optimization-based schedule generator* – OptSG) que é composto por algoritmo genérico, uma técnica de busca heurística que pode obter soluções ótimas ou quase ótimas - e as redes de petri coloridas temporizadas, essas sendo adotadas como uma representação conceitual dos procedimentos decodificados para análise e modelagem de sistemas. Ademais, eles desenvolveram um modelo de programação linear inteiro misturado como *benchmark* para estimar a validação do OptSG. A abordagem que os autores utilizaram para realizar a simulação foi a abordagem de três fases.

Uma vantagem inicialmente levantada por Chien & Chen (2007) é que a utilização do OptSG separa o problema estrutural do problema de configuração, decorrente do uso das redes de petri coloridas.

A arquitetura geral do OptSG é apresentada na figura 9.

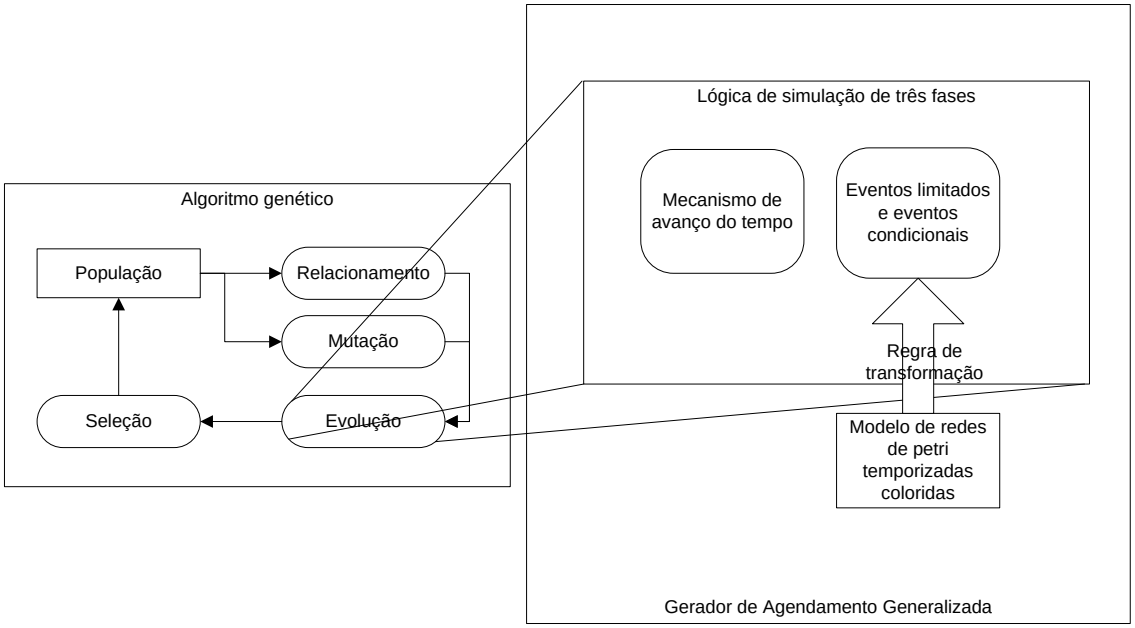


Figura 9 - Gerador de agendamento otimizado (Chien & Chen, 2007, p. 1767)

Na figura 9, primeiro o modelo de redes de petri coloridas representa a análise do comportamento dinâmico do sistema de eventos discretos. Ele é subsequentemente transformado em lógica de simulação da abordagem de três fases para construir o gerador de agendamento generalizado. Por fim, um algoritmo genérico é incorporado na estratégia de agendamento que é provido para determinar a combinação de regras de envio.

Experimentos foram conduzidos para avaliar o desempenho de agendamento do OptSG, considerando dois conjuntos de problemas de diferentes configurações de tamanhos. O primeiro conjunto de problemas foi de tamanho menor, de sorte que a solução ótima pôde ser obtida em um tempo computacional aceitável. Já o segundo conjunto de problemas estava mais próximo das configurações reais, apresentado, um nível de complexidade mais elevado. Daí que, em decorrência do crescimento exponencial da complexidade computacional, fez-se indisponível encontrar uma solução ótima.

Chien & Chen (2007) informam que o modelo baseado em programação linear inteiro misturado pode não ser resolvido como um problema prático em um tempo razoável, o tempo inseparável de setup dependente da sequência e o requerimento de múltiplos recursos foram direcionados simultaneamente no modelo. Assim, a solução ótima do modelo de programação linear é mais realístico e próximo de situações práticas, que aumenta a veracidade na utilização destes modelos na validação da solução do OptSG. Por fim, Chien & Chen (2007)

concluem que, com a experimentação, o OptSG proposto pode realizar agendamentos de bom desempenho, superando a regra heurística de envio e fornecendo uma solução ótima em muitos casos.

## 2.4 Realidade Virtual

As primeiras pesquisas em realidade virtual apareceram no artigo de Sutherland (1965), em que ele observou que um dispositivo de visualização (display), conectado a um computador digital familiariza o usuário com um mundo de fatos virtuais. Ele estudou a imersão e a simulação computacional nesse ambiente virtual.

O termo RV (Realidade Virtual) foi utilizado pela primeira vez no início dos anos 80, por Jaron Lanier, fundador da VPL Researches Inc (Araujo, 1996). Para Kirner & Siscoutto (2007), a realidade virtual é uma "interface avançada do usuário", que lhe permite acessar aplicações tridimensionais geradas pelo computador.

Todo sistema de RV descreve uma tecnologia computacional que permite ao usuário enxergar, interagir e navegar, por intermédio de um dispositivo especial, ao invés do mundo real, um mundo virtual gerado por computador (Vince, 1998). Schuemie *et al* (2001) citam que os fatores que diferenciam RV de outras tecnologias com interface visual é a presença, ou imersão do usuário, e a interação com os elementos virtuais.

Interação é um aspecto crucial da RV, apresentando os seguintes aspectos: a navegação no mundo virtual e a dinâmica do ambiente (Pimentel & Teixeira, 1993). Os dispositivos de interação da RV extrapolam os convencionais encontrados num computador pessoal.

Vince (1998) e Pimentel & Teixeira (1993), citam que os termos "ambiente virtual" e "ambiente sintético" apareceram durante a década de 1990 aonde eles emulam um ambiente que apresenta substitutos para objetos ou ambientes reais. Ainda segundo os autores, a RV pode ser dividida em imersiva e não-imersiva.

A realidade virtual imersiva é aquela em que o usuário não percebe os estímulos externos ao ambiente de RV, pois os seus principais sentidos, como a visão e a audição, são

incitados pelos dispositivos de saída. Já a RV não-imersiva utiliza apenas parte dos sentidos do usuário, possibilitando assim que o ambiente externo influencie em suas percepções.

Outro tipo de classificação que pode ser feita à RV é em relação ao método de percepção visual pelo usuário. Essa por sua vez é subdividida em monoscópica e estereoscópica.

A visualização monoscópica é aquela em que não existe uma diferenciação no que os olhos do observador enxergam, pois ambos recebem a mesma perspectiva da imagem. Já a visualização estereoscópica gera imagens distintas para cada olho do observador com perspectivas distintas. Há ainda a estereoscopia ativa, que utiliza apenas uma fonte de emissão de imagens, sendo sincronizada com os óculos que permitem a passagem alternada da perspectiva da imagem para cada olho e a estereoscopia passiva, que utiliza dois dispositivos geradores de duas imagens simultaneamente, de sorte que suas perspectivas sejam sobrepostas, e um par de lentes polarizadas utilizadas pelo usuário, que permite a cada lente visualizar a imagem do seu gerador específico.

### 2.4.1 Dispositivos de Realidade Virtual

Os dispositivos físicos de Realidade Virtual são divididos basicamente em dispositivos de entrada e dispositivos de saída (Vince, 1998).

Os dispositivos de entrada são os dispositivos que a RV utiliza para captar informações. Os principais dispositivos de entrada são:

a) Rastreadores eletrônicos, ópticos, ultrasônicos, eletromagnetismos e inerciais - utilizados para fornecer o posicionamento espacial do usuário no ambiente RV. A figura 10 ilustra um rastreador.

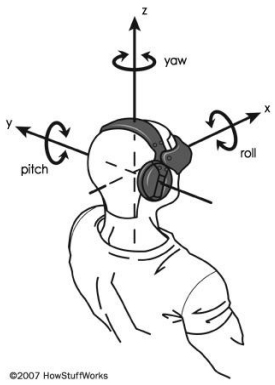


Figura 10 – Rastreador

b) Mouse 3D - tem um funcionamento parecido com o mouse convencional, porém permite o controle de mais eixos. A figura 11 ilustra um Mouse 3D.

Figura 11 - Mouse 3D

c) Luvas - dependendo do modelo, elas podem detectar a posição das mãos do usuário, o movimento dos dedos e até possuir resposta mecânica à estímulos virtuais. A figura 12 apresenta um par de luvas.

Figura 12 - Luva

Os dispositivos de saída são os dispositivos que mostram, não apenas visualmente, os dados que foram tratados pela RV. Os principais dispositivos de saída são:

a) Sensores de *Force Feedback*. – retornam com pressão em pontos definidos ao usuário.

b) Óculos – permitem ao usuário enxergar imagens estereoscópicas. A figura 13 ilustra um desses óculos.

Figura 13 - Óculos de visão 3D estéreo ativo

c) Monitor CRT – dispositivo de visualização padrão encontrado em computadores. Também pode ser de LCD ou plasma. Na figura 14, um monitor CRT é apresentado.

Figura 14 - Monitor CRT

d) *Head Mounted Display* (HMD) - dispositivo que une visualização 3D ativa com rastreamento. A figura 15 ilustra um exemplo do HMD.

Figura 15 - Head Mounted Display

e) *Binocular Omni-Orientation Monitor* (BOOM) – dispositivo semelhante com o HMD, mas que não permite que o usuário tenha livre movimentação, uma vez que seu posicionamento é fixo. A figura 16 ilustra um exemplo do BOOM.

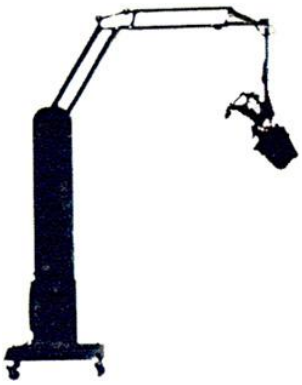

Figura 16 - Binocular Omni-Orientation Monitor

f) Tela em panorâmica – telas de projeção em que a tela não possui quinas ou cantos. Na figura 17, uma tela em panorâmica é ilustrada.


Figura 17 - Tela panorâmica

g) Mesa virtual (*Virtual Table*) – dispositivo de visualização em que a projeção é feita numa superfície geralmente na horizontal e levemente inclinada e que permite a interação do usuário com a RV através de tela sensível ao toque. A figura 18 apresenta um exemplo de uma mesa virtual.

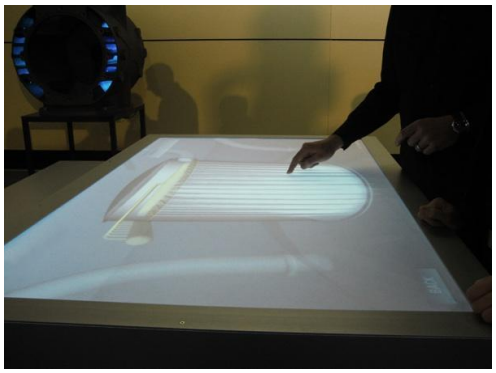

Figura 18 - Mesa Virtual

h) CAVE – esse dispositivo será detalhado mais adiante.

Segundo Cruz-Neira *et al* (1992), a pesquisa moderna em realidade virtual segue quatro caminhos diferentes, mas todos baseados nos dispositivos de visualização: o tubo de raio catódico (ou *Cathode Ray Tube* – CRT), o display acoplado a cabeça (ou *Head-Mounted Display* – HMD), o monitor omni-orientado binocular (ou 'Binocular Omni-Oriented Monitor – BOOM) e a CAVE. As características de cada dispositivo são divididas em duas categorias: campos de imersão e campos de visualização.

Os campos de imersão são divididos em: o campo de visão, que representa o ângulo visual que pode ser visto sem realizar a rotação da cabeça; o panorama, que é a habilidade de um dispositivo visual criar um ambiente imersivo; a perspectiva, que depende da velocidade e da acuidade da sensibilidade do visualizador; a representação física do corpo, que corresponde à capacidade do dispositivo visual representar o usuário no ambiente de realidade virtual e, por fim, a intrusão, que indica a restrição dos sentidos na realidade virtual. Os valores de cada categoria obtidos por cada dispositivo de visualização são expressos a seguir na tabela 2:

| | Campo de Visão | Panorama | Perspectiva | Representação Física de Corpo | Intrusão |
|---|---|---|---|---|---|
| CRT | 45° | Não | Lento | Físico | Não |
| BOOM | 90° ↔ 120 | Rápido | Rápido | Virtual | Parcial |
| HMD | 100° ↔ 140 | Lento | Lento | Virtual | Total |
| CAVE | Total | Rápido | Lento | Físico | Não |

Tabela 2 – Campos de Imersão

Por sua vez, os campos de visualização são divididos em: a acuidade visual, que é a qualidade da interface da realidade virtual determinada pela combinação da resolução e do campo de visão; a linearidade, que é a relação da resolução encontrada na área central e periférica do dispositivo visual; a visualização espacial, que é a habilidade de movimentação sobre um objeto, observando sobre vários ângulos; o refinamento progressivo, que é a habilidade de incremento dinâmico durante uma pausa na interação com o usuário, tendo como padrão de simulação um modelo grosseiro para visualizações rápidas; e, por fim, a colaboração, que é a permissão para mais de um usuário utilizar o dispositivo visual para o envolvimento em realidade virtual. A tabela 3 apresenta um comparativo entre os diversos dispositivos acerca de tais indicadores.

| | Acuidade Visual | Linearidade | Visualização Espacial | Refinamento Progressivo | Colaboração |
|---|---|---|---|---|---|
| CRT | 20/45 | Linear | Limitada | Apenas Local Fixo | Simples Perspetiva |
| BOOM | 20/85* | LEEP | Total | Apenas Local Fixo e Rotação | Duplo Hardware |
| HMD | 20/425 | Linear / LEEP | Total | Apenas Local Fixo e Rotação | Duplo Hardware |
| CAVE | 20/110 | Linear | Total | Apenas Local Fixo | Simples Perspetiva |

Tabela 3 - Campos de Visualização

Em outro artigo, Cruz-Neira *et al.* (1993) diferenciam realidade virtual de outros desenvolvimentos gráficos padrões realizados em computador com as características seguintes:

1- Oclusão;
2- Projeção de perspectiva;
3- Disparidade binocular (projetores estéreos);
4- Movimentos de paralaxe (movimento da cabeça);
5- Convergência (campo de busca óptica);
6- Alojamento (foco visual, como um reflexo de mono-lente no campo de busca);
7- Atmosfera (ambiente);
8- Luzes e sombras.

Os desenvolvimentos gráficos padrões oferecem as características 1,2, 7 e 8, enquanto a realidade virtual acrescenta 3,4 e 5.

### 2.4.2 CAVE

A CAVE (*Cave Automatic Virtual Environment*) foi desenvolvida para ser uma ferramenta para visualizações científicas, ainda segundo Cruz-Neira *et al.* (1993). Outra finalidade seria o teatro de realidade virtual, onde cientistas projetariam o critério de exposição.

A primeira CAVE foi desenvolvida no *Electronic Visualization Laboratory* na University of Illinois em Chicago e é apresentada na SIGGRAPH (*Special Interest Group on GRAPHics and Interactive Techniques*) do mesmo ano (Cruz-Neira *et al.*, 1992).

A CAVE é construída a partir de um número de telas projeções com projetores externos projetando suas imagens. Em seu interior, o usuário é imerso e utiliza óculos para a visualização de imagens estereoscópicas. A figura 19 apresenta um exemplo de CAVE.

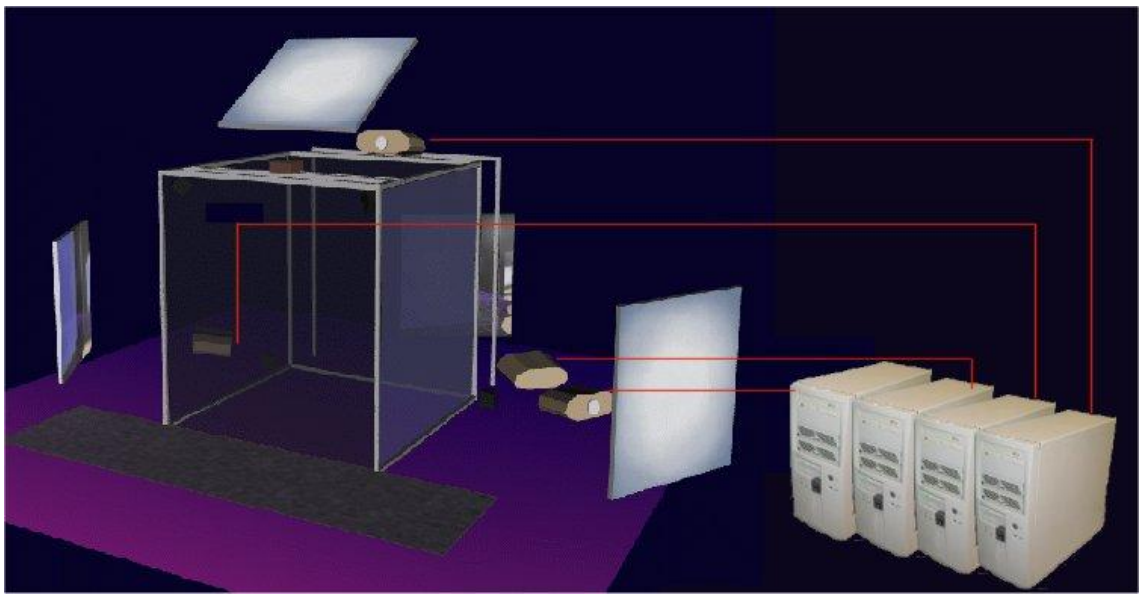

Figura 19 - CAVE

Na figura 19, está detalhado um exemplo de CAVE de quatro lados, em que cada projetor é ligado num PC, formando um cluster. Pode-se perceber a utilização dos espelhos, que são utilizados quando não há espaço suficiente para colocar os projetores a uma distância adequada.

Os objetivos pretendidos com o desenvolvimento da primeira CAVE incluíam:

1- O desejo por imagens coloridas de alta resolução e uma visão periférica sem distorções geométricas;
2- Menos sensibilidade na rotação da cabeça, induzindo a erros;
3- A habilidade de unir Realidade Virtual com dispositivos físicos reais ;
4- A necessidade de guiar e ensinar em modos e mundos artificiais;
5- O desejo de unir supercomputadores em rede e fontes de dados para sucessíveis refinamentos.

Cruz-Neira *et al* (1993) também afirmam que um fato raro de ser notado é que, em computação gráfica, o plano de projeção pode ser posicionado de várias maneiras, não necessariamente perpendicular ao observador. Um exemplo de um plano de projeção não convencional é o *hemisphere*, outro dispositivo de realidade virtual em que as telas não são totalmente perpendiculares umas as outras, podendo inclusive serem curvas.

Defanti *et al* (2009) relatam que, nos últimos 17 anos desde o surgimento da primeira CAVE, centenas outras, contando com as suas variações, apareceram nos mais diversos países. Ainda segundo os referidos autores, a CAVE atualmente está na sua quarta geração. A primeira geração teve como características o uso de estereoscopia ativa e três tubos de CRT, que foram usados em cada um dos projetores de 3 m² com resolução de 1280 px x 1024 px com uma frequência de 120 Hz. O efeito era parecido com a visualização de cores com o brilho da lua.

A segunda geração da CAVE foi desenvolvida em 2001 com projetores de resolução de 1280 px x 1024 px que possuíam até sete vezes mais brilho e custavam cinco vezes mais. Este sistema também foi desenvolvido utilizando a estereoscopia ativa.

A terceira geração, conhecida por StarCAVE, foi desenvolvida em 2007, a projeção inferior ao chão foi adicionada. Utiliza a estereoscopia passiva para efeito de economia de bateria. Os projetores laterais, perpendiculares ao usuário, são compostos por três telas que

possuem uma leve inclinação entre eles, aumentando o poder de visualização do usuário sem o mesmo fazer muito esforços. Em resumo, as adições de projeto foram:

a) Imersão completamente rastreável para um usuário e demais observadores;

b) Aumento no brilho, alta-resolução e alto contraste;

c) Experiência ambiental realista de auditório imersivo, escala para cenas virtuais arbitrárias dinâmicas, para um ou mais usuário.

d) Simples aquisição, operação, manutenção e possíveis atualizações com projetores e computadores úteis.

A quarta geração chamada de Varrier™, ainda está em desenvolvimento e utiliza uma resolução de 40 megapixel para cada olho, sendo feita com mais de 60 painéis com resolução de 1600 px x 1200 px (que não são projetores).

## 2.5 Simulação de sistemas, manufatura e realidade virtual

De acordo com Kelsick *et al* (2003), a realidade virtual tem sido amplamente usada para auxiliar na modelagem de sistemas de manufatura. Basicamente, esse auxílio pode ser dividido em quatro áreas: simulação de processo, arranjo físico de fábrica, prototipagem de métodos de montagem e simulação de fluxo de peça. Como o objetivo deste trabalho não consiste em trabalhar diretamente com a teoria do arranjo físico de chão-de-fábrica nem prototipagem de métodos de montagem, nesta seção, apenas trabalhos que envolvem simulação serão abordados.

### 2.5.1 Realidade virtual para sistemas de manufatura

Desenvolvido por Jones *et al.* (1993), que procurou estudar os impactos do uso da realidade virtual como uma nova dimensão na animação de se simular um sistema de manufatura. Para os autores, a animação em simulação é uma forma de comunicar,

amigavelmente, resultados às pessoas que não possuam o conhecimento em simulação necessário, mas que estejam envolvidas no sistema real. A realidade virtual, com a visualização tridimensional, dá um passo adiante no uso de animação em simulação. Para a geração da realidade virtual, foi utilizado um HMD, um *tracker* e um joystick que ajuda na navegação do ambiente virtual.

Para a realização do trabalho de Jones *et al.* (1993), pacotes de softwares comerciais de simulação AutoMod e AutoSIM foram usados. Da parte da realidade virtual, foram utilizados os softwares VEOS (*Virtual Environment Operating System*) e uma plataforma independente de biblioteca de renderização gráfica, a *Imager Library*. O principal objetivo foi prover ao usuário, dentro do ambiente virtual, a possibilidade de construir uma fábrica virtual e, posteriormente, alterar os parâmetros do funcionamento, tornando possível a prática do "e-se" presente nos simuladores.

Porém, devido à estrutura do software AutoMod, as alterações não foram possíveis. Então substituiu-se o AutoMod pelo AutoView, para permitir ao usuário experimentar uma visualização estática tridimensional e tendo como únicas interações a opção de iniciar ou parar a simulação.

A partir do AutoView, foi criado um pacote modificado chamado AMVE (*AutoMod Virtual Environment*), que é basicamente as mesmas funcionalidades do originário com a adição de duas funções, escritas em linguagem c: av-init-model, e carregada apenas uma vez, lê os scripts do AutoView, gera as geometrias iniciais e os atributos de todos os objetos, e a função av-step-animation, coração do laço da animação. Na figura 20, o fluxograma com as duas funções está representado.

As maiores dificuldades encontradas por Jones *et al.* (1993) foram a baixa velocidade da animação, medidas pelo número de quadro por segundo, que no experimento ficou em três a cinco por segundo, que é bem inferior ao ideal, que é de mais de vinte e cinco quadros por segundo. A segunda é que a ferramenta VEOS não é otimizada para transmissão de dados ou apropriado banco de dados de padrões. Por último, a rede utilizada era compartilhada por todos os computadores do setor, muitos não pertencente à esse projeto, tornando-a lenta na troca de informações.

A forma encontrada para a metodologia de avaliação foi a de resposta qualitativa e as discussões dos usuários que o utilizaram para comparar o protótipo com o produto existente AutoView.

Foi observado que quem utilizou o AMVE teve uma participação mais ativa na simulação que os que utilizaram apenas o AutoView. E que a fábrica virtual capturou a imaginação dos participantes.

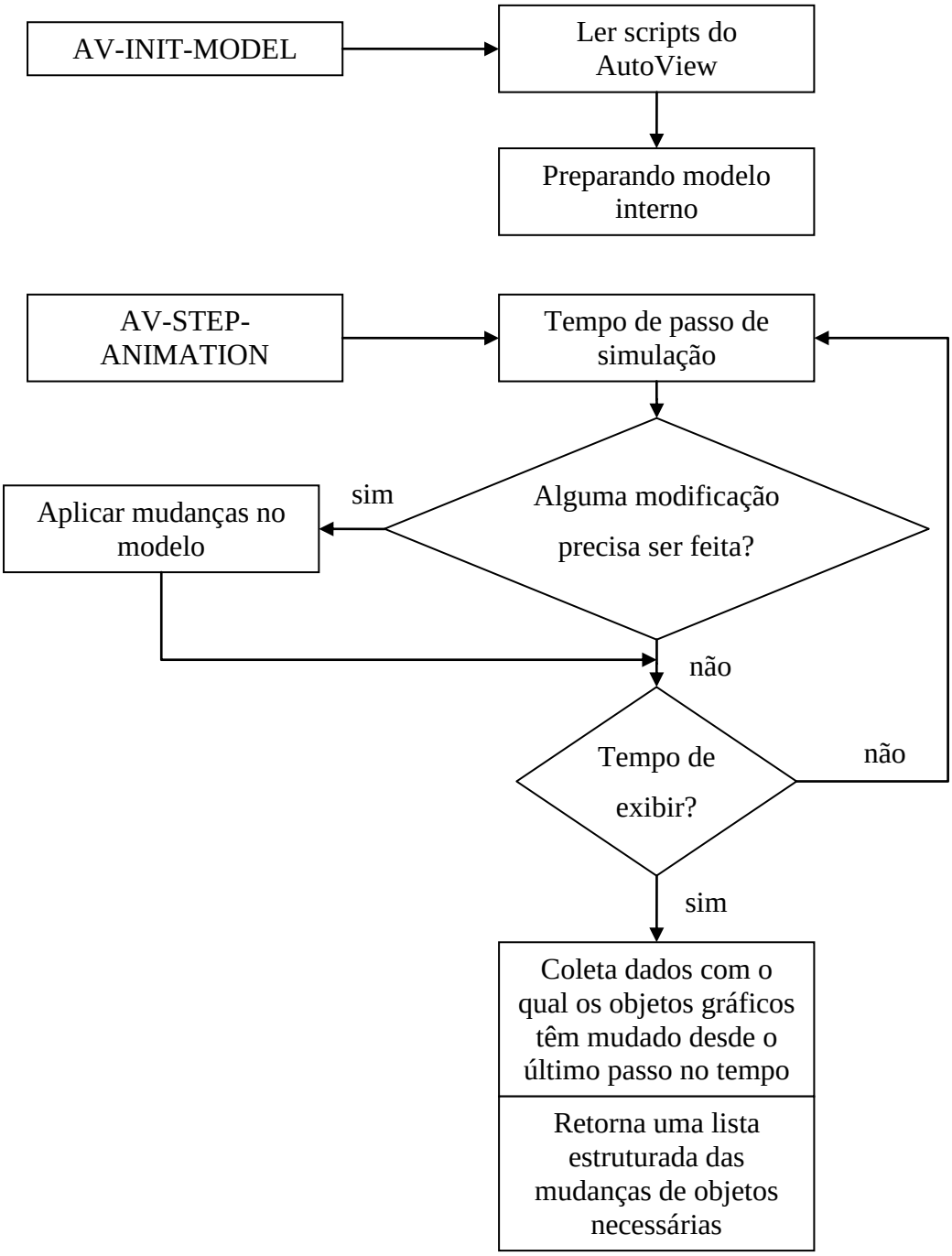


Figura 20 - Operação conceitual das duas chamadas da AMVE(Jones *et al.*, 1993, p. 884)

### 2.5.2 Construção de um ambiente virtual para simulação de manufatura

Trabalho de pesquisa que buscou projetar e desenvolver um sistema aberto de ambiente virtual para sistemas de manufatura (Xu *et al.*, 2000). Ele tinha como objetivos: (1) Investigar a estrutura de dados de modelagem dos ambientes virtuais e a relação interna entre os dados modelados e os dados de manufatura; (2) Um método para definição de dados de ligação entre o ambiente virtual e processos físicos de manufatura; (3) Desenvolver um sistema de banco de dados que gerencie automaticamente os dados do ambiente virtual e os dados de manufatura, removendo dados complicados ao usuário e fornecendo uma interface fácil de usar para usuários construírem e configurar um ambiente virtual específico.

A implementação da metodologia resultou em um sistema computacional intitulado *Knowledge Acquisition and Management using Virtual Reality*[1] (KAMVR). O sistema possui quatro camadas, com cada camada possuindo seus módulos. A primeira camada é uma interface do usuário com a realidade virtual, podendo o usuário ser capaz de realizar criação, alteração e eliminação, dentre outros.

A segunda camada, o gerenciador do sistema, é o manipulador de conhecimento e gerenciador de dados. A terceira é dedicada ao banco de dados do ambiente virtual de manufatura e um banco de dados de arquivos com domínio específico. A quarta, e última camada contém o sistema de manufatura real baseado no planejamento de processo e controle de workshop.

Por fim, os autores concluem que a simulação de sistemas de manufatura utilizando tecnologia de realidade virtual é válida em ambientes virtuais grandes e complexos.

[1] Aquisição e gerenciamento de conhecimento utilizando realidade virtual, tradução livre.

### 2.5.3 Implementação de simulação de eventos em um ambiente virtual

Trabalho realizado por Kelsick *et al* (2003) que descreve um aplicativo de realidade virtual aonde os resultados de um simulador de eventos discretos de uma célula de manufatura, ou *VRFactory*, são integrados em um modelo virtual da célula para produzir um ambiente virtual. A célula de manufatura virtual é um modelo animado, imersivo e animado.

Foi escolhido ser desenvolvido uma interface que utilize uma linguagem de simulação comercial, a AweSim, já existente ao invés de ser desenvolvida uma nova, pois, segundo os autores, o software de simulação e o ambiente virtual podem ser otimizados com os respectivos potenciais máximos.

Para o auxilio da interação do usuário no ambiente virtual, foram utilizados os periféricos luva e rastreador de posição do usuário. Porém, podem ser simulados em um ambiente *desktop* com o uso de mouse e teclado. Um menu é disponibilizado para que o usuário possa modificar parâmetros acerca dos elementos do ambiente.

A geometria dos elementos foi desenvolvida utilizando softwares CAD e modeladores. Após serem criadas, as geometrias dos elementos são transladadas definitivamente para suas posições no chão-de-fábrica ou para posições relativas à outros objetos, se for necessário. A composição de cada elemento é feita por objetos que podem se movimentar relativamente ao resto, fornecendo uma noção de realismo, quando estiver simulando o seu funcionamento.

A célula de usinagem flexível é composta de quatro centros de usinagem Makino, um centro de usinagem horizontal Enshu, um veículo guiado por trilhos, uma esteira, uma máquina de rebarba Cascade, e um serviço automático e sistema de reabastecimento. A estrutura do programa VRFactory permite modificações adicionais e possível acomodação de máquinas de rebarba, limpeza e afiação no futuro.

A variável independente mais importante para a simulação do sistema de manufatura é o tempo. Cada estágio ou movimento de uma peça é definido por um tempo específico de início no arquivo de eventos discretos. Depois de toda a simulação tiver sido completada, variáveis contadores e de rastro foram usados para gravar a utilização de recursos e a movimentação e localização das entidades. O grande gargalo ficou no limitado controle que é encontrado nos sistemas de manufatura reais.

# 3. DESENVOLVIMENTO DE UM NÚCLEO DE SIMULADOR DE EVENTOS DISCRETOS PARA SISTEMAS DE MANUFATURA COM VISUALIZAÇÃO 3D

Neste capítulo as etapas para o desenvolvimento do núcleo de simulador de eventos discretos para sistemas de manufatura com visualização 3D serão apresentadas. Na primeira etapa, as principais ferramentas utilizadas serão abordadas. A etapa seguinte discorrerá sobre as definições das áreas. A modelagem computacional será feita na terceira etapa. Na última etapa, o protótipo do núcleo desenvolvido será apresentado.

## 3.1 Ferramentas

Nesta seção, as ferramentas computacionais que foram utilizadas para a construção do núcleo do simulador serão descritas.

### 3.1.1 Linguagem de modelagem unificada

A linguagem de modelagem unificada (*Unified Modeling Language* – UML) é um conjunto de padrões de modelagem orientados a objetos. Ela é uma linguagem utilizada para expressar projetos de modelagem e, por não ter processos, não pode ser considerada um método (France *et al.*, 1998; Fowler & Scott, 2000; Larman, 2007).

A UML se encontra, em nível de detalhamento, em uma situação intermediária entre a linguagem humana, pois é mais precisa, e o código de programação, que é menos detalhada. Ela também é genérica, devido à sua independência de qualquer processo.

Há duas fases distintas na UML. Na primeira, durante a análise orientada a objetos, há a ênfase em encontrar e descrever os objetos que irão compor o sistema. Já na segunda, durante o projeto orientado a objetos, dá-se destaque na definição dos objetos de software e como eles colaboram para a satisfação dos requisitos (France *et al.*, 1998).

A UML é composta por vários diagramas. Porém, como o escopo do presente trabalho não é o estudo aprofundado da linguagem, um destaque maior nos diagramas de classe e de sequência será feito, que se encontram na primeira e na segunda fase, respectivamente, e que foram utilizados para desenvolver o núcleo do simulador de eventos discretos para sistemas de manufatura com visualização 3D.

*Diagrama de classe*

Os diagramas de classes são utilizados para a modelagem estática de um conjunto de classes que possuem relações entre si. Elas possuem definição semelhante(quase idêntica) à utilizada na seção 2.1, porém, ao invés de entidades, a UML irá agrupar objetos similares. Cada classe terá a composição fornecida pela figura 21.

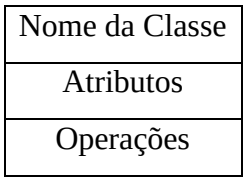


Figura 21 - Definição de classe

Na figura 21, os atributos seguem a mesma definição do tópico 2.1, enquanto que operação é a declaração dos comportamentos dinâmicos da classe. Além disso, um método é a implementação de uma operação dentro de um objeto individualizado.

As relações entre classes do diagrama de classe podem ser do tipo: associação, que representa a atuação de uma classe em relação à outra; agregação, onde uma classe é composta por instâncias(s) de outra(s); composição, idem à anterior, porém com mais consistência, pois a(s) classe(s) que compõe(m) a outra só existe(m) mediante a existência da que é composta; por fim, herança representa o relacionamento em que as classes filhas herdam todas as características da classe mãe, e novas informações podem ser adicionadas. A figura 22 exibe um exemplo de diagrama de classe.

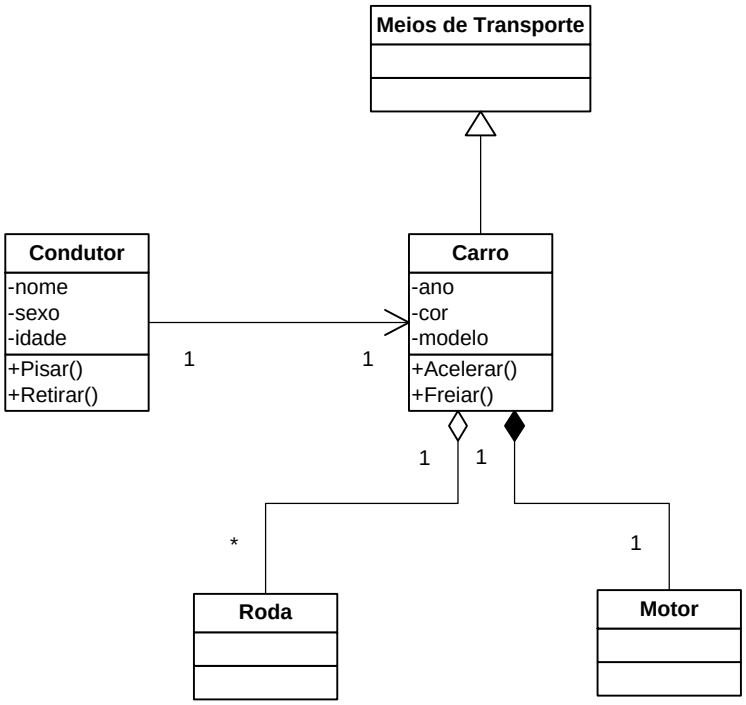


Figura 22 - Exemplo de diagrama de classe

No exemplo da figura 22 a classe Carro demonstra como funciona cada relacionamento. Na primeira relação, cada objeto da classe Condutor se associa com um objeto da classe Carro, como pode ser visto na numeração em cada extremidade (1 para 1), ou seja, cada condutor dirige apenas um carro, ou um carro é conduzido por apenas um condutor, já que a seta está no sentido Condutor para Carro. Outro relacionamento é entre a classe Carro e a classe Meios de Transporte, sendo do tipo herança, pois todo carro é do tipo meio de transporte. As duas últimas relações, Agregação e Composição, acontecem entre a classe Carro e as classes Roda e Motor, respectivamente. Na primeira, um Carro tem, ou pode ter, mais de uma Roda (o símbolo "*" no final da associação), e na segunda, ele possui um Motor. A diferença entre elas é que a eliminação do Carro, como classe, não fará o termino da classe Roda, mas fará o termino da classe Motor.

*Diagrama de sequência*

O diagrama de sequência é utilizado para modelar o funcionamento de cada cenário, ou processo de execução básico, de um sistema. Esse diagrama tem como características: um objeto é representado como uma caixa na parte superior; a linha vertical abaixo do objeto é chamada de linha de vida dele; e uma mensagem é uma flecha entre as linhas de vida de dois objetos. Existem outras características do diagrama de sequência, porém não listadas aqui, já que essas são suficientes para um entendimento básico do núcleo do simulador.

A remoção de um objeto por outro é representado com um X na linha de vida desse, simulando seu fim. A figura 23 apresenta um exemplo de um diagrama de sequência.

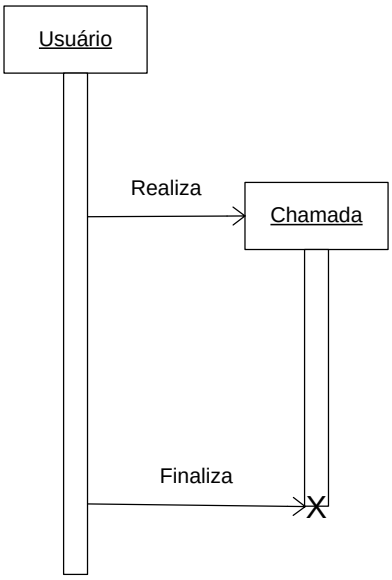


Figura 23 - Exemplo de um diagrama de sequência

No exemplo da figura 23, um usuário realiza uma simples chamada, por um período de tempo e depois ela é desligada. Nesse exemplo, o comprimento vertical da barra que fica

abaixo do nome do objeto Chamada representa graficamente a quantidade de tempo da ligação, pois, após seu fim, o objeto é destruído.

### 3.1.2 VR Juggler

VR Juggler é um software orientado a objetos, expansível, de gerenciamento de ambientes virtuais e que suporta desenvolvimento de aplicativos interativos e imersivos. Ele utiliza plataforma virtual (PV), que possui como característica principal permitir ao desenvolvedor focar basicamente nos seus aplicativos (Bierbaum *et al.*, 2001).

Para que um software utilize PV, ele deve possuir, além da supracitada, as seguintes características: (1) Abstrair a complexidade de ambientes virtuais usando componentes locais e distribuídos; (2) Fornecer possibilidade de crescimento em relação a esses dispositivos; (3) Flexibilizar a configuração do hardware; (4) Permitir que a configuração de software e hardware seja dinâmica; (5) Permitir a execução de múltiplos aplicativos ao mesmo tempo; (6) Fornecer ferramentas para melhorar o desempenho das camadas que compõem a aplicação.

O VR Juggler, sendo composto por PV, permite ainda a utilização das seguintes características:

a) Independência do sistema operacional

O aplicativo projetado para funcionar no ambiente VR Juggler é portável, podendo ser utilizado em vários sistemas operacionais, pois a PV do VR Juggler faz uma camada de abstração dos componentes de hardware e software entre o aplicativo e o sistema operacional da máquina.

b) Ambiente de operações

Cada aplicativo que o projetista desenvolver para o VR Juggler irá se comportar como objeto, controlado pelo núcleo do VR Juggler. Essa característica permite que vários aplicativos possam ser executados na mesma instância do VR Juggler.

c) Suporte à *Application Programming Interface* (API) gráfica multimídia

As APIs gráficas auxiliam o usuário no desenvolvimento de aplicativos para o ambiente virtual, pois fornecem um conjunto de rotinas que facilitam a manipulação de

elementos virtuais. Há vários desses modelos de APIs no VR Juggler que podem ser escolhidos. Ademais, a PV do VR Juggler encapsula todos os seus comportamentos no gerenciador de desenho.

Para o presente trabalho, a API escolhida foi a OpenSG, que será estudada na Seção 3.1.3.

A camada da PV do VR Juggler é composta basicamente por uma interface do núcleo, que coordena todas as rotinas básicas do seu funcionamento, e do gerenciador de desenho. Ele funciona como mostrado na figura 24.

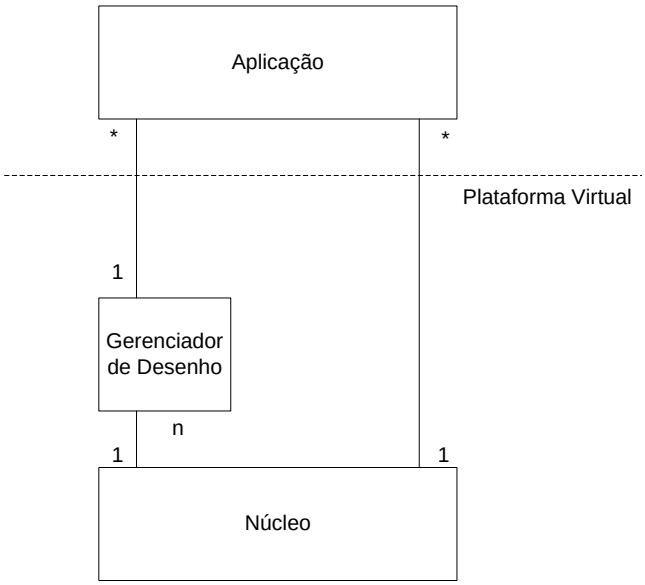


Figura 24 - Aplicação e Interface PV (Bierbaum *et al.*, 2001)

Na figura 24, é possível observar que o núcleo controla a execução de aplicativos e o gerenciamento da comunicação com os dispositivos de entrada e de saída da realidade virtual. Ele funciona como um mediador que auxilia na comunicação do aplicativo com o ambiente.

As informações que o núcleo do VR Juggler utiliza para se comunicar com os hardwares do ambiente e suas configurações são armazenadas em um arquivo baseado em XML. Ele é criado e modificado em um aplicativo externo baseado em Java que permite, inclusive, realizar modificações em tempo de execução.

### 3.1.3 OpenSG

OpenSG é uma API gráfica portável utilizada no desenvolvimento de aplicativos em ambientes de RV. Ela utiliza internamente outra API gráfica, a OpenGL. A diferença é que essa trata as propriedades dos elementos da cena individualmente, e aquela as manipula em conjunto, tornando possível o controle mais universal (Opensg, 2009).

A OpenSG possui diversas características, como *multithread*, *FieldContainers* e grafo de cena. Dessas, será dado ênfase à última, já que as duas primeiras não serão explicitamente utilizadas.

O grafo de cena é uma estrutura de nós interconectados que são percorridos todas as vezes que a cena é gerada para visualização. Essa estrutura é hierárquica, logo, cada operação realizada no nó pai é transferida para os nós descendentes (Opensg, 2009). Das operações que podem ser realizadas, a operação de transformação modifica o estado do objeto na cena, em relação a um eixo, ou a combinação de mais de um. Dela, há três tipos: a transformação de translação é um deslocamento linear da posição base para uma nova posição; a transformação de rotação é um deslocamento angular sobre um eixo; e a transformação de escala é uma operação que modifica a proporção das dimensões do objeto. Essa operação será a única a ser utilizada no presente trabalho e mais especificamente os dois primeiros tipos. A figura 25 apresenta um exemplo de grafo de cena.

No exemplo da figura, o grafo de cena foi utilizado para demonstrar o funcionamento básico de um carro em um ambiente de RV. O nó hierarquicamente mais superior está acima do nó carro e qualquer transformação aplicada nele repercutirá em todos os nós descendentes. Esse nó representa as transformações aplicadas ao nó carro que, ao se locomover, fará o mesmo a todas as rodas. Ademais, cada nó de transformação ascendente a cada roda só afetará ela mesma, ou seja, quando uma delas for rotacionada, o que é uma transformação bastante comum, não acontecerá o mesmo para os outros elementos, o carro e as outras rodas.

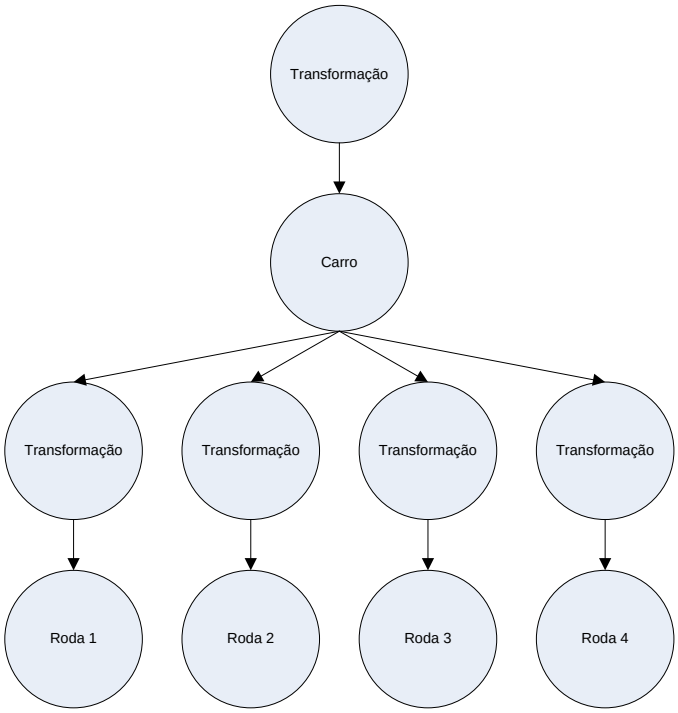


Figura 25 - Exemplo de grafo de cena

## 3.2 Definições de sistema de manufatura estudado

A partir desse tópico, as definições do funcionamento do núcleo de simulador de eventos discretos para sistemas de manufatura com visualização tridimensional serão feitas.

A primeira definição é o de sistemas de manufatura utilizado nesse estudo. Das funções existentes, esse trabalho utiliza principalmente operação e planejamento, agendamento e controle da produção, essa última auxiliando principalmente no controle, tanto numérico quanto visual, do que foi planejado.

Assim, na função operação, o objetivo é focado na operação de transformação física da peça, sendo a de montagem não estudada. Além da transformação da peça, há três tipos de operações auxiliares: movimentação, armazenamento e verificação; cada qual sendo representada pelo tipo de equipamento existente.

Dessas operações, cada uma é ligada a apenas um equipamento, possuindo o mesmo nome, enquanto o equipamento pode ser utilizado por mais de uma operação. Há equipamentos que possuem a capacidade de reter mais de um produto, porém apenas o mais antigo será processado por vez.

Toda a movimentação de recursos dentro de um processo de manufatura é representada apenas por uma entidade. Ou seja, a mesma peça que entra como matéria-prima, é trabalhada em todas as operação, até que se tenha um produto acabado.

Na figura 26, sobre a lógica de funcionamento do sistema de manufatura, inicialmente, quando um produto entrar no processo de manufatura, é verificado se o equipamento da operação seguinte está livre. Em caso afirmativo, o produto será alocado nele, enquanto que, em caso contrário, o produto ficará esperando na sua posição.

Uma vez dentro do equipamento, verifica-se o produto no que diz respeito ao fato de ser ou não o mais antigo internamente. Se sim, ele deve ser imediatamente operado, e caso não seja, fica esperando até que a condição torne-se verdadeira.

Após cada operação ter sido concluída, o produto é checado para caso dele ter percorrido todas as operações do processo de manufatura. Se isso for verdadeiro, o produto é retirado da fabrica mas, se for falso, é realocado na etapa de verificação do equipamento da próxima operação, e a lógica é reiniciada.

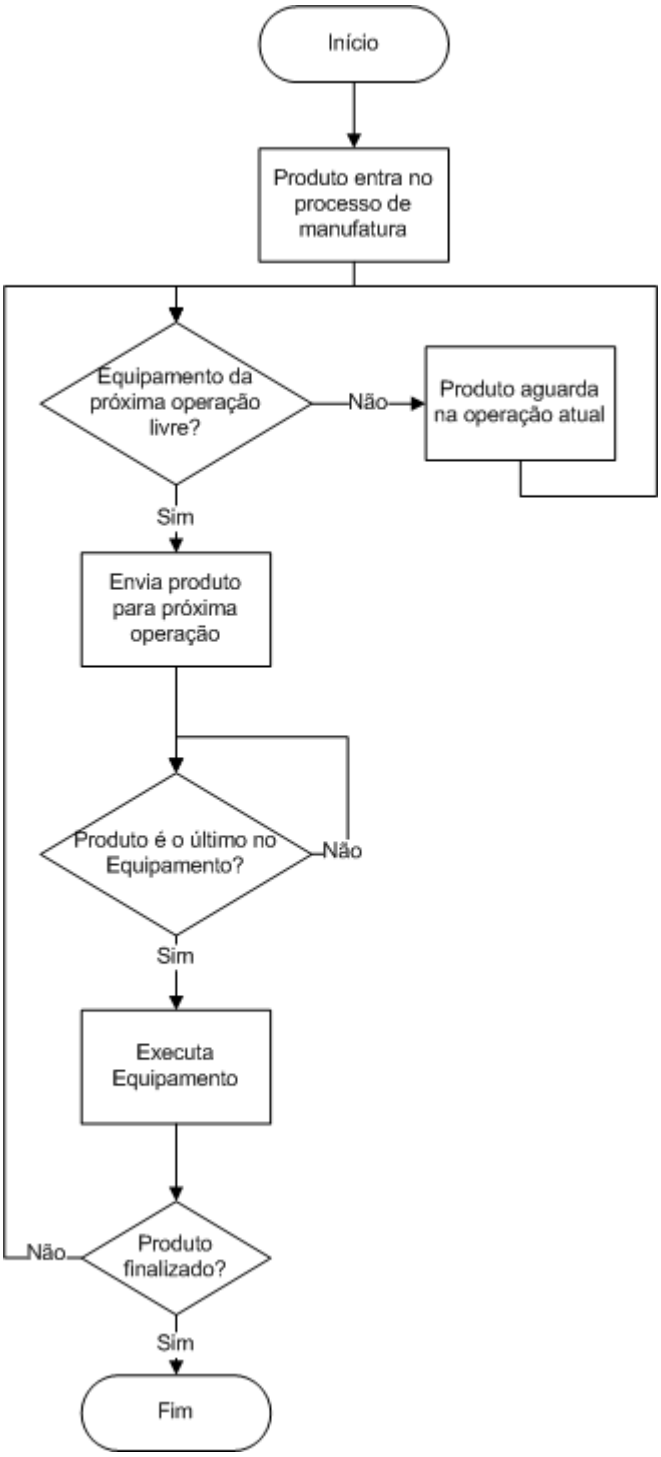


Figura 26 - Fluxograma da lógica de funcionamento do sistema de manufatura

## 3.3 Definições do núcleo do simulador de sistema de manufatura

O paradigma de simulação utilizado na elaboração do núcleo do simulador de sistema de manufatura foi uma combinação da abordagem três fases, detalhada no item e) da seção 2.1.2, por permitir o processamento paralelo de recursos e a sua divisão permite a sincronização da sequência do processamento dos métodos com o VR Juggler, com a abordagem orientada a objetos, detalhada no item f) da seção 2.1.2. Como o sistema é construído utilizando a linguagem de uso geral C++, algumas adaptações à realidade do trabalho desenvolvido foram feitas. Uma delas é a dispensa da elaboração explícita da lista de eventos futuros, devido à estrutura da programação orientada a objetos, pois os objetos já armazenam informações importantes, como por exemplo, o tempo de execução e a capacidade para as entidades máquinas e a posição dentro do processo de manufatura para os produtos.

A lógica do sistema de manufatura, descrita na Seção 3.2., servirá como base para o núcleo do simulador. As suas etapas são divididas em camadas que identifica onde o produto se encontra. Na tabela 4, todas as atividades, seus valores, a camada a qual pertencem e o caminho a percorrer, em caso de conflito da lógica estão descritas.

| Nome | Valor | Camada | Resultado | |
|---|---|---|---|---|
| | | | Sim | Não |
| Equipamento da próxima operação livre? | C1 | 1 | B1 | - |
| Envia produto para próxima operação | B1 | 1 | C2 | - |
| Produto é o último no equipamento? | C2 | 2 | B2 | - |
| Processa produto | B2 | 3 | C3 | - |
| Produto finalizado? | C3 | 4 | Retira produto da fábrica | C1 |

Tabela 4 - Valores dos elementos da lógica da manufatura

Tradicionalmente, a primeira fase, a fase A, é responsável pela atualização do *clock* do simulador. Porém, nesse trabalho, a atualização do *clock* é responsabilidade de outra classe no

sistema, em que é feito de forma independente, sendo no presente estudo desnecessária sua utilização.

Na segunda fase, a fase B, as atividades incondicionais são executadas, ou seja, em função do tempo do *clock*, e que estejam liberadas. Dessas, o elemento B1, mesmo sendo incondicional, é uma atividade que não depende do tempo. Por essa razão, ela é executada exatamente no instante em que a condição C1 for verdadeira. Ela envia o produto para o equipamento da próxima operação, alocando-o no início da fila interna e retirando-o do equipamento da operação corrente. Já o elemento B2 fará a execução do equipamento, através do método *run()*. Ela será a única atividade incondicional e em função do tempo do *clock*.

A terceira e última fase, a fase C, é responsável pela análise das atividades condicionais e não são dependentes funcionais do tempo, que examinam os estados do produto. Dessas, a atividade C1 verifica se o equipamento da próxima operação está livre para receber o produto. Já a segunda, a C2, examina se o produto é o último na fila interna do equipamento, para só então liberá-lo para a terceira camada. A C3, a última camada, analisa se o produto está finalizado.

## 3.4 Definições de ambiente de realidade virtual

### 3.4.1 Elementos do ambiente de realidade virtual

No ambiente de RV, os elementos que representam as unidades físicas do sistema de manufatura são compostos por dois tipos de objetos gráficos: os objetos móveis e os objetos estáticos. A principal característica dos objetos móveis é a possibilidade de ocorrer operações de transformações, enquanto que nos objetos estáticos não é encontrada essa características.

Para exemplificar, em um carro as rodas fazem o papel de objetos móveis, enquanto que os bancos fazem o papel de objetos estáticos. Logicamente, quando um carro se movimentar, os bancos farão o mesmo, porém eles não possuem a capacidade de se movimentar independentemente.

O simulador gerencia os objetos móveis que pertencem a uma unidade física através do uso de eventos visuais. Esses representam as operações de transformação básicas que podem ocorrer a um objeto móvel e sua sequência representa o roteiro quando a unidade física estiver em funcionamento.

Ainda segundo o exemplo da estrutura de um carro, se for observado o comportamento do motor quando esse recebe aceleração, alguns objetos internos realizam movimentos simples. Dentre esses objetos, pode-se destacar que os pistões fazem translação e o eixo faz rotação, todos numa sequência pré-determinada.

### 3.4.2 Laboratório multiusuário de visualização imersiva

O Laboratório Multiusuário de Visualização Imersiva (LMVI) da Universidade de São Paulo (USP), situado em São Carlos – SP, possui um dispositivo para multiprojeção de realidade virtual baseado em CAVE. Esse dispositivo é composto por três telas de projeção com dimensões: 3,2m x 2,4m para a tela frontal, 2,4m x 2,4m para a tela referente ao lado direito e 3,2m x 2,4m para a tela do chão. Na figura 27 está esquematizada a estrutura com as dimensões.

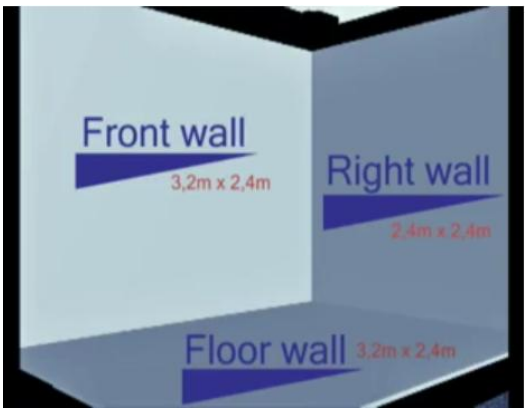


Figura 27 - Dimensões das telas de multi projeção

Para cada tela de projeção existem dois projetores Christie modelo DS+60/DW30/Matrix 3000 e um computador equipado com placa de vídeo Nvidia Quadro FX4500 e 4GB de memória RAM.

As conexões entre os computadores e os projetores são feitas através de cabos DVI. Existe um computador principal de onde é possível acessar ou controlar os três computadores responsáveis pelas telas.

A CAVE do LMVI permite o modo de visualização estéreo passivo, onde cada projetor é responsável pela imagem referente à visualização para cada olho. As imagens referentes ao olho direito e esquerdo são projetadas sobrepostas na tela para que, seja possível obter, pela utilização de óculos estereoscópicos, a sensação de imersão em quaisquer ambientes virtuais. Os óculos estereoscópicos utilizados no LMVI são da marca INFITEC *filters*. Esse tipo de sistema possibilita a imersão simultânea de vários usuários dentro de um ambiente virtual, bastando apenas que cada um destes usuários esteja equipado com um óculo estereoscópico.

O sistema operacional utilizado nos quatro computadores do LMVI é o Microsoft Windows XP Professional.

## 3.5 Modelagem computacional

As entidades que compõem o núcleo do simulador de sistemas de manufatura são separadas em três categorias principais: unidades físicas, unidades de informações e produto, como exibidas na figura 28.


Figura 28 - Unidades básicas do simulador

Na figura 28, as unidades físicas representam o agrupamento de tudo que for tangível e que interaja diretamente com o produto no processo produtivo. Elas são subdivididas em: a transformação, responsável pela alteração física do produto, agregando valor; o

armazenamento, local em que os produtos permanecem enquanto esperam até serem transformados; a movimentação, que faz a ligação entre elementos transformação e armazenamento; e a verificação, que avalia se um produto pós-processamento está apto a prosseguir no processo de manufatura. As opções para o produto nesta última unidade são: ser aceito, voltar para retrabalho ou ser descartado.

Já as unidades de informações dizem respeito à movimentação, ao processamento e ao armazenamento de informações do processo produtivo. Elas existem para tornar o sistema dinâmico, "inteligente" e com mais poder de tomada de decisão. Suas aplicações são: a sequência que um produto necessita percorrer; os eventos que vão ocorrer na simulação como um todo; dentre outros.

Finalmente, e mais importante, o produto é o elemento básico que ditará os parâmetros e os requisitos acerca do funcionamento do simulador e do sistema de manufatura.

Para uma definição mais ampla, o simulador está inserido em uma estrutura hierárquica composta por camadas abstratas que se comunicam através de interfaces, como pode ser visualizado na figura 29.

Nela, o sistema representado é constituído pelas camadas simulação (o núcleo), lógica, fábrica, processos, objetos, sendo composto por produtos e equipamentos e sistema de visualização e informação, que agrega gerenciador de eventos de cena e gerenciados de informações geradas.

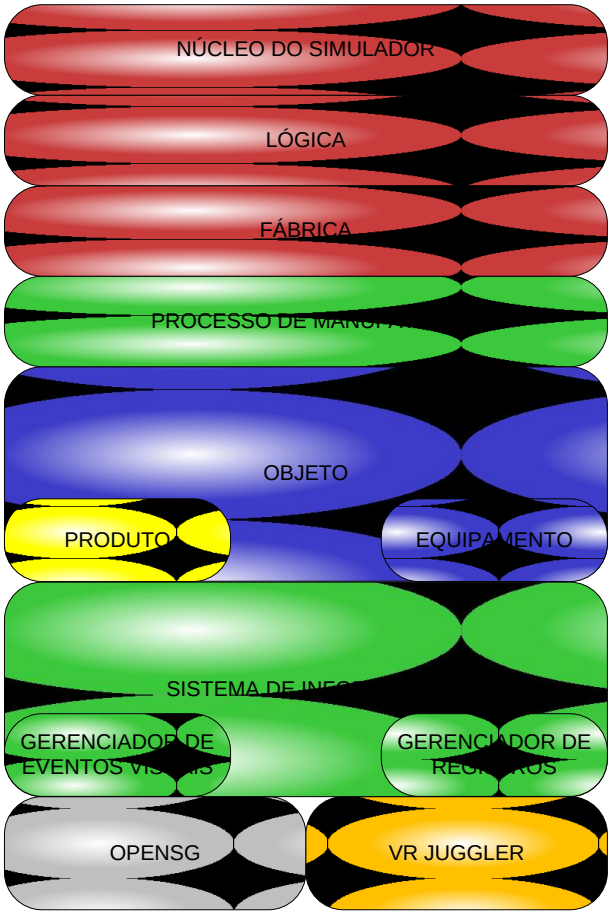


Figura 29 - Camadas do simulador

### 3.5.1 Diagrama de Classe

A partir da definição das camadas realizada na Seção 3.5., a modelagem das classes será feita, criando uma camada de gerenciamento em alto nível. A escolha de programação orientada a objetos foi feita devido à similaridade que esta abordagem tem com situações que ocorrem no mundo real. Na figura 30, o diagrama de classes do núcleo do simulador é apresentado.

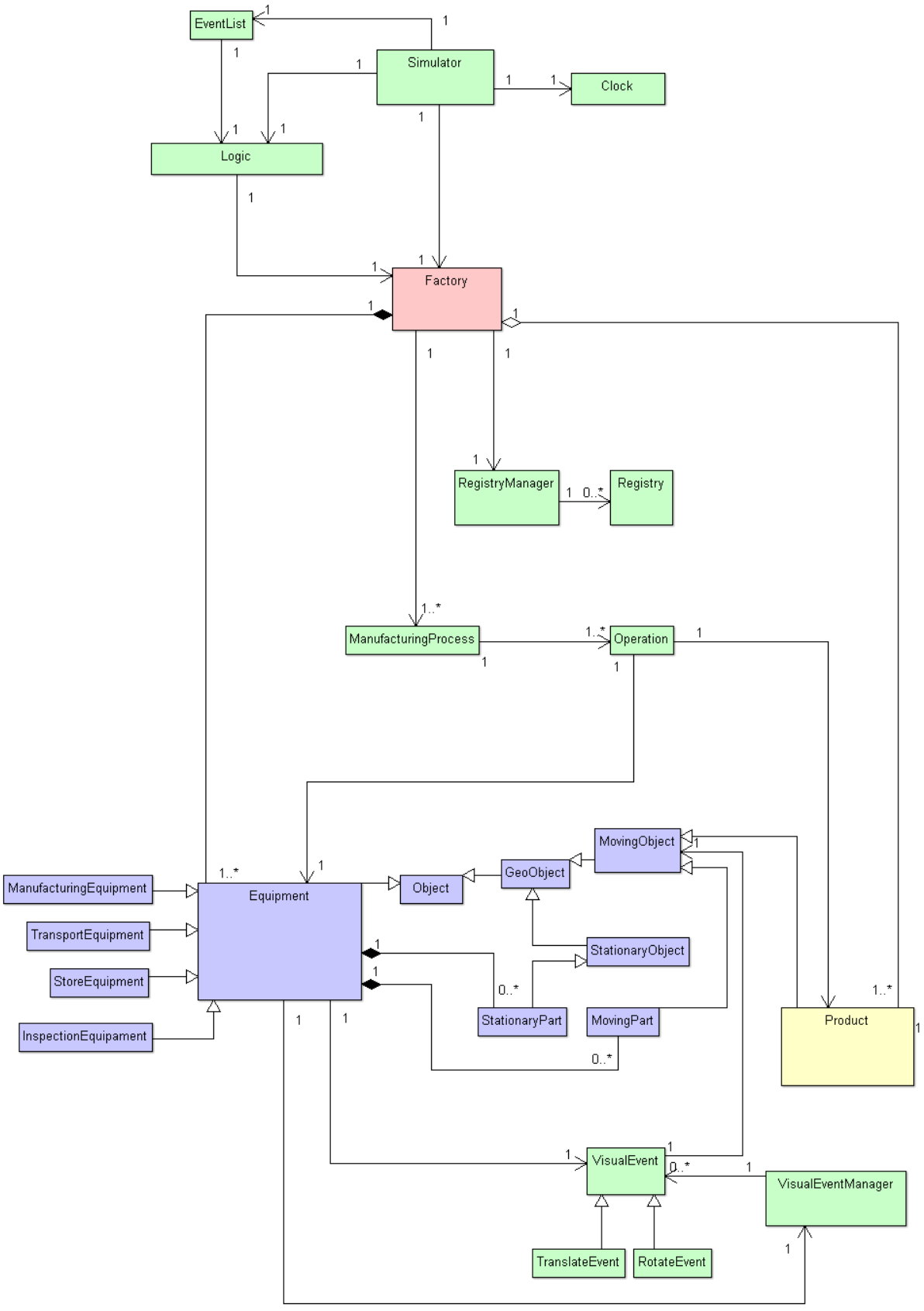


Figura 30 - Diagrama de classes do simulador

Na figura 30, por motivo de espaço, as classes estão representadas apenas por seus nomes já que todas as outras características podem ser encontradas no anexo A. A sua distribuição espacial é semelhante à dos elementos das camadas apresentadas anteriormente, inclusive com respeito às cores. No topo está localizada a classe *Simulator*, que gerencia todas as operações do núcleo do simulador. Ela se associa de um para um com a classe *Clock*, que gerencia o tempo global da simulação, um para um com a classe *EventList*, que fará a ação de execução das duas fases do núcleo do simulador, um para um com a classe *Logic*, que, por sua vez, cuidará da lógica do sistema de manufatura e um para vários com a classe *Factory*, pois a estrutura do núcleo do simulador permite que ele organize mais de uma fábrica.

Já a classe *Factory* possui ligações de um para um com a classe *RegistryManager*, para que todo evento na fábrica seja registrado, e um para vários com a classe *ManufacturingProcess*, pois uma fábrica pode, e deve, produzir mais de um tipo de produto. Por fim, a classe *Factory* tem relação de agregação de um para vários com a classe *Product*, já que os produtos são objetos dinâmicos, e uma relação de composição de um para vários com a classe *Equipment*, devido à natureza estática dos objetos equipamentos.

Por sua vez, a classe *ManufacturingProcess* possui associação de um para vários com a classe *Operation*, representando cada etapa do processo de manufatura e esse se associa como um para um com as classes *Equipment* e *Product*, pois sua execução, a cada instante, será feita de um produto por um equipamento.

A classe *Equipment* é herdada da classe *Object*, pois ela contém informações do seu posicionamento espacial. Além dessa herança, ela é composta pelas classes *MovingPart* e *StationaryPart*, enquanto que a classe *Product* é herdada da classe *MovingObject*. O funcionamento das classes a partir da classe *Object* é discutido a seguir.

A classe *Object* possui como característica principal o posicionamento espacial dos objetos físicos virtualmente representados. A classe *GeoObject* é herdada da classe *Object*, ela faz a representação visual geométrica de cada objeto físico virtualmente representado. Cada objeto instanciado da classe Geo*Object* terá um nó pertencente à API gráfica OpenSG. A partir da classe *GeoObject*, as classes *StationaryObject* e *MovingObject* são herdadas. A primeira é responsável pelos objetos fixos que irão compor a geometria imóvel do equipamento e a segunda representam os objetos que fazem movimentos independentes, do

tipo rotação e translação das partes internas do *Equipment* e o *Product*. Essa classe adiciona um nó de transformação dentro do grafo de cena numa uma posição hierarquicamente superior ao nó de gemetria da classe *GeoObject* para que receba as transformações de rotação e translação. A classe *VisualEvent* representa cada movimentação simples, seja de rotação ou de translação, realizada por uma instância da classe *MovingObject*, e o conjunto daquela é gerenciado pela classe *VisualEventManager*.

### 3.5.2 Diagramas de Sequência

Nesta seção, dois comportamentos básicos serão detalhados. São eles: o diagrama de sequência de lógica de funcionamento da fábrica e o diagrama de sequência do gerenciamento de ambiente virtual.

O diagrama de sequência do funcionamento da fábrica modelará o cenário básico do funcionamento da lógica do sistema de manufatura, como pode ser visto na figura 31.

Nela, o objeto genérico da classe *Simulator*, denominado "*:Simulator*" contem a regra para entrada de novos produtos na fábrica simulada e, após sua criação, passa o mesmo para o objeto genérico da classe *Logic*, denominado "*:Logic*". Sob seu controle, o objeto "*:Logic*" controla se o objeto genérico da classe *Product*, denominado "*:Product*", está finalizado, atravessando todas as operações do processo. Em caso afirmativo, o objeto genérico da classe *Simulator*, denominado "*:Simulator*", o eliminará. Já e em caso negativo, o objeto "*:Logic*" verifica qual será o próximo equipamento de acordo com a sequência de operações do produto. Feita a seleção do próximo objeto genérico da classe *Equipment*, denominado "*:Equipment*", o objeto "*:Logic*" verifica se aquele está livre. Para respostas afirmativas, é enviado o produto para esse equipamento, que executa a si mesmo, através do método *run()* e gera a animação tridimensional; a ser discutido no próximo diagrama de sequência.

O diagrama de sequência da geração da realidade virtual tem início quando o objeto "*:Equipment*" executa o método *run()*.

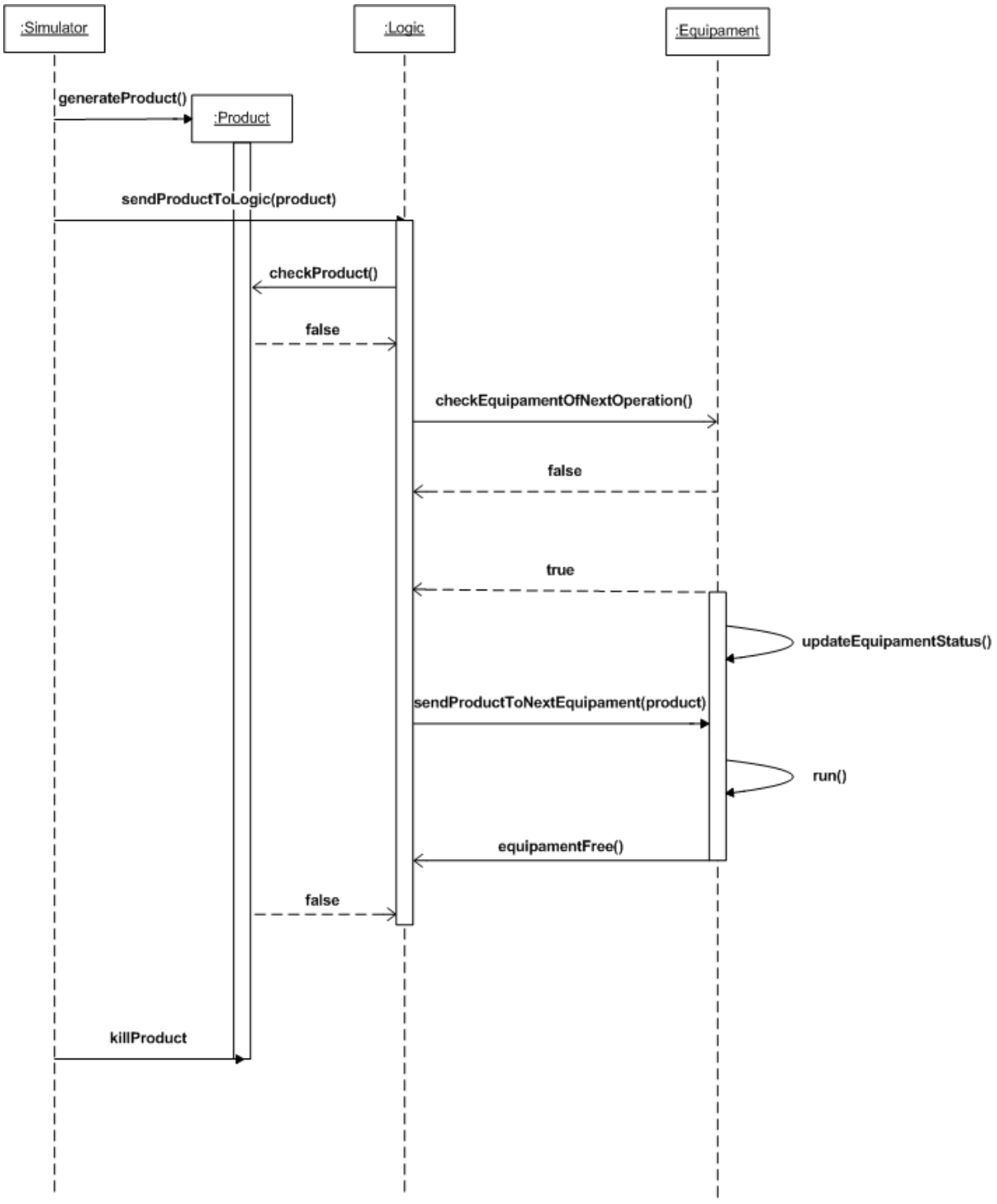


Figura 31 - Diagrama de sequência da lógica do funcionamento da fábrica

Na figura 32, o objeto "*:Equipment*" cria objetos genéricos da classe *VisualEvent*, denominado "*:VisualEvent*". Consecutivamente, a partir de suas criações, o objeto "*:Equipment*" coloca o objeto "*:VisualEvent*" em uma pilha dentro do objeto genérico da Classe *VisualEventManager*, denominado "*:VisualEventManager*".

O objeto "*:EventoVistual*" se associa à um objeto genérico da classe *MovingObject*, denominado "*:MovingObject*". Ao fazer isso, o "*:MovingObject*" estará informando ao "*:MovingObject*" sobre o seu funcionamento.

Após ter feito a inicialização de todos os objetos "*:VisualEvent*", o objeto "*:VisualEventManager*" administra o funcionamento de todos os objetos "*:VisualEvent*" no momento em que eles acontecem.

A área tracejada identifica que as operações irão se repetir enquanto a variável *time*, que serve como argumento para as operações, for menor ou igual à um valor pré-configurado pré-configurado, representando a animação das partes móveis.

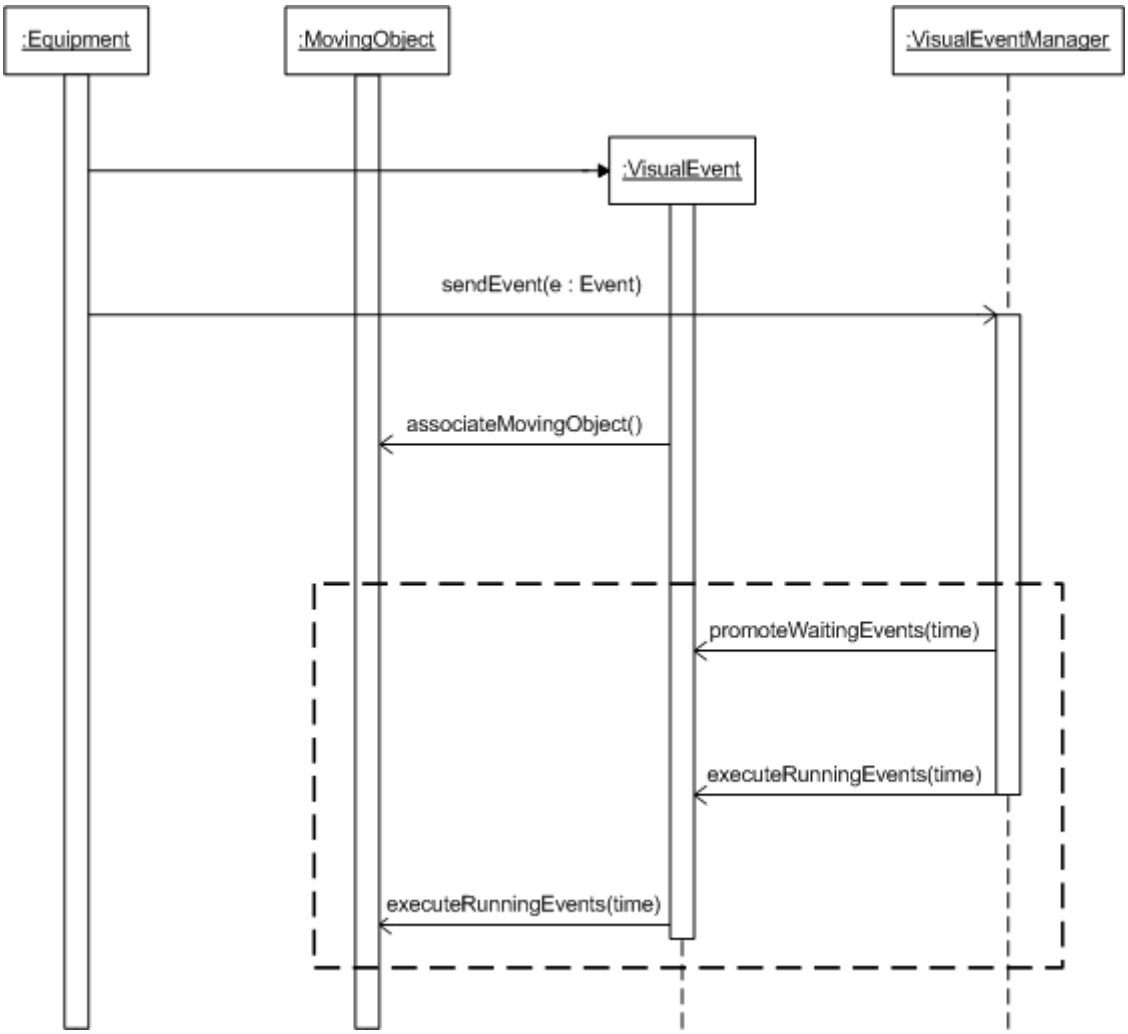


Figura 32 - Diagrama de sequência da geração de objetos gráficos

## 3.6 Prototipação

Neste tópico serão definidas as informações básicas do núcleo do simulador para entender o funcionamento do mesmo. Assim, o protótipo do núcleo do simulador foi utilizado para verificar se o mesmo está funcionando, através de um exemplo no próximo item. Como mostra a figura 33.

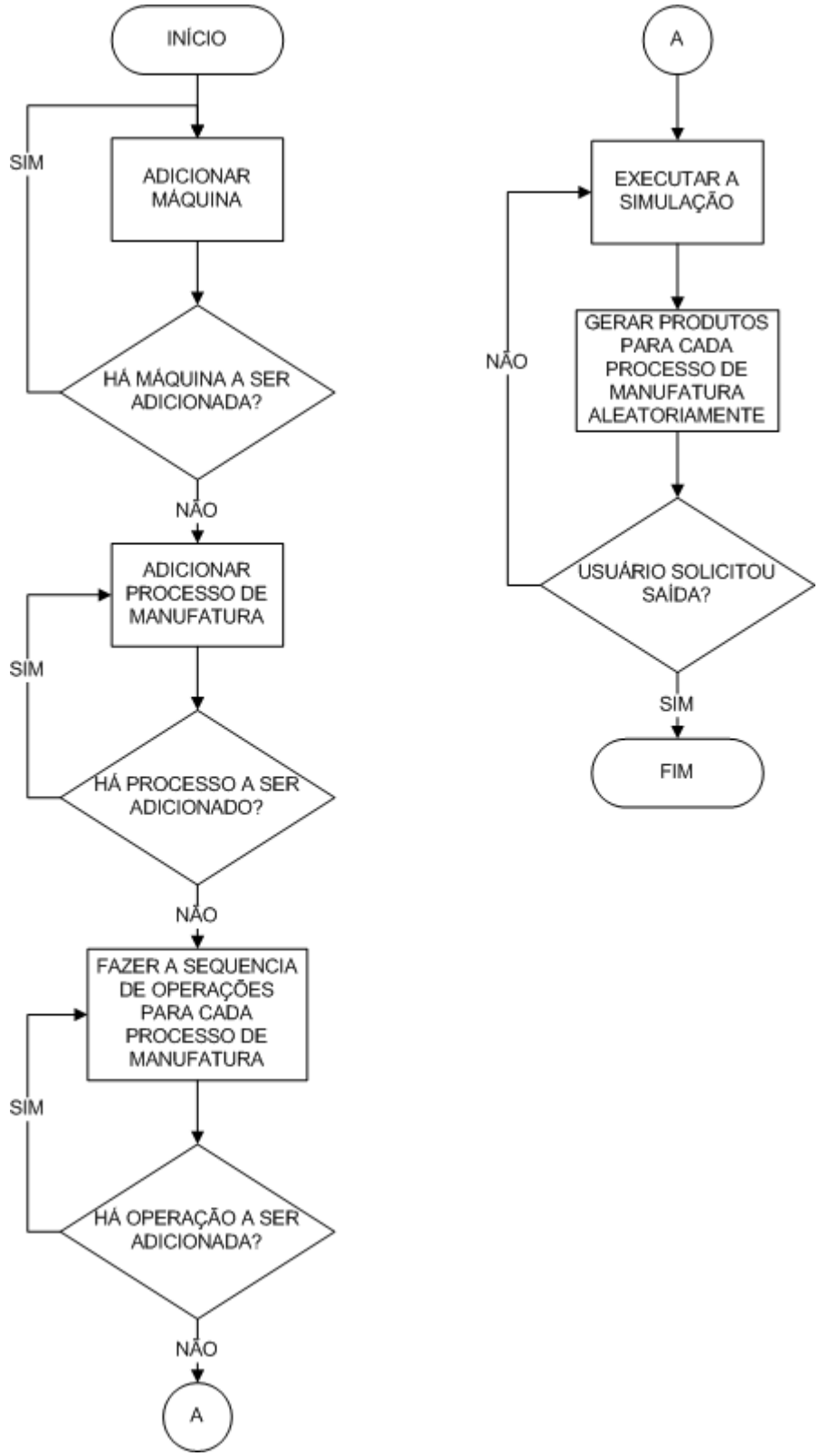


Figura 33 - Protótipo do núcleo do simulador

Conforme a figura 33, devem ser seguido os passos desse fluxograma para realizar experimentos de simulação utilizando o protótipo do núcleo de simulação descrito neste trabalho. Eles estão divididos em dois grupos. Inicialmente, no grupo da esquerda, de pré-simulação, estão os passos de inicialização da indústria a ser simulada, que consiste na inclusão de equipamentos, através da operação "*add*Equipment*", em que o asterisco (*) deverá ser substituído pelo tipo de equipamento (transformação, transporte, armazenagem ou verificação) da classe *Factory*, de processos de manufatura, através da operação "*addManufacturingProcess*" da classe *Factory*, e de operações, através da operação "*addOperation*" da classe *ManufacturingProcess*.

Já no grupo da direita, de simulação,prosseguindo no fluxograma estão as etapas recursivas de atualização ocorrem, com a execução do simulador e inserção de novos produtos na fábrica aleatoriamente, através da operação "*generateRandomProduct*" da classe *Simulator*, colocada na operação "*preFrame*" da mesma classe, e o usuário tem a opção de encerrar a simulação, pressionando a tecla *ESC*.

Como ainda não há na CAVE nenhuma forma de interação do usuário e a interação dele com o ambiente virtual não está nos objetivos desse trabalho, as informações serão alteradas diretamente no código.

Como explicados anteriormente, os equipamentos, são divididos em quatro tipos: movimentação, armazenamento, transformação e verificação. No simulador, cada um desses terá seu comportamento visual pré-definido. Sendo assim, o usuário fará a inclusão, em nível de linguagem de programação C++, das informações descritas anteriormente e compete a ele realizar toda a configuração das bibliotecas descritas anteriormente no tópico de ferramentas computacionais. Maiores informações podem ser adquiridas no Anexo A, que contém todo o código-fonte da aplicação.

# 4. TESTE DE APLICAÇÃO DA PROPOSTA

O objetivo desse capítulo é verificar, através de uma aplicação, a proposta exposta nos capítulos anteriores. Ele é dividido em dois tópicos, a composição dos elementos que irão formar o exemplo e os resultados e suas análises.

Já o objetivo principal da aplicação consiste em gerar de um conjunto de dados hipotéticos que se assemelhe, em nível de informação e visual, a um sistema real de manufatura simples. Os seus parâmetros são os que estão definidos nas tabelas a seguir.

Outro ator importante a se destacar é que o teste de aplicação não possui caráter de validação, pois necessitaria de um exemplo externo que se aproxime do mesmo. Então, a sua verificação será feita, tanto na visualização de uma geração de dados a serem analisados como a sua visualização tridimensional.

## 4.1 Composição

Na composição, os primeiros elementos a serem detalhados da aplicação são os equipamentos. Mais detalhes na tabela 5.

| ID | Nome | Tipo | Inicio Aleatório | Final Aleatório | Tempo | x | y | z | Capacidade |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Conv1 | Transporte | - | - | 20 | 0 | 0 | 0 | 1 |
| 2 | Conv2 | Transporte | - | - | 20 | 0,95 | 0 | -0,75 | 1 |
| 3 | Buf1 | Armazenamento | - | - | 1 | 0,95 | 0 | 0 | 2 |
| 4 | Turning | Manufatura | 30 | 50 | - | 1,65 | 0 | 0 | 1 |
| 5 | Insp1 | Inspeção | - | - | 10 | 2,35 | 0 | 0 | 1 |
| 6 | Conv3 | Transporte | - | - | 20 | 3,3 | 0 | 0 | 1 |
| 7 | Conv4 | Transporte | - | - | 20 | 4,25 | 0 | -0,75 | 1 |
| 8 | Buf2 | Armazenamento | - | - | 1 | 4,25 | 0 | 0 | 2 |
| 9 | Milling | Manufatura | 10 | 20 | - | 4,95 | 0 | 0 | 1 |
| 10 | Insp2 | Inspeção | - | - | 10 | 5,65 | 0 | 0 | 1 |

Tabela 5 - Equipamentos da aplicação

Na tabela 5, as linhas representam cada equipamento e as colunas as informações deles, que são: a ID é um número que identifica o equipamento unicamente e que terá efeito

apenas didático, pois não será utilizado na aplicação; o nome é outra forma de identifica-lo; o tipo é qual agrupamento que ele pertence; o início aleatório e o final aleatório indicam a faixa do tempo de funcionamento quando ele for aleatório; o tempo indica quando ele for fixo; o x, o y e o z informam o centro geométrico e o conjunto deles o arranjo físico, como visto na figura 34; e, no fim, a sua capacidade.

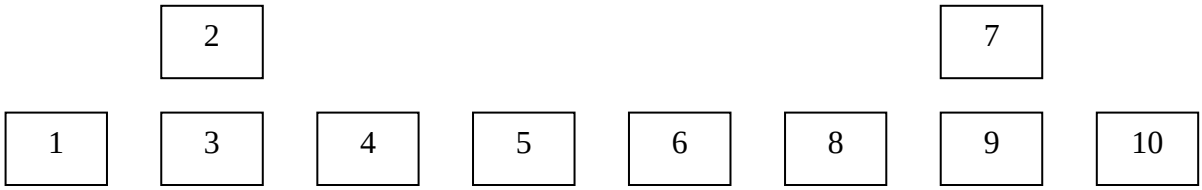


Figura 34 - Arranjo físico global

Na figura 34, cada número representa a ID do equipamento, com sua posição geométrica dentro do arranjo físico. Além dos equipamentos, a aplicação possui três processos de manufatura e suas respectivas operações. Nas figuras 35 a 37, estão explicitados os processos de manufatura para cada tipo de produto.

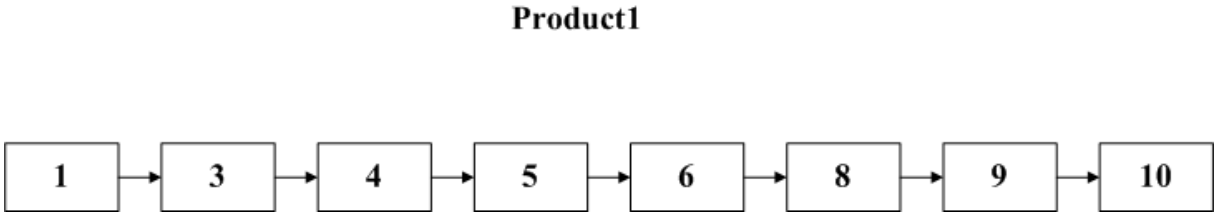


Figura 35 - Processo de manufatura do produto Product1

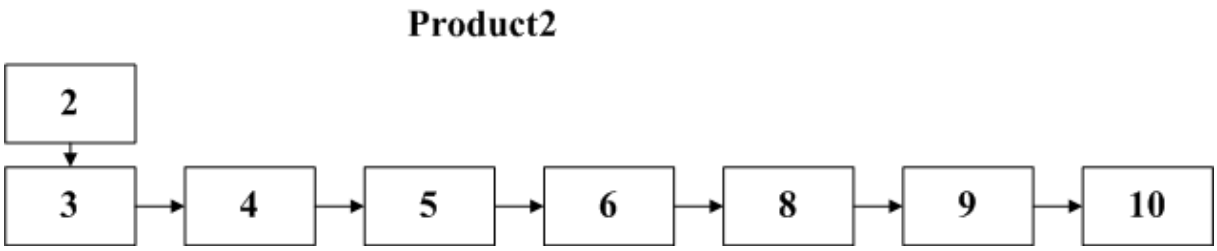


Figura 36 - Processo de manufatura do produto Product2

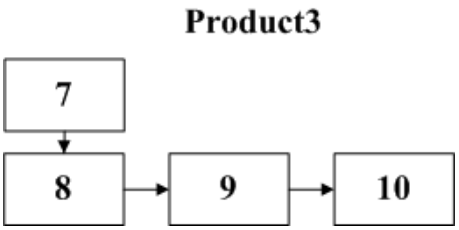


Figura 37 - Processo de manufatura do produto Product3

Os processos de manufatura dos produtos 1 e 2, da figura 35 e da figura 36, respectivamente, possuem um processo de manufatura bem simular, já que atravessam os mesmos equipamentos de manufatura, o 4 e o 9. Enquanto que o produto 3, do arranjo físico da figura 37, só será manufaturado pelo equipamento 9.

## 4.2 Resultados

A simulação do aplicativo foi feita utilizando dois períodos de tempo. Do primeiro, de 600 segundos, foram coletadas informações acerca do comportamento dos equipamentos. Já o segundo teste aconteceu por um período de 6000 segundos. Esse por sua vez foi idealizado para aferir informações dos produtos.

Na primeira parte, sob uma simulação de 600 segundos os resultados dos equipamentos são agrupados por tipo, por terem comportamentos semelhantes. Esses resultados mostram quantos e quais são os produtos que passaram por eles e por quanto tempo, total e absoluto, ele foi utilizado, além do tempo, absoluto e por produtos, da espera.

O primeiro agrupamento será o dos equipamentos de transportes. Como podem ser vistos nas tabelas 6 a 9. As colunas indicam a fase em que o produto está, a camada da lógica, o nome do produto, seu ID de acordo com a ordem de chegada, a posição dentro do processo de manufatura, o Equipamento, o tempo do relógio do simulador, quanto tempo ele esperou dentro do Equipamento e quanto tempo ele foi operado.

| Fase | Camada | Produto | ID | Posição | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product1 | 1 | 0 | Conv1 | 3 | 0 | 21 |
| C | 1 | Product1 | 2 | 0 | Conv1 | 24 | 0 | 22 |
| C | 1 | Product1 | 3 | 0 | Conv1 | 46 | 19 | 22 |
| C | 1 | Product1 | 4 | 0 | Conv1 | 87 | 45 | 21 |
| C | 1 | Product1 | 5 | 0 | Conv1 | 153 | 47 | 21 |
| C | 1 | Product1 | 6 | 0 | Conv1 | 221 | 47 | 21 |
| C | 1 | Product1 | 7 | 0 | Conv1 | 289 | 33 | 21 |
| C | 1 | Product1 | 8 | 0 | Conv1 | 343 | 98 | 21 |
| C | 1 | Product1 | 9 | 0 | Conv1 | 462 | 0 | 21 |
| Tempo Total | | | | | | | | 480 |
| Taxa de Utilização | | | | | | | | 80% |
| Tempo Total de Espera | | | | | | | | 289 |
| Média de Espera por produto | | | | | | | | 32,11 |

Tabela 6 - Equipamento Conv1

| Fase | Camada | Produto | ID | Posição | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product2 | 1 | 0 | Conv2 | 8 | 0 | 21 |
| C | 1 | Product2 | 2 | 0 | Conv2 | 29 | 350 | 22 |
| C | 1 | Product2 | 3 | 0 | Conv2 | 401 | 98 | 21 |
| C | 1 | Product2 | 4 | 0 | Conv2 | 520 | 56 | 21 |
| C | 1 | Product2 | 5 | 0 | Conv2 | 597 | 0 | 3 |
| Tempo Total | | | | | | | | 592 |
| Taxa de Utilização | | | | | | | | 99% |
| Tempo Total de Espera | | | | | | | | 504 |
| Média de Espera por Produto | | | | | | | | 100,8 |

Tabela 7 - Equipamento Conv2

| Fase | Camada | Produto | ID | Posição | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product1 | 1 | 4 | Conv3 | 98 | 0 | 20 |
| C | 1 | Product2 | 1 | 4 | Conv3 | 164 | 19 | 20 |
| C | 1 | Product1 | 2 | 4 | Conv3 | 232 | 5 | 20 |
| C | 1 | Product1 | 3 | 4 | Conv3 | 300 | 0 | 20 |
| C | 1 | Product1 | 4 | 4 | Conv3 | 354 | 0 | 20 |
| C | 1 | Product1 | 5 | 4 | Conv3 | 412 | 0 | 20 |
| C | 1 | Product1 | 6 | 4 | Conv3 | 473 | 0 | 20 |
| C | 1 | Product1 | 7 | 4 | Conv3 | 531 | 0 | 20 |
| Tempo Total | | | | | | | | 184 |
| Taxa de Utilização | | | | | | | | 31% |
| Tempo Total de Espera | | | | | | | | 24 |
| Média de Espera por Produto | | | | | | | | 3 |

Tabela 8 - Equipamento Conv3

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product3 | 1 | 0 | Conv4 | 18 | 0 | 21 |
| C | 1 | Product3 | 2 | 0 | Conv4 | 39 | 0 | 22 |
| C | 1 | Product3 | 3 | 0 | Conv4 | 61 | 0 | 22 |
| C | 1 | Product3 | 4 | 0 | Conv4 | 83 | 0 | 22 |
| C | 1 | Product3 | 5 | 0 | Conv4 | 105 | 0 | 22 |
| C | 1 | Product3 | 6 | 0 | Conv4 | 127 | 0 | 22 |
| C | 1 | Product3 | 7 | 0 | Conv4 | 149 | 6 | 22 |
| C | 1 | Product3 | 8 | 0 | Conv4 | 222 | 0 | 21 |
| C | 1 | Product3 | 9 | 0 | Conv4 | 313 | 0 | 21 |
| C | 1 | Product3 | 10 | 0 | Conv4 | 335 | 0 | 22 |
| C | 1 | Product3 | 11 | 0 | Conv4 | 437 | 0 | 21 |
| Tempo Total | | | | | | | | 244 |
| Taxa de Utilização | | | | | | | | 41% |
| Tempo Total de Espera | | | | | | | | 6 |
| Média de Espera por Produto | | | | | | | | 0,55 |

Tabela 9 - Equipamento Conv4

Nas tabelas 6 a 9, pode ser concluído que os produtos Product1 e o Product2 possuem os tempos totais e a média de espera por produto dos equipamentos de transporte que eles utilizam mais elevados que o equipamento utilizado pelo produto Product3. Isso é devido

principalmente ao maior número das operações de seus respectivos processos de manufatura, fazendo com que sua liberação fique dependente de mais equipamentos estarem livres.

O próximo grupo, das tabelas 10 e 11, compreende os equipamentos de armazenamento. As colunas indicam a fase em que o produto está, a camada da lógica, o nome do produto, seu ID de acordo com a ordem de chegada, a posição dentro do processo de manufatura, o Equipamento, o tempo do relógio do simulador, quanto tempo ele esperou dentro do Equipamento e quanto tempo ele foi operado.

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product1 | 1 | 1 | Buf1 | 24 | 0 | 1 |
| C | 1 | Product2 | 1 | 1 | Buf1 | 29 | 56 | 1 |
| C | 1 | Product1 | 2 | 1 | Buf1 | 46 | 64 | 2 |
| C | 1 | Product1 | 3 | 1 | Buf1 | 87 | 66 | 2 |
| C | 1 | Product1 | 4 | 1 | Buf1 | 153 | 66 | 2 |
| C | 1 | Product1 | 5 | 1 | Buf1 | 221 | 52 | 1 |
| C | 1 | Product1 | 6 | 1 | Buf1 | 289 | 56 | 2 |
| C | 1 | Product1 | 7 | 1 | Buf1 | 343 | 59 | 2 |
| C | 1 | Product2 | 2 | 1 | Buf1 | 401 | 56 | 2 |
| C | 1 | Product1 | 8 | 1 | Buf1 | 462 | 75 | 2 |
| C | 1 | Product2 | 3 | 1 | Buf1 | 520 | 0 | 2 |
| Tempo Total | | | | | | | | 569 |
| Taxa de Utilização | | | | | | | | 95% |
| Tempo Total de Espera | | | | | | | | 550 |
| Média de Espera por Produto | | | | | | | | 50 |

Tabela 10 - Equipamento Buf1

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product3 | 1 | 1 | Buf2 | 39 | 0 | 1 |
| C | 1 | Product3 | 2 | 1 | Buf2 | 61 | 6 | 1 |
| C | 1 | Product3 | 3 | 1 | Buf2 | 83 | 8 | 1 |
| C | 1 | Product3 | 4 | 1 | Buf2 | 105 | 17 | 1 |
| C | 1 | Product1 | 1 | 5 | Buf2 | 119 | 23 | 2 |
| C | 1 | Product3 | 5 | 1 | Buf2 | 127 | 26 | 2 |
| C | 1 | Product3 | 6 | 1 | Buf2 | 149 | 25 | 2 |
| C | 1 | Product3 | 7 | 1 | Buf2 | 177 | 26 | 2 |
| C | 1 | Product2 | 1 | 5 | Buf2 | 204 | 24 | 2 |
| C | 1 | Product3 | 8 | 1 | Buf2 | 243 | 28 | 2 |
| C | 1 | Product1 | 2 | 5 | Buf2 | 258 | 24 | 2 |
| C | 1 | Product1 | 3 | 5 | Buf2 | 321 | 20 | 1 |
| C | 1 | Product3 | 9 | 1 | Buf2 | 335 | 25 | 2 |
| C | 1 | Product3 | 10 | 1 | Buf2 | 357 | 28 | 2 |
| C | 1 | Product1 | 4 | 5 | Buf2 | 375 | 25 | 2 |
| C | 1 | Product1 | 5 | 5 | Buf2 | 433 | 21 | 1 |
| C | 1 | Product3 | 11 | 1 | Buf2 | 458 | 24 | 1 |
| C | 1 | Product1 | 6 | 5 | Buf2 | 494 | 12 | 1 |
| C | 1 | Product1 | 7 | 5 | Buf2 | 552 | 0 | 1 |
| Tempo Total | | | | | | | | 391 |
| Taxa de Utilização | | | | | | | | 65% |
| Tempo Total de Espera | | | | | | | | 362 |
| Média de Espera por Produto | | | | | | | | 19,05 |

Tabela 11 - Equipamento Buf2

Das tabelas 10 e 11, confirmando a conclusão supracitada, quanto mais equipamentos ainda restam para chegar ao fim do processo de manufatura, maior será o tempo e, consequentemente, a taxa de utilização do equipamento e o tempo de espera.

O terceiro grupo, das tabelas 12 e 13, abrange os equipamentos de manufatura. As colunas indicam a fase em que o produto está, a camada da lógica, o nome do produto, seu ID de acordo com a ordem de chegada, a posição dentro do processo de manufatura, o Equipamento, o tempo do relógio do simulador, quanto tempo ele esperou dentro do Equipamento e quanto tempo ele foi operado.

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product1 | 1 | 2 | Turning | 26 | 0 | 60 |
| C | 1 | Product2 | 1 | 2 | Turning | 87 | 0 | 66 |
| C | 1 | Product1 | 2 | 2 | Turning | 153 | 0 | 68 |
| C | 1 | Product1 | 3 | 2 | Turning | 221 | 0 | 68 |
| C | 1 | Product1 | 4 | 2 | Turning | 289 | 0 | 54 |
| C | 1 | Product1 | 5 | 2 | Turning | 343 | 0 | 58 |
| C | 1 | Product1 | 6 | 2 | Turning | 401 | 0 | 61 |
| C | 1 | Product1 | 7 | 2 | Turning | 462 | 0 | 58 |
| C | 1 | Product2 | 2 | 2 | Turning | 520 | 0 | 77 |
| C | 1 | Product1 | 8 | 2 | Turning | 597 | 0 | 3 |
| Tempo Total | | | | | | | | 573 |
| Taxa de Utilização | | | | | | | | 96% |

Tabela 12 - Equipamento Turning

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product3 | 1 | 2 | Milling | 41 | 0 | 27 |
| C | 1 | Product3 | 2 | 2 | Milling | 69 | 0 | 24 |
| C | 1 | Product3 | 3 | 2 | Milling | 93 | 0 | 31 |
| C | 1 | Product3 | 4 | 2 | Milling | 124 | 0 | 25 |
| C | 1 | Product1 | 1 | 6 | Milling | 149 | 0 | 28 |
| C | 1 | Product3 | 5 | 2 | Milling | 177 | 0 | 27 |
| C | 1 | Product3 | 6 | 2 | Milling | 204 | 0 | 28 |
| C | 1 | Product3 | 7 | 2 | Milling | 232 | 0 | 26 |
| C | 1 | Product2 | 1 | 6 | Milling | 258 | 0 | 30 |
| C | 1 | Product3 | 8 | 2 | Milling | 288 | 0 | 26 |
| C | 1 | Product1 | 2 | 6 | Milling | 314 | 0 | 29 |
| C | 1 | Product1 | 3 | 6 | Milling | 343 | 0 | 27 |
| C | 1 | Product3 | 9 | 2 | Milling | 370 | 0 | 30 |
| C | 1 | Product3 | 10 | 2 | Milling | 400 | 0 | 27 |
| C | 1 | Product1 | 4 | 6 | Milling | 427 | 0 | 29 |
| C | 1 | Product1 | 5 | 6 | Milling | 456 | 0 | 28 |
| C | 1 | Product3 | 11 | 2 | Milling | 484 | 0 | 24 |
| C | 1 | Product1 | 6 | 6 | Milling | 508 | 0 | 30 |
| C | 1 | Product1 | 7 | 6 | Milling | 554 | 0 | 25 |
| Tempo Total | | | | | | | | 521 |
| Taxa de Utilização | | | | | | | | 87% |

Tabela 13 - Equipamento Milling

Duas conclusões acerca das informações das tabelas 12 e 13 são: não está acontecendo filas após o produto ser manufaturado, logo os equipamentos de transporte e armazenamento estão desempenhando seus papéis; e, igual aos grupos de equipamentos anteriores, quanto mais distante da última operação do processo de manufatura, maior será sua utilização.

E, por último, nas tabelas 14 e 15 o grupo dos equipamentos de verificação. As colunas indicam a fase em que o produto está, a camada da lógica, o nome do produto, seu ID de acordo com a ordem de chegada, a posição dentro do processo de manufatura, o Equipamento, o tempo do relógio do simulador, quanto tempo ele esperou dentro do Equipamento e quanto tempo ele foi operado.

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Espera | Execução |
|---|---|---|---|---|---|---|---|---|
| C | 1 | Product1 | 1 | 3 | Insp1 | 87 | 0 | 10 |
| C | 1 | Product2 | 1 | 3 | Insp1 | 153 | 0 | 10 |
| C | 1 | Product1 | 2 | 3 | Insp1 | 221 | 0 | 10 |
| C | 1 | Product1 | 3 | 3 | Insp1 | 289 | 0 | 10 |
| C | 1 | Product1 | 4 | 3 | Insp1 | 343 | 0 | 10 |
| C | 1 | Product1 | 5 | 3 | Insp1 | 401 | 0 | 10 |
| C | 1 | Product1 | 6 | 3 | Insp1 | 462 | 0 | 10 |
| C | 1 | Product1 | 7 | 3 | Insp1 | 520 | 0 | 10 |
| C | 1 | Product2 | 2 | 3 | Insp1 | 597 | 0 | 2 |
| Tempo Total | | | | | | | | 82 |
| Taxa de Utilização | | | | | | | | 14% |

Tabela 14 - Equipamento Insp1

| Fase | Camada | Produto | ID | Posicao | Equipamento | Tempo | Execução |
|---|---|---|---|---|---|---|---|
| C | 1 | Product3 | 1 | 3 | Insp2 | 69 | 10 |
| C | 1 | Product3 | 2 | 3 | Insp2 | 93 | 10 |
| C | 1 | Product3 | 3 | 3 | Insp2 | 124 | 10 |
| C | 1 | Product3 | 4 | 3 | Insp2 | 149 | 10 |
| C | 1 | Product1 | 1 | 7 | Insp2 | 177 | 10 |
| C | 1 | Product3 | 5 | 3 | Insp2 | 204 | 10 |
| C | 1 | Product3 | 6 | 3 | Insp2 | 232 | 10 |
| C | 1 | Product3 | 7 | 3 | Insp2 | 258 | 9 |
| C | 1 | Product2 | 1 | 7 | Insp2 | 288 | 10 |
| C | 1 | Product3 | 8 | 3 | Insp2 | 314 | 10 |
| C | 1 | Product1 | 2 | 7 | Insp2 | 343 | 10 |
| C | 1 | Product1 | 3 | 7 | Insp2 | 370 | 10 |
| C | 1 | Product3 | 9 | 3 | Insp2 | 400 | 10 |
| C | 1 | Product3 | 10 | 3 | Insp2 | 427 | 10 |
| C | 1 | Product1 | 4 | 7 | Insp2 | 456 | 10 |
| C | 1 | Product1 | 5 | 7 | Insp2 | 484 | 10 |
| C | 1 | Product3 | 11 | 3 | Insp2 | 508 | 10 |
| C | 1 | Product1 | 6 | 7 | Insp2 | 538 | 10 |
| C | 1 | Product1 | 7 | 7 | Insp2 | 580 | 10 |
| Tempo Total | | | | | | | 189 |
| Taxa de Utilização | | | | | | | 32% |

Tabela 15 - Equipamento Insp2

Nos dois equipamentos das tabelas 14 e 15 a inspeções funcionaram corretamente, já que não foi programado nenhum tipo de rejeição, e não houve espera no Insp1, devido a não ocorrência no Insp2, por se tratar da última operação dos três processos de manufatura.

A segunda etapa da simulação da aplicação fornece informações sobre o tempo de processamento de cada produto e foi feito por um período de 6000 segundos. Mais detalhes nos gráficos das figuras 38, 39 e 40.

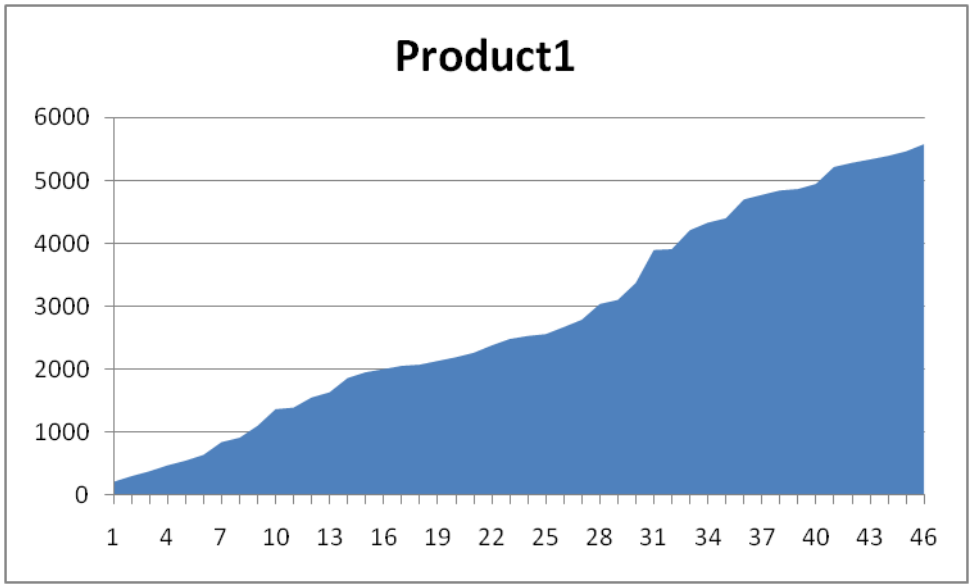


Figura 38 - Tempo de processamento do produto Product1

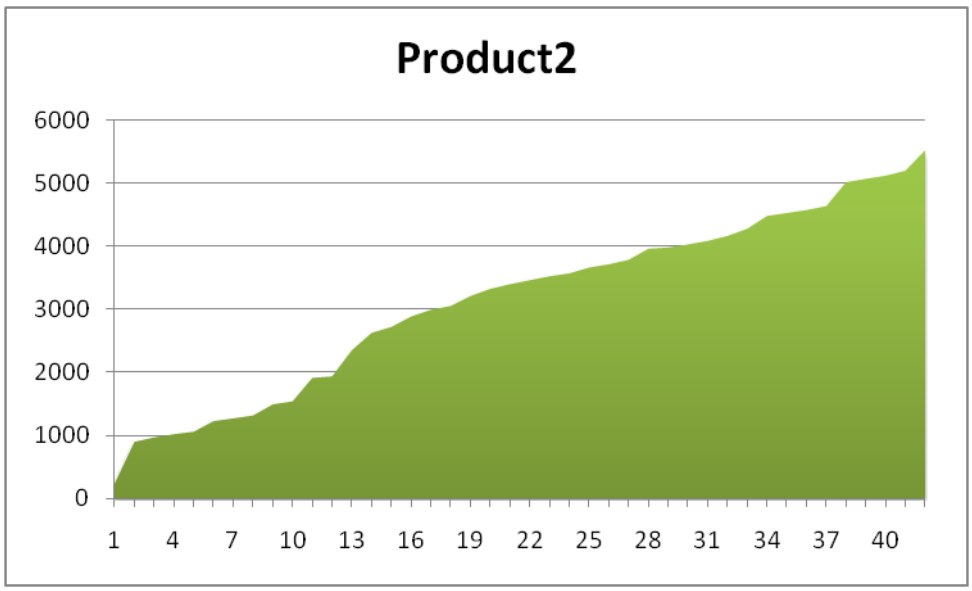


Figura 39 - Tempo de processamento do produto Product2

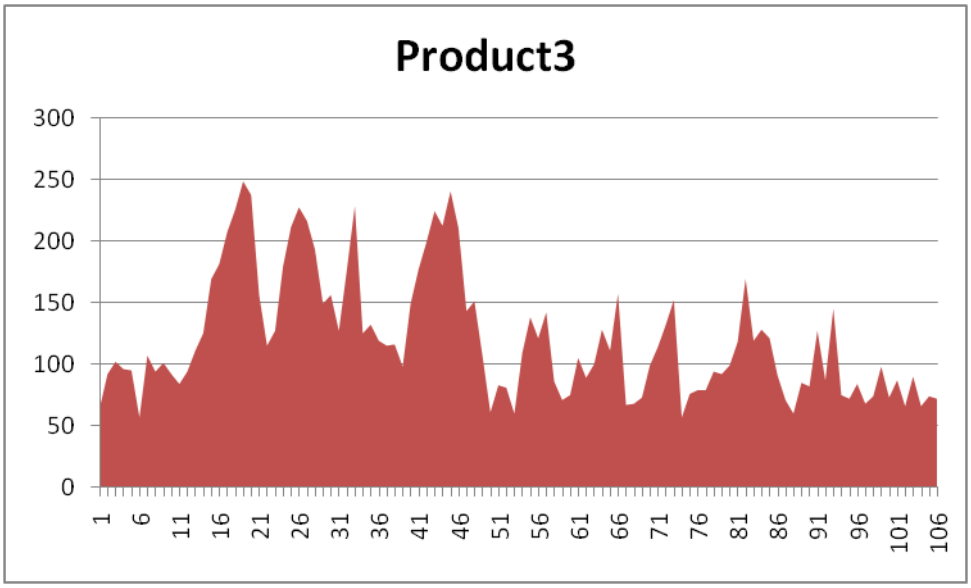


Figura 40 - Tempo de processamento do produto Product3

Nos gráficos, o eixo y representa o tempo de processamento e o eixo x a identificação de cada produto. Na figura 38 e na figura 39, os tempos de processamento dos produtos Product1 e Product2 possuem inclinações semelhantes, devido aos seus aumentos crescentes. Para estabilizar essa crescente nos tempos, novos equipamentos para suas operações podem ser adicionados no processo de manufatura. Já na figura 40 os tempos de processamento do produto Product3 pertencem à uma faixa relativamente constante, logo o terceiro é mais estável que os dois primeiros, apresentando, inclusive, na segunda metade uma perceptível diminuição.

Por fim, as figuras 41 e 42 exibem o funcionamento do simulador em 3D. Inicialmente serão feitas as imagens em ambiente desktop e posteriormente em ambiente CAVE.

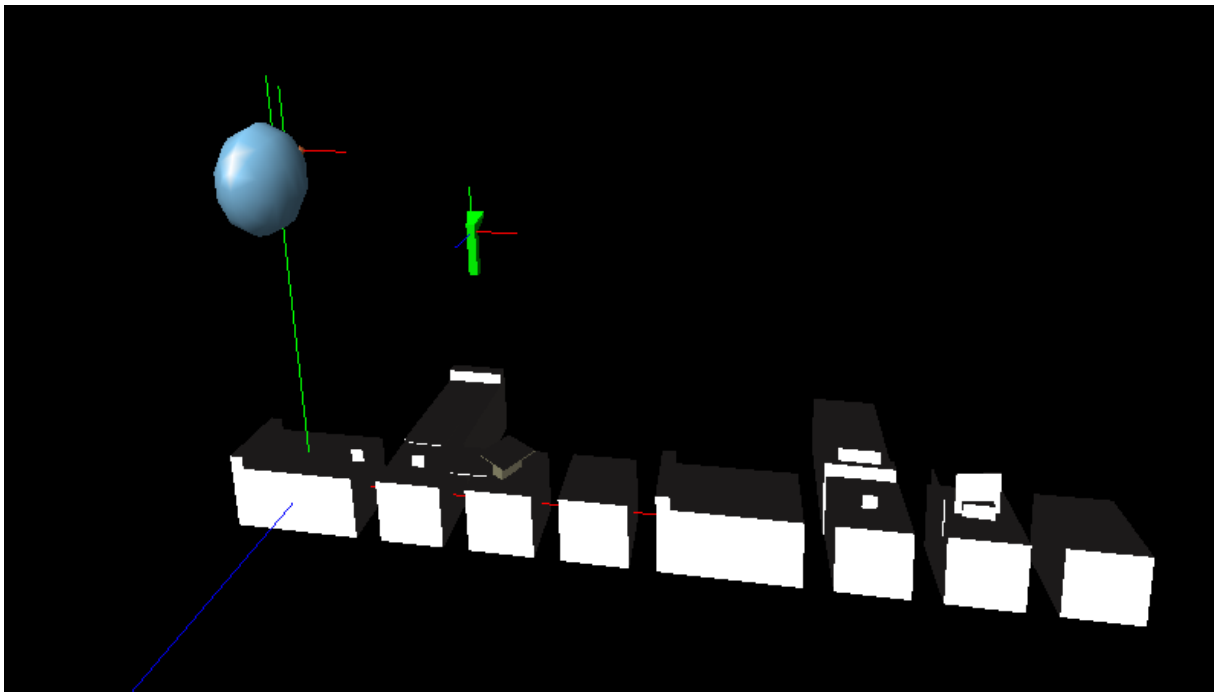

Figura 41 - Visualização do aplicativo do simulador em Desktop

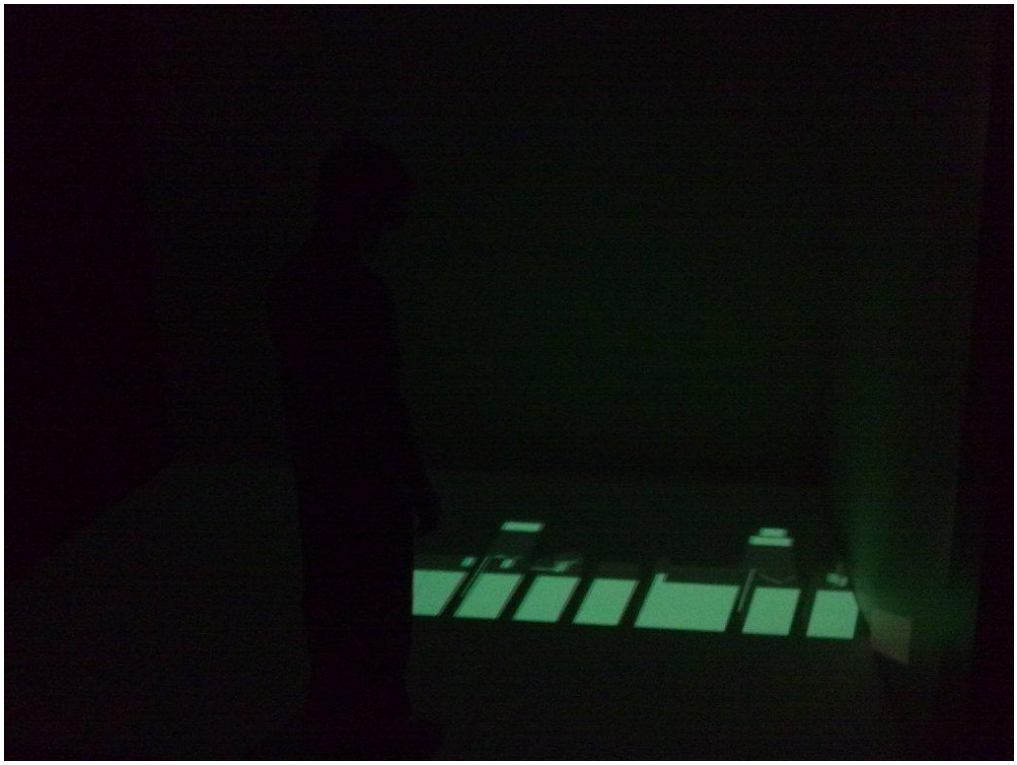

Figura 42 - Visualização do aplicativo do simulador em CAVE

Na figura 41, está representada uma visualização da aplicação do simulador em ambiente desktop. Nele, a cabeça do usuário e a *wand* que serve como entrada de dados para interação são simuladas por figuras semelhantes. Enquanto que na figura 42 está a mesma cena, só que visto em ambiente CAVE.

# 5. CONCLUSÃO

Conforme o objetivo principal, o desenvolvimento do núcleo de simulador de eventos discretos para sistemas de manufatura com visualização 3D, que pode ser conferida nas classes presentes no Apêndice A, mostrou-se eficaz, no sentido que em seu teste foram gerados dados que condizem com um sistema de manufatura apresentado anteriormente e foi possível a visualização gráfica em realidade virtual, ambos apresentados no Capítulo 4.

Para tanto, fez-se necessário que a lógica do simulador de eventos discretos para sistemas de manufatura e a estrutura dos elementos virtuais para visualização em ambiente tridimensional estivessem desenvolvidas com sucesso, além de estarem implementadas em C++, que foi a linguagem escolhida. Assim, todos os objetivos pretendidos foram alcançados.

A escolha do paradigma de três fases foi uma escolha fundamental para que o núcleo do simulador pudesse ser executado, devido as suas características como paralelismo de execução e divisão das atividades em dependentes ou não de tempo. Sua lógica teve uma boa combinação com a lógica de sistemas de manufatura. Outro fator importante para a eficácia do trabalho foi o uso do paradigma orientado a objetos, que tornou o núcleo do simulador bem próximo ao que realmente existe e mais fácil de ser interpretado.

A utilização da realidade virtual representa um avanço na visualização e na imersão da simulação, não só de sistemas de manufatura, mas em geral. A ênfase dada nesse trabalho no aspecto da visualização demonstrou também que, o uso da interação do usuário enriquecerá a experiência vivida pelo usuário.

Por fim, o núcleo do simulador mostrou-se preparado para o papel de auxílio à tomada de decisão, já que foi verificado que os parâmetros da aplicação podem ser alterados, resultando cenários diferentes, com seus respectivos estados.

## 5.1 Proposta para trabalhos futuros

Inicialmente, os próximos trabalhos que seguirem esse podem trabalhar com especificações da lógica de sistema de manufatura para diferentes arranjos físicos, como: distribuídos, celular, funcional, dentre outros. Já na parte de realidade virtual, poderia ser incluídos elementos de interação do usuário com o ambiente, tornando a simulação uma experiência muito mais didática. A estrutura do núcleo do simulador já está preparada para essa inserção, já que o VR Juggler já possui os módulos que têm essa função.

Observa-se, também, que em nenhum momento a eficiência da aplicação foi uma busca presente neste trabalho. Obviamente essa é uma característica que se espera ser encontrada em aplicativos de realidade virtual, mas que fica aqui mais como uma proposta de trabalhos futuros do que algo que serviu para abalizar o presente.

E finalmente, fica a proposta de validação do núcleo, ou o seu uso em outra estrutura, em ambientes reais de sistema de manufatura. Seja para o uso em projetos de novas plantas industriais ou para a mudança de política que a empresa queira por ventura realizar em suas instalações.

## 6. REFERÊNCIAS[2]

ARAUJO, R. B. D. **Especificação e Análise de um Sistema Distribuído de Realidade Virtual**. (Tese). Departamento de Engenharia Elétrica, Universidade de São Paulo, São Paulo, 160 p, 1996.

ASKIN, R. G. and STANDRIDGE, C. R. **Modeling and Analysis of Manufacturing Systems**. 1 ed.: Wiley. 461 p. 1993

BALAKSHIN; B. **Fundamentals of Manufacturing Engineering**. Moscow. 574 p. 1971

BANKS, J.; CARSON II, J. S. and NELSON, B. L. **Discrete-Event System Simulation**. 2nd ed. 548 p. 1996

BENJAAFAR, S.; HERAGU, S. S. and IRANI, S. A. **Next Generation Factory Layouts: Research Challenges and Recent Progress**. Interfaces, v.32, n.6, p.58-76, 2002

BIERBAUM, A.; JUST, C.; HARTLING, P.; MEINERT, K., BAKER, A. and CRUZ-NEIRA, C. **VR Juggler: A Virtual Platform for Virtual Reality Application Development**. Proceedings of the Virtual Reality 2001 Conference (VR'01): IEEE Computer Society 2001.

BISCHAK, D. P. and ROBERTS, S. D. **Object-oriented simulation**. Proceedings of the 23rd conference on Winter simulation. Phoenix, Arizona, United States: IEEE Computer Society 1991.

CHIEN, C.-F. and CHEN, C.-H. **Using genetic algorithms (GA) and a coloured timed Petri net (CTPN) for modelling the optimization-based schedule generator of a generic production scheduling system**. International Journal of Production Research, v.45, n.8, p.1763 - 1789, 2007

CRUZ-NEIRA, C.; SANDIN, D. J. and DEFANTI, T. A. **Surround-screen projection-based virtual reality: the design and implementation of the CAVE**. Proceedings of the 20th annual conference on Computer graphics and interactive techniques1993

CRUZ-NEIRA, C.; SANDIN, D. J.; DEFANTI, T. A.; KENYON, R. V. and HART, J. C. **The CAVE: audio visual experience automatic virtual environment**. Commun. ACM, v.35, n.6, p.64-72, 1992

DANILEVSKY, V. **Manufacturing Engineering**. Moscow: Mir. 550 p. 1973

DE TONI, A. and TONCHIA, S. **Manufacturing flexibility: a literature review**. International Journal of Production Research, v.36, n.6, p.1587 - 1617, 1998

[2] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

DEFANTI, T. A.; DAWE, G.; SANDIN, D. J.; SCHULZE, J. P.; OTTO, P.; GIRADO, J.; KUESTER, F.; SMARR, L. and RAO, R. **The StarCAVE, a third-generation CAVE and virtual reality OptIPortal**. Future Generation Computer Systems, v.25, n.2, p.169-178, 2009

DRIRA, A.; PIERREVAL, H. and HAJRI-GABOUJ, S. **Facility layout problems: A survey**. Annual Reviews in Control, v.31, n.2, p.255-267, 2007

FOWLER, M. and SCOTT, K. **UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos**. 2 ed. Porto Alegre 2000

FRANCE, R.; EVANS, A.; LANO, K. and RUMPE, B. **The UML as a formal modeling notation**. Computer Standards & Interfaces, v.19, n.7, p.325-334, 1998

JOINES, J. A. and ROBERTS, S. D. **Design of object-oriented simulations in C++**. Proceedings of the 28th conference on Winter simulation. Coronado, California, United States: IEEE Computer Society 1996.

JONES, K. C.; CYGNUS, M. W.; STORCH, R. L. and FARNSWORTH, K. D. **Virtual reality for manufacturing simulation**. Proceedings of the 25th conference on Winter simulation. Los Angeles, California, United States: ACM 1993.

KELSICK, J.; VANCE, J. M.; BUHR, L. and MOLLER, C. **Discrete Event Simulation Implemented in a Virtual Environment**. Journal of Mechanical Design, v.125, n.3, p.428-433, 2003

KIRNER, C. and SISCOUTTO, R. A. **Fundamentos da Realidade Virtual e Aumentada**. In: Realidade Virtual e Aumentada: Conceitos, Projeto e Aplicações Porto Alegre: Editora SBC – Sociedade Brasileira de Computação. p.2922007

LARMAN, C. **Utilizando UML e padrões - Uma introdução a análise e ao projeto orientados** Porto Alegre: Bookman. 696 p. 2007

LAW, A. and KELTON, D. **Simulation Modeling and Analysis**. 3 edition. McGraw-Hill Science/Engineering/Math. 800 p. 1999

LIN, L. and BEDWORTH, D. D. **A semi-generative approach to computer-aided process planning using group technology**. Comput. Ind. Eng., v.14, n.2, p.127-137, 1988

NANCE, R. E. **A history of discrete event simulation programming languages**. In: (Ed.). History of programming languages---II: ACM. p.369-4271996

O'REILLY, J. J. and LILEGDON, W. R. **Introduction to AweSim**. Proceedings of the 31st conference on Winter simulation: Simulation---a bridge to the future - Volume 1. Phoenix, Arizona, United States: ACM 1999.

OPENSG. **Tutorial/OpenSG1- OpenSG - Trac.** Disponível em:< http://opensg.vrsource.org/trac/wiki/Tutorial/OpenSG1/Basics>. Acesso em: 2 de novembro de 2009

PEGDEN, D. and HAM, I. **Simulation of Manufacturing Systems Using SIMAN**. CIRP Annals - Manufacturing Technology, v.31, n.1, p.365-369, 1982

PIDD, M. **Computer Simulation in Management Science**. 5th Edition. John Wiley & Sons. 328 p. 2004

PIMENTEL, K. and TEIXEIRA, K. **Virtual reality: through the new looking glass**. Windcrest/McGraw-Hill. 301 p. 1993

PORTO, A. J. V. **Desenvolvimento de um Método de Integração do Planejamento de Fabricação e do Planejamento e Controle da Produção, baseado na Flexibilidade do Processo de Fabricação**. (Tese). Departamento de Engenharia Mecânica, Universidade de São Paulo, São Carlos, 255 p, 1990.

SCHUEMIE, M.J.; VAN DER STRAATEN, P.; KRIJN, M. and VAN DER MAST, C.A.P.G. **Research on presence in virtual reality: A survey**. Cyberpsychology and Behavior 4 (2), pp. 183-201. 2001

SURESH KUMAR, N. and SRIDHARAN, R. **Simulation modelling and analysis of part and tool flow control decisions in a flexible manufacturing system**. Robotics and Computer-Integrated Manufacturing, v.25, n.4-5, 2009/10//, p.829-838, 2009

SUTHERLAND, I. E. **The Ultimate Display**. IFIP 1965

VINCE, J. **Essential Virtual Reality Fast: How to Understand the Techniques and Potential of Virtual Reality**. Springer-Verlag New York, Inc. 192 p. 1998

XU, Z.; ZHAO, Z. and BAINES, R. W. **Constructing virtual environments for manufacturing simulation**. International Journal of Production Research, v.38, n.17, p.4171 - 4191, 2000

# APÊNDICE A – CLASSES

## Clock.h

```
#ifndef _CLOCK_H_
#define _CLOCK_H_

#include <OpenSG/OSGGLUT.h>

class Clock {

public:

	Clock(){
		this->relativeClock = 0;
		this->scale = 1;

		stopped = false;

		absoluteClock = 0;
		auxiliarClock = 0;
	}
	Clock(long int clock, float scale) {
		this->relativeClock = clock;
		this->scale = scale;

		stopped = false;

		absoluteClock = 0;
		auxiliarClock = 0;

	}


	virtual long int getClock();
	virtual void printClock();
	virtual void setScale(float scale);
	virtual void stopClock();
	virtual void resumeClock();

private:

	bool stopped;

	long int absoluteClock;
	long int auxiliarClock;
	long int relativeClock;

	float scale;
};

#endif
```

Clock.cpp

```
#include <stdio.h>

#include "Clock.h"

long int Clock::getClock() {
	if (!stopped)
		absoluteClock = glutGet(GLUT_ELAPSED_TIME);


	return scale*this->absoluteClock + this->relativeClock;
}

void Clock::printClock() {
	printf ("Current date/time:");
	printf ("%d\n", scale*this->absoluteClock + this->relativeClock);
}

void Clock::setScale(float scale) {
	if (scale > 0) {
		this->scale = scale;
	}
}

void Clock::stopClock() {
	if (!stopped) {
		stopped = true;
		auxiliarClock = glutGet(GLUT_ELAPSED_TIME);
	}
}


void Clock::resumeClock() {
	if (stopped) {
		relativeClock = glutGet(GLUT_ELAPSED_TIME) - auxiliarClock;
		stopped = false;
	}
}
```

Equipment.h

```
#ifndef _EQUIPMENT_H_
#define _EQUIPMENT_H_

#include <queue>
#include <stack>
#include <cstring>

using namespace std;

#include "EventManager.h"
#include "Event.h"
#include "RotateEvent.h"
#include "TranslateEvent.h"

#include "Object.h"
#include "Product.h"
#include "StationaryPart.h"
#include "MovingPart.h"

class Equipment : public Object {

public:

	Equipment() : Object(){
		this->name = NULL;
		this->status = false;
		this->capacity = 1;
		this->lasttime = 0;
		this->products = new queue <Product *>();
		this->stationaryParts = new stack<StationaryPart*>();
		this->movingParts = new vector<MovingPart*>();
		this->em = new EventManager();
		this->createEvent = true;
	}

	Equipment(char *equipmentname,float x,float y,float z) : Object() {

		this->name = new char[strlen(equipmentname) + 1];
		strcpy (this->name, equipmentname);
		this->status = false;
		this->capacity = 1;
		this->lasttime = 0;
		this->products = new queue <Product *>();
		this->stationaryParts = new stack<StationaryPart*>();
		this->movingParts = new vector<MovingPart*>();
		this->em = new EventManager();
		this->createEvent = true;
		this->setPosition(x,y,z);
	}

	~Equipment() {
		delete this->products;
		delete this->stationaryParts;
		delete this->movingParts;
		delete this->name;
	}

	char *getName();
	void setAvaliable();
	void setBusy();
```

```
        bool isBusy();
        void setCapacity(int c);
        int getCapacity();
        void listProducts();
        void removeProduct();
        void addProduct(Product *p);
        Product* getNextProduct();
        void run(long int t);

protected:

        virtual void visualAction(long int t) = 0;
        void sendEvent(Event* e);
        void registerMovingPart(MovingPart* mp);
        void registerStationaryPart(StationaryPart* sp);

private:

        EventManager* em;
        stack<StationaryPart*>* stationaryParts;
        vector<MovingPart*>* movingParts;
        char *name;
        int size;
        bool status;
        int capacity;
        bool createEvent;
        long int lasttime;
        queue <Product *> *products;

};

#endif
```

Equipment.cpp

```
#include "Equipment.h"

char *Equipment::getName() {
	return this->name;
}

void Equipment::setAvaliable() {
	this->status = false;
}

void Equipment::setBusy() {
	this->status = true;
}

bool Equipment::isBusy() {
	return status;
}

void Equipment::setCapacity(int c) {
	this->capacity = c;

}

int Equipment::getCapacity() {
	return this->capacity;
}

void Equipment::listProducts() {
	for (int i=0; i < (int) this->products->size();i++) {

		cout << "Produto ";
		cout << this->products->front()->getName();

		Product *p = this->products->front();
		this->products->pop();
		this->products->push(p);
	}
}

void Equipment::removeProduct() {
	this->removeComponent(this->getNextProduct()->getSceneGraph());
	this->products->pop();
	if ((int) this->products->size() < this->capacity) {
		this->setAvaliable();
	}
}

void Equipment::addProduct(Product *p) {

	this->addComponent(p->getSceneGraph());
	this->products->push(p);
	if ((int) this->products->size() == this->capacity) {
		this->setBusy();
	}
}

Product* Equipment::getNextProduct() {
	return this->products->front();
```

```
}


void Equipment::run(long int t) {
	if (t > this->lasttime) {
		if (createEvent) {
			visualAction(t);
			this->createEvent = false;
		}else {
			bool finish = this->em->update(t);
			if (finish) {
				this->getNextProduct()->setCPosition(4);
				this->createEvent = true;
				this->lasttime = t;
			}
		}
	}
}

void Equipment::sendEvent(Event* e) {
	this->em->addEvent(e);
}

void Equipment::registerMovingPart(MovingPart* mp) {
	addComponent(mp->getSceneGraph());
	movingParts->push_back(mp);
}

void Equipment::registerStationaryPart(StationaryPart* sp) {
	addComponent(sp->getSceneGraph());
	stationaryParts->push(sp);
}
```

Event.h (VisualEvent)

```
#include "Equipment.h"

char *Equipment::getName() {
	return this->name;
}

void Equipment::setAvaliable() {
	this->status = false;
}

void Equipment::setBusy() {
	this->status = true;
}

bool Equipment::isBusy() {
	return status;
}

void Equipment::setCapacity(int c) {
	this->capacity = c;

}

int Equipment::getCapacity() {
	return this->capacity;
}

void Equipment::listProducts() {
	for (int i=0; i < (int) this->products->size();i++) {

		cout << "Produto ";
		cout << this->products->front()->getName();

		Product *p = this->products->front();
		this->products->pop();
		this->products->push(p);
	}
}

void Equipment::removeProduct() {
	this->removeComponent(this->getNextProduct()->getSceneGraph());
	this->products->pop();
	if ((int) this->products->size() < this->capacity) {
		this->setAvaliable();
	}
}

void Equipment::addProduct(Product *p) {

	this->addComponent(p->getSceneGraph());
	this->products->push(p);
	if ((int) this->products->size() == this->capacity) {
		this->setBusy();
	}
}

Product* Equipment::getNextProduct() {
	return this->products->front();
```

```
}


void Equipment::run(long int t) {
	if (t > this->lasttime) {
		if (createEvent) { //primeira iteracao / criar eventos visuais e
coloca-los na waiting
			visualAction(t);
			this->createEvent = false;
		}else {
			bool finish = this->em->update(t);
			if (finish) {
				this->getNextProduct()->setCPosition(4);
				this->createEvent = true;
				this->lasttime = t;
			}
		}
	}
}

void Equipment::sendEvent(Event* e) {
	this->em->addEvent(e);
}

void Equipment::registerMovingPart(MovingPart* mp) {
	addComponent(mp->getSceneGraph());
	movingParts->push_back(mp);
}

void Equipment::registerStationaryPart(StationaryPart* sp) {
	addComponent(sp->getSceneGraph());
	stationaryParts->push(sp);
}
```

Event.cpp (VisualEvent)

```
#include "Event.h"

void Event::run(long int t) {

	if (t > this->startTime) {
		if (t <= this->startTime + this->duration) {
			this->action(t);
		}else{
			this->abortEvent();
		}
	}
}

void Event::abortEvent() {
	this->state = 1;
}

void Event::setState(int s) {
	if (s < 2) {
		this->state = s;
	}
}

int Event::getStartTime() const{
	return this->startTime;
}

char Event::getState() {
	return this->state;
}

bool Event::isTerminated() {
	return (this->state == 1);
}

void Event::setDuration(long int t) {
	if (!this->durationFlag) {
		this->duration = t;
		this->durationFlag = true;
	}
}

float Event::getDuration() {
	return this->duration;
}

void Event::setStartTime(long int t) {
	this->startTime = t;
	this->startTimeFlag = true;
}
```

EventList.h

```
#ifndef _EVENT_LIST_H_
#define _EVENT_LIST_H_

#include <iostream>

using namespace std;

#include "Logic.h"
#include "Product.h"
#include "RegistryManager.h"

class EventList {

	public:

		EventList(Logic *l, vector<Product *> *productsList) {
			this->Logic = l;
			this->listOfProducts = productsList;
			this->Registrymanager = new RegistryManager();
		}

		~EventList() {
			this->Registrymanager->writeFile();
			delete this->Registrymanager;
		}

		virtual void executeAPhase();
		virtual void executeBPhase(long int t);
		virtual void executeCPhase(long int t);
		virtual RegistryManager *returnRegistryManager();

	private:

		int cposition;
		Logic *Logic;
		vector<Product *> *listOfProducts;
		RegistryManager *Registrymanager;

};

#endif
```

EventList.cpp

```
#include "EventList.h"

void EventList::executeAPhase(){ }

void EventList::executeBPhase(long int t) {
	for (int i = (int) this->listOfProducts->size() - 1; i >= 0 ;i--) {
		if (this->listOfProducts->at(i)->getCPosition() == 3) {
			this->Logic->returnFactory()->returnManufacturingProcess(this-
>listOfProducts->at(i)->getName())->returnOperationByIndex(this->listOfProducts-
>at(i)->getPosition())->returnEquipment()->run(t);

			this->Registrymanager->addRegistry('B',this->listOfProducts-
>at(i)->getCPosition(),(listOfProducts->at(i)->getName()),listOfProducts->at(i)-
>getID(),listOfProducts->at(i)->getPosition(),this->Logic->returnFactory()-
>returnManufacturingProcess(this->listOfProducts->at(i)->getName())-
>returnOperationByIndex(this->listOfProducts->at(i)->getPosition())-
>returnEquipment()->getName(),t,"Executando");
		}
	}
}

void EventList::executeCPhase(long int t) {
	for (int i = (int) this->listOfProducts->size() - 1; i >= 0 ;i--) {
		if ((this->listOfProducts->at(i)->getCPosition()) == 1) { //C1
			if (!this->Logic->checkNextEquipment(this->listOfProducts-
>at(i))) {
				this->Logic->sendProductToNextOperation(this-
>listOfProducts->at(i));
				this->listOfProducts->at(i)->setCPosition(2);
			}else if (this->Logic->checkNextEquipment(this->listOfProducts-
>at(i)) ) {
				if (listOfProducts->at(i)->getPosition() > -1)
					this->Registrymanager->addRegistry('C',this-
>listOfProducts->at(i)->getCPosition(),(listOfProducts->at(i)-
>getName()),listOfProducts->at(i)->getID(),listOfProducts->at(i)-
>getPosition(),this->Logic->returnFactory()->returnManufacturingProcess(this-
>listOfProducts->at(i)->getName())->returnOperationByIndex(this->listOfProducts-
>at(i)->getPosition())->returnEquipment()->getName(),t,"Esperando");

			} else if (this->Logic->checkNextEquipment(this->listOfProducts-
>at(i)) == 2) { //Se o produto estiver na ultima operacao
				this->listOfProducts->at(i)->setCPosition(4); //Camada C4
			}

		}else if ((this->listOfProducts->at(i)->getCPosition()) == 2) { //C2
			//cout << endl << "Executando C2 do Produto " << this-
>listOfProducts->at(i)->getName() << " com ID " << this->listOfProducts->at(i)-
>getID() << endl;
			if (this->Logic->productIsTheLast(this->listOfProducts->at(i)))
{
				this->listOfProducts->at(i)->setCPosition(3);
			}

		}else if (this->listOfProducts->at(i)->getCPosition() == 4) { //C3
			if (!this->Logic->checkProduct(this->listOfProducts->at(i))) {
				this->listOfProducts->at(i)->setCPosition(1);
			}else{
```

```
                        cout << "Produto " << this->listOfProducts->at(i)-
>getName() << " com ID " << this->listOfProducts->at(i)->getID() << " finalizado"
<< endl;

                        this->Registrymanager->addRegistry('C',this-
>listOfProducts->at(i)->getCPosition(),(listOfProducts->at(i)-
>getName()),listOfProducts->at(i)->getID(),listOfProducts->at(i)-
>getPosition(),this->Logic->returnFactory()->returnManufacturingProcess(this-
>listOfProducts->at(i)->getName())->returnOperationByIndex(this->listOfProducts-
>at(i)->getPosition())->returnEquipment()->getName(),t,"Finalizado");

                        this->Logic->killProduct(listOfProducts->at(i));
                        this->listOfProducts->erase(this->listOfProducts-
>begin()+i);
                    }
                }
        }
}

RegistryManager *EventList::returnRegistryManager() {
        return this->Registrymanager;
}
```

EventManager.h

```
#ifndef _EVENT_MANAGER_H_
#define _EVENT_MANAGER_H_

#include <queue>
#include <vector>

#include "Event.h"
#include "EventCompare.h"


using namespace std;

class EventManager {

public:

	EventManager() {
		this->runningEvents = new vector<Event*>();
		this->waitingEvents = new priority_queue<Event*, vector<Event*>,
EventCompare>();

		this->currentTime = -1;

		currentTime = 0;
	}

	~EventManager() {
		delete this->runningEvents;
		delete this->waitingEvents;
	}

	virtual void addEvent(Event *e);
	virtual bool update(long int time);
	virtual int getTime();
	virtual bool setTime(long int time);
	virtual void incrementTime();
	virtual void incrementTime(long int time);

private:

	void promoteWaitingEvents(long int time);
	void executeRunningEvents(long int time);
	int currentTime;
	vector<Event*>* runningEvents;
	priority_queue<Event*, vector<Event*>, EventCompare>* waitingEvents;
};

#endif
```

## EventManager.cpp

```
#include "EventManager.h"

void EventManager::addEvent(Event* e) {
	if (e->getStartTime() > currentTime) {
		this->waitingEvents->push(e);
	}
}

int EventManager::getTime() {
	return this->currentTime;
}

void EventManager::incrementTime() {
	this->currentTime++;
}

void EventManager::incrementTime(long int time) {
	this->currentTime += time;
}

bool EventManager::setTime(long int time) {
	bool set;

	if (time > this->currentTime) {
		this->currentTime = time;
		set = true;
	} else {
		set = false;
	}

	return set;
}

bool EventManager::update(long int time) {
	bool set;
	set = this->setTime(time);
	if (set) {
		this->promoteWaitingEvents(time);
		this->executeRunningEvents(time);
	}
	return (!this->runningEvents->size() && !this->waitingEvents->size());
}

void EventManager::promoteWaitingEvents(long int time) {

	bool finished = false;
	Event* e;

	while (!finished) {
		if ((!this->waitingEvents->empty()) && (this->waitingEvents->top()-
>getStartTime() <= this->currentTime)) {
			e = this->waitingEvents->top();
			this->waitingEvents->pop();
			this->runningEvents->push_back(e);
		}else{
			finished = true;
		}
	}
}

void EventManager::executeRunningEvents(long int time) {
```

```
	int i;

	i = 0;
	while (i < (int) runningEvents->size()) {

		this->runningEvents->at(i)->run(time);

		if (runningEvents->at(i)->isTerminated()) {
			delete this->runningEvents->at(i);
			this->runningEvents->at(i) = this->runningEvents->at(((int)
this->runningEvents->size()) - 1);
			this->runningEvents->pop_back();
		}else{
			i++;
		}
	}

}
```

## Factory.h

```
#ifndef _FACTORY_H_
#define _FACTORY_H_

#include <iostream>
#include <vector>

using namespace std;

#include "Product.h"
#include "ManufacturingProcess.h"
#include "Equipment.h"
#include "ManufacturingEquipment.h"
#include "TransportEquipment.h"
#include "StoreEquipment.h"
#include "InspectionEquipment.h"
#include "RegistryManager.h"


class Factory {

	public:
		Factory() {
			this->name = NULL;
			this->listOfProducts = new vector<Product *>();
			this->listOfManufacturingProcess = new vector
<ManufacturingProcess *> ();
			this->listOfEquipments = new vector<Equipment *>();
			this->registrymanager = new RegistryManager ();
		}

		Factory (char *factoryname) {
			this->name = (char *) malloc((strlen(factoryname) + 1) *
sizeof(char));
			strcpy (this->name, factoryname);
			this->listOfProducts = new vector<Product *>();
			this->listOfManufacturingProcess = new
vector<ManufacturingProcess *>();
			this->listOfEquipments = new vector<Equipment *>();
			this->registrymanager = new RegistryManager ();
		}

		~Factory() {
			delete[] this->name;
			delete this->listOfProducts;
			delete this->listOfManufacturingProcess;
			delete this->listOfEquipments;
			delete this->registrymanager;
		}

		void addProduct(char *productname,float we, float he, float de);
		void removeProduct(char *productname);
		Product *returnProductByName(char *productname);
		Product *returnProductByID (char *productname,int n);
		void listProducts();
		char *getName();
		void setName(char *factoryname);
		void addManufacturingEquipment (char *equipmentname,int ri,int
re,float x,float y,float z);
		void addTransportEquipment (char *equipmentname,float x,float y,float
z);
		void addStoreEquipment (char *equipmentname,float x,float y,float z);
```

```
            void addInspectionEquipment (char *equipmentname,float x,float y,float
z);
            void removeEquipment (char *equipmentname);
            Equipment *returnEquipment (char *equipmentname);
            Equipment *returnEquipmentByIndex (int n);
            int equipmentSize();
            void addManufacturingProcess (char *processname);
            void removeManufacturingProcess (char *processname);
            ManufacturingProcess *returnManufacturingProcess (char *processname);
            vector<Product *> *returnProducts();
            int totalProducts();
            int totalEquipments();

      private:

            char *name;
            vector <Product *> *listOfProducts;
            vector <ManufacturingProcess *> *listOfManufacturingProcess;
            vector <Equipment *> *listOfEquipments;
            Equipment *getProductFeature (Product *pr);
            RegistryManager *registrymanager;

};

#endif
```

## Factory.cpp

```
#include "Factory.h"

void Factory::addProduct(char *productname, float we, float he, float de) {
	Product *p = new Product(productname,we, he, de, 0, 0, 0);
	this->returnManufacturingProcess(p->getName())->addProduct(p);
	this->listOfProducts->push_back(p);
}

void Factory::removeProduct(char *productname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfProducts->size())) {
		if (!strcmp(listOfProducts->at(i)->getName(), productname)) {
			listOfProducts->erase(listOfProducts->begin()+i);
		} else {
			i++;
		}
	}
	if (!found) { }
}


Product *Factory::returnProductByName(char *productname) {
	for (int i=0; i < (int) this->listOfProducts->size(); i++)
		if (!strcmp(this->listOfProducts->at(i)->getName(),productname)) {
			return this->listOfProducts->at(i);
		} else if (i == ((int) this->listOfProducts->size()-1)) {
			return NULL;
		}
}

Product *Factory::returnProductByID (char *productname,int n) {
	for (int i=0; i < (int) this->listOfProducts->size(); i++)
		if ((!strcmp(this->listOfProducts->at(i)->getName(),productname)) &&
(n == this->listOfProducts->at(i)->getID())) {
			return this->listOfProducts->at(i);
		} else if (i == ((int) listOfProducts->size()-1)) {
			return NULL;
		}
}

void Factory::listProducts() {
	if (this->listOfProducts->size() == 0) {
	} else {
		for (int i=0;i < (int) listOfProducts->size();i++) {
			cout << "Produto: " << listOfProducts->at(i)->getName() << ", na
posicao " << i+1 << endl;
		}
	}
}

char *Factory::getName() {
	return this->name;
}

void Factory::setName (char *factoryname) {
	this->name = (char *) malloc((strlen(factoryname) + 1) * sizeof(char));
	strcpy (this->name, factoryname);
}
```

```
void Factory::addManufacturingEquipment(char *equipmentname,int ri,int re,float
x,float y,float z) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfEquipments->size())) {
		if (!strcmp(this->listOfEquipments->at(i)->getName(), equipmentname))
{
				found = true;
		} else {
			i++;
		}
	}
	if (!found) {
		Equipment *e = new ManufacturingEquipment(equipmentname,ri,re,x,y,z);
		this->listOfEquipments->push_back(e);
	}
}

void Factory::addTransportEquipment(char *equipmentname,float x,float y,float z) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfEquipments->size())) {
		if (!strcmp(this->listOfEquipments->at(i)->getName(), equipmentname))
{
				found = true;
		} else {
			i++;
		}
	}
	if (!found) {
		Equipment *e = new TransportEquipment(equipmentname,x,y,z);
		this->listOfEquipments->push_back(e);
	}
}

void Factory::addStoreEquipment(char *equipmentname,float x,float y,float z) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfEquipments->size())) {
		if (!strcmp(this->listOfEquipments->at(i)->getName(), equipmentname))
{
				found = true;
		} else {
			i++;
		}
	}
	if (!found) {
		Equipment *e = new StoreEquipment(equipmentname,x,y,z);
		this->listOfEquipments->push_back(e);
	}
}

void Factory::addInspectionEquipment(char *equipmentname,float x,float y,float z) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfEquipments->size())) {
		if (!strcmp(this->listOfEquipments->at(i)->getName(), equipmentname))
{
				found = true;
		} else {
			i++;
		}
```

```
	}
	if (!found) {
		Equipment *e = new InspectionEquipment(equipmentname,x,y,z);
		this->listOfEquipments->push_back(e);
	}
}


void Factory::removeEquipment(char *equipmentname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfEquipments->size())) {
		if (!strcmp(this->listOfEquipments->at(i)->getName(), equipmentname))
{
			this->listOfEquipments->erase(this->listOfEquipments-
>begin()+i);
		} else {
			i++;
		}
	}
	if (!found) { }
}

Equipment *Factory::returnEquipment (char *equipmentname) {
	for (int i=0; i < (int) this->listOfEquipments->size(); i++)
		if (!strcmp(this->listOfEquipments->at(i)->getName(),equipmentname)) {
			return this->listOfEquipments->at(i);
		} else if (i == ((int) this->listOfEquipments->size()-1)) {
			return NULL;
		}
}

Equipment *Factory::returnEquipmentByIndex(int n) {
	return this->listOfEquipments->at(n);
}

int Factory::equipmentSize() {
	return (int) this->listOfEquipments->size();
}

void Factory::addManufacturingProcess (char *processname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfManufacturingProcess->size())) {
		if (!strcmp(this->listOfManufacturingProcess->at(i)->getName(),
processname)) {
				found = true;
		} else {
			i++;
		}
	}
	if (!found) {
		ManufacturingProcess *m = new ManufacturingProcess(processname);
		this->listOfManufacturingProcess->push_back(m);
	}
}

void Factory::removeManufacturingProcess (char *processname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i< (int) this->listOfManufacturingProcess->size())) {
		if (!strcmp(this->listOfManufacturingProcess->at(i)->getName(),
processname)) {
			this->listOfManufacturingProcess->erase(this-
>listOfManufacturingProcess->begin()+i);
			found = true;
```

```
			} else {
				i++;
			}
		}
		if (!found) { }
}

ManufacturingProcess *Factory::returnManufacturingProcess (char *processname){
		for (int i=0; i < (int) this->listOfManufacturingProcess->size(); i++)
			if (!strcmp(this->listOfManufacturingProcess->at(i)-
>getName(),processname)) {
				return this->listOfManufacturingProcess->at(i);
			}
			return NULL;

}

vector<Product *> *Factory::returnProducts() {
		return listOfProducts;
}

int Factory::totalProducts() {
		return (int) this->listOfProducts->size();
}

int Factory::totalEquipments() {
		return (int) this->listOfEquipments->size();
}
```

## GeoObject.h

```
#ifndef _GEO_OBJECT_H_
#define _GEO_OBJECT_H_

#include <OpenSG/OSGSimpleGeometry.h>
#include "Object.h"

class GeoObject : public Object {

public:

	GeoObject(char name[]) : Object() {
		this->geoNode = SceneFileHandler::the().read(name);
		this->addComponent(this->geoNode);
	}

	GeoObject() : Object()	{ }

	GeoObject(float x, float y, float z, char name[]) : Object(x,y,z){
		this->geoNode = SceneFileHandler::the().read(name);
		this->addComponent(this->geoNode);
	}

	GeoObject(float width, float height, float depth, float x, float y, float z)
: Object(x,y,z){
		this->geoNode = makeBox(width,height,depth,1,1,1);
		this->addComponent(this->geoNode);
	}

	~GeoObject() { }


protected:

	NodePtr geoNode;

private:


};

#endif
```

InspectionEquipment.h

```
#ifndef _INSPECTION_EQUIPMENT_H_
#define _INSPECTION_EQUIPMENT_H_

#include "Equipment.h"

class InspectionEquipment : public Equipment {

	public:

		InspectionEquipment() : Equipment() {}
		InspectionEquipment(char *equipmentName,float x,float y,float z) :
Equipment(equipmentName,x,y,z) {
			StationaryPart *sp = new StationaryPart(.5,.5,.5,0,0,0);
			this->registerStationaryPart(sp);
		}
		~InspectionEquipment() { }

	protected:
		virtual void visualAction(long int t) {
			TranslateEvent *teProduct = new TranslateEvent (this-
>getNextProduct());
			teProduct->setTime(t,10);
			teProduct->setInitialValues(0,0.3,0.0);
			teProduct->setFinalValues(0,0.3,0.0);
			sendEvent(teProduct);

		}
};

#endif
```

## Logic.h

```
#ifndef _LOGIC_H_
#define _LOGIC_H_

#include <vector>

#include "Factory.h"

class Logic {
	public:
		Logic () {
			this->name = NULL;
			this->factory = NULL;
		}

		Logic (char *Logicname)  {
			this->name = (char *) malloc((strlen(Logicname) + 1) *
sizeof(char));
			strcpy (this->name, Logicname);
			this->factory = NULL;
		}

		Logic (Factory *fa)  {
			this->setFactory(fa);
		}

		~Logic() {
			delete[] this->name;
		}

		int checkNextEquipment(Product *p);
		bool checkCurrentEquipment(Product *p);
		bool checkProduct(Product *p);
		void sendProductToNextOperation (Product *p);
		void setFactory(Factory *f);
		void removeFactory();
		char *getName();
		void setName(char *Logicname);
		Factory *returnFactory();
		bool productIsTheLast (Product *p);
		void killProduct(Product *p);

	private:
		char *name;
		Factory *factory;

};

#endif
```

# Logic.cpp

```
#include "Logic.h"

int Logic::checkNextEquipment(Product *p) {
	if (((p->getPosition())+1) < this->factory->returnManufacturingProcess(p-
>getName())->getTotalOperations()) {
		return factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex((p->getPosition()+1))->returnEquipment()->isBusy();
	} else {
		return 2;
	}
}

bool Logic::checkCurrentEquipment(Product *p) {
	return factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex(p->getPosition())->returnEquipment()->isBusy();
}
bool Logic::checkProduct(Product *p) {
	if ((p->getPosition()+1) == (p->getTotalOperations())) {
		p->setStatus(true);
	}
	return p->getStatus();
}

void Logic::sendProductToNextOperation(Product *p) {
	if (p->getPosition() > -1) {
		this->factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex((p->getPosition()))->removeProduct();
	}
	p->incrementPosition();
	this->factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex(p->getPosition())->addProduct(p);
}

void Logic::setFactory (Factory *f) {
	this->factory = f;

}

void Logic::removeFactory() {
	this->factory = NULL;

}

char *Logic::getName() {
	return this->name;
}

void Logic::setName (char *Logicname) {
	this->name = (char *) malloc((strlen(Logicname) + 1) * sizeof(char));
	strcpy (this->name, Logicname);
}

Factory *Logic::returnFactory() {
	return this->factory;
}

bool Logic::productIsTheLast (Product *p) {
```

```
	if ( p == (this->factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex(p->getPosition())->returnEquipment()->getNextProduct())) {
		return true;
	}else{
		return false;
	}
}

void Logic::killProduct(Product *p) {
	this->factory->returnManufacturingProcess(p->getName())-
>returnOperationByIndex(p->getPosition())->returnEquipment()->removeProduct();
}
```

ManufacturingEquipment.h

```
#ifndef _MANUFACTURING_EQUIPMENT_H_
#define _MANUFACTURING_EQUIPMENT_H_

#include "Equipment.h"
#include "MovingPart.h"

class ManufacturingEquipment : public Equipment {

	public:

		ManufacturingEquipment(char *equipmentName,int ri,int re,float x,float
y,float z)	: Equipment(equipmentName,x,y,z) {
			StationaryPart *sp = new
StationaryPart(0.5,0.5,0.5,0.0,0.0,0.0);
			sp->setPosition(0,0,0);
			this->registerStationaryPart(sp);
			this->m = new MovingPart(0.1,0.1,0.2,0.0,0.0,0.0);
			this->m->setPosition(0.0,0.3,0.0);
			this->registerMovingPart(this->m);
			this->rinit = ri;
			this->rend = re;
		}

	protected:

		virtual void visualAction(long int t) {
			int ra = rand() % this->rinit + this->rend;

			RotateEvent *r = new RotateEvent (this->m);
			r->setVector(0.0,1.0,0.0);
			r->setInitialValue(0);
			r->setRotationTax(0.5);
			r->setTime(t,ra);
			sendEvent(r);

			this->getNextProduct()->setPosition(0,0.40,0.0);
			RotateEvent *r1 = new RotateEvent (this->getNextProduct());
			r1->setVector(0.0,1.0,0.0);
			r1->setInitialValue(0);
			r1->setRotationTax(0.5);
			r1->setTime(t,ra);
			sendEvent(r1);
		}

	private:

		MovingPart *m;
		int rinit;
		int rend;

};

#endif
```

## ManufacturingProcess.h

```
#ifndef _MANUFACTURINGPROCESS_H_
#define _MANUFACTURINGPROCESS_H_

#include <iostream>
#include <vector>

#include "Equipment.h"
#include "Product.h"
#include "Operation.h"

using namespace std;

class ManufacturingProcess {

	public:

		ManufacturingProcess () {
			this->totalproducts = 0;
			this->name = NULL;
			this->listOfProducts = new vector<Product *>();
			this->listOfOperations = new vector<Operation *>();
		}

		ManufacturingProcess (char *processname) {
			this->totalproducts = 0;
			this->name = (char *) malloc ((strlen(processname) + 1) *
sizeof(char));
			strcpy(this->name, processname);
			this->listOfProducts = new vector<Product *>();
			this->listOfOperations = new vector<Operation *>();
		}

		~ManufacturingProcess()   {
			delete[] this->name;
			delete this->listOfProducts;
			delete this->listOfOperations;
		}


		void addProduct(Product *p);
		void removeProduct (char *productname);
		void clearOperationList();
		void addOperation (char *operationname);
		char *getName();
		Operation *returnOperationByName (char *operationname);
		Operation *returnOperationByIndex (int n);
		bool getFirstOperation();
		void changeFirstOperation();
		int sizeOfOperations();
		int getTotalProducts();
		int getCurrentProducts();
		int getTotalOperations();

	private:
		int totalproducts;
		vector<Product *> *listOfProducts;
		char *name;
		vector<Operation *> *listOfOperations;
};

#endif
```

# ManufacturingProcess.cpp

```
#include "ManufacturingProcess.h"

void ManufacturingProcess::addProduct(Product *p) {
	if (!strcmp(p->getName(),this->name)) {
		p->setTotalOperations((int) this->listOfOperations->size());
		this->listOfProducts->push_back(p);
		this->totalproducts++;
		p->setID(this->totalproducts);
	} else { }
}

void ManufacturingProcess::removeProduct (char *productname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfProducts->size())) {
		if (!strcmp(this->listOfProducts->at(i)->getName(), productname)) {
			this->listOfProducts->erase(this->listOfProducts->begin()+i);
			found = true;
		} else {
			i++;
		}
	}
	if (!found) { }
}

void ManufacturingProcess::clearOperationList() {
	this->listOfOperations->clear();
}

void ManufacturingProcess::addOperation(char *operationname) {
	bool found = false;
	int i = 0;
	while ((!found) && (i<(int) this->listOfOperations->size())) {
		if (!strcmp(this->listOfOperations->at(i)->getName(),operationname)) {
				found = true;
		} else {
			i++;
		}
	}
	if (!found) {
		Operation *o = new Operation(operationname);
		this->listOfOperations->push_back(o);
	}
}

char *ManufacturingProcess::getName() {
	return this->name;
}

Operation *ManufacturingProcess::returnOperationByName (char *operationname) {
	for (int i=0;i < (int) listOfOperations->size();i++) {
		if (!strcmp(listOfOperations->at(i)->getName(),operationname)) {
			return listOfOperations->at(i);
		} else if (i == listOfOperations->size()-1) {
			return NULL;
		}
	}
}
```

```
Operation *ManufacturingProcess::returnOperationByIndex(int n) {
	return this->listOfOperations->at(n);
}

int ManufacturingProcess::sizeOfOperations() {
	return (int) this->listOfOperations->size();
}

int ManufacturingProcess::getTotalProducts() {
	return this->totalproducts;
}

int ManufacturingProcess::getCurrentProducts() {
	return (int) this->listOfProducts->size();
}

int ManufacturingProcess::getTotalOperations() {
	return (int) this->listOfOperations->size();
}
```

# MovingObject.h

```
#ifndef _MOVING_OBJECT_H_
#define _MOVING_OBJECT_H_

#include "GeoObject.h"

class MovingObject : public GeoObject {

	public:
		MovingObject(char *name) : GeoObject(name) {
			this->state = 0;
		}

		MovingObject(float x, float y, float z, char name[]) : GeoObject(x, y,
z, name) {
			this->state = 0;
		}

		MovingObject() : GeoObject() {
			this->state = 0;
		}

		MovingObject(float width, float height, float depth, float x, float y,
float z) : GeoObject(width,height,depth,x,y,z) {
			this->state = 0;
		}

		~MovingObject() { }
		virtual bool isBusy();
		virtual bool isFree();
		virtual void setBusy();
		virtual void setFree();

	private:

		int state;

};

#endif
```

MovingObject.cpp

```
#include "MovingObject.h"

bool MovingObject::isBusy() {
	return (this->state != 0);
}

bool MovingObject::isFree() {
	return (this->state == 0);
}

void MovingObject::setBusy() {
	this->state++;
}

void MovingObject::setFree() {
	if (this->isBusy()) {
		this->state--;
	}
}
```

MovingPart.h

```
#ifndef _MOVING_PART_H_
#define _MOVING_PART_H_

#include "MovingObject.h"

class MovingPart : public MovingObject {

       public:

              MovingPart(char name[]) : MovingObject(name){ }
              MovingPart(float width, float height, float depth, float x, float y,
float z) : MovingObject(width,height,depth,x,y,z) {}
              ~MovingPart() { }

};

#endif
```

## Object.h

```
#ifndef _OBJECT_H_
#define _OBJECT_H_

#include <OpenSG/OSGTransform.h>
#include <OpenSG/OSGConfig.h>
#include <OpenSG/OSGGLUT.h>
//#include <OpenSG/OSGSimpleGeometry.h>
//#include <OpenSG/OSGGLUTWindow.h>
#include <OpenSG/OSGGL.h>
#include <OpenSG/OSGNode.h>
#include <OpenSG/OSGPointLight.h>
#include <OpenSG/OSGSceneFileHandler.h>
#include <iostream>

using namespace std;

OSG_USING_NAMESPACE

class Object {

public:

	Object() {
		this->m.setIdentity();
		this->vPosition.setValues(0,0,0);
		this->transNode = Node::create();
		this->transCore = Transform::create();


		beginEditCP (this->transCore, Transform::MatrixFieldMask);
			this->m.setTranslate(this->vPosition);
			this->transCore->setMatrix(this->m);
		endEditCP (this->transCore, Transform::MatrixFieldMask);

		beginEditCP (this->transNode, Node::CoreFieldMask);
			this->transNode->setCore(this->transCore);
		endEditCP (this->transNode, Node::CoreFieldMask);

	}
	Object(float x, float y, float z) {

		this->m.setIdentity();
		this->vPosition.setValues(x,y,z);
		this->transNode = Node::create();
		this->transCore = Transform::create();


		beginEditCP (this->transCore, Transform::MatrixFieldMask);
			this->m.setTranslate(this->vPosition);
			this->transCore->setMatrix(this->m);
		endEditCP (this->transCore, Transform::MatrixFieldMask);

		beginEditCP (this->transNode, Node::CoreFieldMask);
			this->transNode->setCore(transCore);
		endEditCP (this->transNode, Node::CoreFieldMask);
	}
	~Object() {
		this->transNode = NullFC;
		this->transCore = NullFC;
	}
```

```
	virtual void setPosition(float x, float y, float z);
	virtual void getPosition(float& x, float& y, float& z);
	virtual void setRotation(float x, float y, float z, float d);
	virtual NodePtr getSceneGraph();
	virtual void addComponent(NodePtr n);
	virtual void removeComponent(NodePtr n);
protected:

	NodePtr transNode;
	TransformPtr transCore;

private:

	Quaternion qRotation;
	Matrix m;
	Vec3f vPosition;

};


#endif
```

## Object.cpp

```
#include "Object.h"

void Object::setPosition(float x, float y, float z = 0) {

	this->vPosition.setValues(x,y,z);

	beginEditCP (this->transCore, Transform::MatrixFieldMask);
		this->m.setTranslate(vPosition);
		this->transCore->setMatrix(m);
	endEditCP (this->transCore, Transform::MatrixFieldMask);
}

void Object::getPosition(float& x, float& y, float& z) {

	x = this->vPosition[0];
	y = this->vPosition[1];
	z = this->vPosition[2];
}

void Object::setRotation(float aX, float aY, float aZ, float d) {

	Matrix m1;

	this->qRotation = Quaternion(Vec3f(aX,aY,aZ),d);

	m1.setRotate(this->qRotation);

	this->m.mult(m1);

	beginEditCP(this->transCore, Transform::MatrixFieldMask);
		this->transCore->setMatrix(m);
	endEditCP(this->transCore, Transform::MatrixFieldMask);
}

NodePtr Object::getSceneGraph()
{
	return this->transNode;
}

void Object::addComponent(NodePtr n)
{

	beginEditCP (this->transNode);
		//addRefCP(n);
		this->transNode->addChild(n);
		//subRefCP(n);
	endEditCP (this->transNode);
}

void Object::removeComponent(NodePtr n)
{

	beginEditCP (this->transNode);
		addRefCP(n);
		this->transNode->subChild(n);
	endEditCP (this->transNode);
}
```

# Operation.h

```
#ifndef _OPERATION_H_
#define _OPERATION_H_

#include <iostream>

using namespace std;

#include "Equipment.h"


class Operation{

      public:
                Operation()  {
                      this->name = NULL;
                      this->equipment = NULL;
                }

                Operation(char *operationname) {
                      this->name = (char *) malloc((strlen(operationname) + 1) *
sizeof(char));
                      strcpy (this->name, operationname);
                      this->equipment = NULL;
                }

                ~Operation() {
                      delete this->equipment;
                      delete[] this->name;
                }

                void setEquipment(Equipment *e);
                Equipment *returnEquipment();
                void removeEquipament();
                void setName(char *operationname);
                char *getName();
                void changeEquipmentStatus();
                void addProduct(Product *p);
                void removeProduct();

      private:
                Equipment *equipment;
                char *name;

};

#endif
```

## Operation.cpp

```
#include"Operation.h"

void Operation::setEquipment(Equipment *e) {
	this->equipment = e;
}

Equipment *Operation::returnEquipment() {
	return equipment;
}

void Operation::removeEquipament() {
	equipment = NULL;
}

void Operation::setName(char *operationname) {
	name = (char *) malloc((strlen(operationname) + 1) * sizeof(char));
	strcpy (name, operationname);
}

char *Operation::getName() {
	return this->name;
}

void Operation::changeEquipmentStatus() {
	if (this->equipment->isBusy() == 0) {
		this->equipment->setBusy();
	} else {
		this->equipment->setAvaliable();
	}
}

void Operation::addProduct(Product *p) {
	if (this->equipment == NULL) {
	} else {
		this->equipment->addProduct(p);
	}
}

void Operation::removeProduct() {
	this->equipment->removeProduct();
}
```

# Product.h

```
#ifndef _PRODUCT_H_
#define _PRODUCT_H_

#include <iostream>
#include <vector>

using namespace std;

#include "MovingPart.h"

class Product : public MovingPart {

      public:
                Product() : MovingPart(NULL){
                     this->name = NULL;
                     this->totaloperations = 0;
                     this->position = -1;
                     this->status = false;
                     this->cposition = 1;
                }

                Product(char productname[]) : MovingPart(productname) {
                       this->name = (char *) malloc((strlen(productname) + 1) *
sizeof(char));
                       strcpy (this->name, productname);
                       this->totaloperations = 0;
                       this->position = -1;
                       this->status = false;
                       this->cposition = 1;
                }

                Product(char productname[], float width, float height, float depth,
float x, float y, float z) : MovingPart(width,height,depth,x,y,z) {
                       this->name = (char *) malloc((strlen(productname) + 1) *
sizeof(char));
                       strcpy (this->name, productname);
                       this->totaloperations = 0;
                       this->position = -1;
                       this->status = false;
                       this->cposition = 1;
                }

                ~Product() {
                       delete[] this->name;
                }

                char *getName();
                void incrementPosition();
                void decrementPosition();
                int getPosition();
                void setStatus(bool b);
                bool getStatus();
                void setTotalOperations(int n);
                int getTotalOperations();
                void setCPosition(int n);
                int getCPosition();
                void setID(int n);
                int getID();
```

```
      private:
                char *name;
                int id;
                int totaloperations;
                int position;
                bool status;
                int cposition;
};

#endif
```

Product.cpp

```
#include "Product.h"

char *Product::getName() {
	return this->name;
}

void Product::incrementPosition() {
	if (this->status == true) {
	} else if (this->position == (this->totaloperations)) {
		this->position++;
		this->status = true;
	}else {
		this->position++;
	}
}

void Product::decrementPosition() {
	if (this->getPosition() == 0) {
	} else
		this->position--;
}

int Product::getPosition() {
	return this->position;
}

void Product::setStatus(bool b) {
	this->status = b;
}

bool Product::getStatus() {
	return status;
}

void Product::setTotalOperations(int n) {
	this->totaloperations = n;
}

int Product::getTotalOperations() {
	return this->totaloperations;
}

void Product::setCPosition(int n) {
	this->cposition = n;
}

int Product::getCPosition () {
	return this->cposition;
}

void Product::setID(int n) {
	this->id = n;
}

int Product::getID() {
	return this->id;
}
```

## Registry.h

```
#ifndef _REGISTRY_H_
#define _REGISTRY_H_

#include <iostream>

using namespace std;

class Registry {

	public:

		Registry (char phase,int layer,char *productname,int ID,int
position,char *equipmentname, long int time, char *status) {
			this->product = (char *) malloc((strlen(productname) + 1) *
sizeof(char));
			strcpy(this->product,productname);
			this->ID = ID;
			this->position = position;
			this->equipment = (char *) malloc((strlen(equipmentname) + 1) *
sizeof(char));
			strcpy(this->equipment,equipmentname);
			this->phase = phase;
			this->layer = layer;
			this->time = time;
			this->status = (char *) malloc((strlen(status) + 1) *
sizeof(char));
			strcpy(this->status,status);
		}

		~Registry () {
			delete this->product;
			delete this->equipment;
			delete this->status;
		}

		void printRegistry();
		char *getProductName();
		int getID();
		int getPosition();
		char *getEquipmentName();
		char getPhase();
		int getLayer();
		long int getTime();
		char *getStatus();

	private:

		char *product;
		int ID;
		int position;
		char *equipment;
		char phase;
		int layer;
		long int time;
		char *status;
};

#endif
```

Registry.cpp

```
#include "Registry.h"

void Registry::printRegistry() {
	cout << "Fase: " << this->phase << ", camada: " << this->layer << ",
produto:" << this->product << ", com ID:" << this->ID << ", na Posicao:" << this-
>position << ", no Equipamento:" << this->equipment << ", no Tempo: " << this->time
<< ". \n";
}

char *Registry::getProductName() {
	return this->product;
}

int Registry::getID() {
	return this->ID;
}

int Registry::getPosition() {
	return this->position;
}

char *Registry::getEquipmentName() {
	return this->equipment;
}

char Registry::getPhase() {
	return this->phase;
}

int Registry::getLayer() {
	return this->layer;
}

long int Registry::getTime() {
	return this->time;
}

char *Registry::getStatus() {
	return this->status;
}
```

RegistryManager.h

```
#ifndef _REGISTRY_MANAGER_H_
#define _REGISTRY_MANAGER_H_

#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

#include "Registry.h"

class RegistryManager {

	public:

		RegistryManager () {
			this->listOfRegistrys = new vector<Registry *> ();
		}

		~RegistryManager () {
			delete this->listOfRegistrys;
		}

		void addRegistry (Registry *r);
		void addRegistry (char pha,int lay,char *productname,int ID,int
position,char *equipmentname, long int time, char *status);
		void listRegistrys ();
		void writeFile ();

	private:

		vector<Registry *> *listOfRegistrys;

};

#endif
```

RegistryManager.cpp

```
#include "RegistryManager.h"

void RegistryManager::addRegistry (Registry *l) {
	this->listOfRegistrys->push_back(l);
}

void RegistryManager::addRegistry (char pha, int lay, char *productname,int ID,int
position,char *equipmentname,long int time, char *status) {
	Registry *l = new
Registry(pha,lay,productname,ID,position,equipmentname,time,status);
	this->listOfRegistrys->push_back(l);
}

void RegistryManager::listRegistrys () {
	for (int i = 0;i < (int) this->listOfRegistrys->size()-1;i++) {
		this->listOfRegistrys->at(i)->printRegistry();
	}
}

void RegistryManager::writeFile () {
	/*ofstream myfile;
	myfile.open ("example.txt");
	myfile << this->listOfRegistrys->size() << " registros. \n";
	for (int i = 0;i < (int) this->listOfRegistrys->size();i++) {
		myfile << "Fase: " << this->listOfRegistrys->at(i)->getPhase() << ",
camada: " << this->listOfRegistrys->at(i)->getLayer() << ", produto:" << this-
>listOfRegistrys->at(i)->getProductName() << ", com ID:" << this->listOfRegistrys-
>at(i)->getID() << ", na Posicao:" << this->listOfRegistrys->at(i)->getPosition()
<< ", no Equipamento:" << this->listOfRegistrys->at(i)->getEquipmentName() << ", no
Tempo: " << this->listOfRegistrys->at(i)->getTime() << ". \n";
	}
	myfile.close();*/
	ofstream myfile;
	myfile.open ("example.csv");
	myfile << "Fase;Camada;Produto;ID;Posicao;Equipamento;Tempo;Status \n";
	for (int i = 0;i < (int) this->listOfRegistrys->size();i++) {
		myfile << this->listOfRegistrys->at(i)->getPhase() << ";" << this-
>listOfRegistrys->at(i)->getLayer() << ";" << this->listOfRegistrys->at(i)-
>getProductName() << ";" << this->listOfRegistrys->at(i)->getID() << ";" << this-
>listOfRegistrys->at(i)->getPosition() << ";" << this->listOfRegistrys->at(i)-
>getEquipmentName() << ";" << this->listOfRegistrys->at(i)->getTime() << ";" <<
this->listOfRegistrys->at(i)->getStatus() << "\n";
	}
	myfile.close();
}
```

RotateEvent.h

```
#ifndef _ROTATE_EVENT_H_
#define _ROTATE_EVENT_H_

#include "Event.h"


#include "MovingObject.h"

class RotateEvent : public Event {

	public:

		RotateEvent(MovingObject* o) : Event() {
			this->object = o;
			this->object->setBusy();

			this->startFlag = false;
			this->endFlag = false;

			this->axisX = 0;
			this->axisY = 0;
			this->axisZ = 0;

			this->startAngle = 0;
			this->vel = 0;
		}

		~RotateEvent()  {
			this->object->setFree();
		}


		virtual void setVector(float x, float y, float z);
		virtual void setInitialValue(float a);
		virtual void setRotationTax(float a);
		virtual void setTime(long int itime, long int duration);
		virtual void action(long int t);

	private:

		int sTime;

		MovingObject* object;

		bool startFlag;
		bool endFlag;

		float axisX;
		float axisY;
		float axisZ;

		float startAngle;

		float vel;

};

#endif
```

RotateEvent.cpp

```
#include "RotateEvent.h"

void RotateEvent::setVector(float x, float y, float z){
	this->axisX = x;
	this->axisY = y;
	this->axisZ = z;
}

void RotateEvent::setInitialValue(float a) {

	if (!startFlag) {
		this->startAngle = a;
		this->startFlag = true;
	}
}

void RotateEvent::setRotationTax(float a) {
	this->vel = a;
}

void RotateEvent::setTime(long int itime, long int duration) {
	this->sTime = itime;
	this->setStartTime(itime);
	this->setDuration(duration);
}

void RotateEvent::action(long int t) {
	this->object->setRotation(this->axisX, this->axisY, this->axisZ, this->vel);
}
```

## Simulator.h

```
#ifndef _SIMULATOR_H_
#define _SIMULATOR_H_

#include <iostream>
#include <stdlib.h>

using namespace std;

#include "OpenSGNav.h"
#include "Factory.h"
#include "Logic.h"
#include "Clock.h"
#include "EventList.h"

class Simulator : public vrj1::OpenSGNav {
	public:

		Simulator (vrj::Kernel* kern)
			: vrj1::OpenSGNav(kern) {
			this->name = NULL;

		}

		Simulator (vrj::Kernel* kern,char *simulatorname)
			: vrj1::OpenSGNav(kern) {
			this->name = (char *) malloc((strlen(simulatorname) + 1) *
sizeof(char));
			strcpy (this->name, simulatorname);
		}

		~Simulator() {
			cout << "Simulator::~Simulator() called\n";
		}

		virtual void setFactory(char *factoryname);
		virtual Factory *returnFactory();
		virtual char *getName();
		virtual void generateProductOnFactory(char *productname,float we,
float he, float de);
		virtual Logic *returnLogic();
		virtual EventList *returnEventList();
		virtual void init();
		virtual void initScene();
		virtual NodePtr getScene();
		virtual void draw();
		virtual void contextInit();
		virtual float getDrawScaleFactor();
		virtual void preFrame();
		virtual void intraFrame();
		virtual void postFrame();
		virtual void exit();
		virtual void generateRandomProduct (int ran, char *productname,float
we, float he, float de);

	private:

		char *name;
		Factory *factory;;
		Logic *logic;
		Clock *clock;
		EventList *eventlist;
```

```
            NodePtr localRoot;
};
#endif
```

## Simulator.cpp

```
#include "Simulator.h"

void Simulator::setFactory(char *factoryname) {
	this->factory->setName(factoryname);
	this->logic->setName(factoryname);
	this->logic->setFactory(this->factory);
}

Factory *Simulator::returnFactory() {
	return this->factory;
}


char *Simulator::getName() {
	return name;
}

void Simulator::generateProductOnFactory(char *productname,float we, float he,
float de) {
	this->factory->addProduct(productname,we,he,de);

}

Logic *Simulator::returnLogic() {
	return this->logic;
}

EventList *Simulator::returnEventList() {
	return this->eventlist;
}

void Simulator::init() {
	vrj1::OpenSGNav::init();
	this->factory = new Factory();
	this->logic = new Logic(this->factory);
	this->clock = new Clock();
	this->eventlist = new EventList(this->logic,this->factory->returnProducts());
}

void Simulator::initScene() {
	vrj1::OpenSGNav::initScene();
	for (int i=0; i<this->factory->totalEquipments(); i++) {
		this->addNode(this->factory->returnEquipmentByIndex(i)-
>getSceneGraph());
	}
}

NodePtr Simulator::getScene() {
	return vrj1::OpenSGNav::getScene();
}

void Simulator::draw() {
	vrj1::OpenSGNav::draw();
}

void Simulator::contextInit() {
	vrj1::OpenSGNav::contextInit();
}

void Simulator::preFrame() {
	this->eventlist->executeAPhase();
```

```
	this->eventlist->executeBPhase(this->clock->getClock());
	this->eventlist->executeCPhase(this->clock->getClock());
	vrj1::OpenSGNav::preFrame();
}

float Simulator::getDrawScaleFactor() {
	return vrj1::OpenSGNav::getDrawScaleFactor();
}

void Simulator::intraFrame() {
   vrj1::OpenSGNav::intraFrame();
}

void Simulator::postFrame() {
   vrj1::OpenSGNav::postFrame();
}

void Simulator::exit() {
	cout << "Simulator::exit() called\n";
	delete[] this->name;
	delete this->factory;
	delete this->logic;
	delete this->clock;
	delete this->eventlist;
	cout << "Simulator::exit() Finish\n";
	vrj1::OpenSGNav::exit();
}

void Simulator::generateRandomProduct(int ran, char *productname,float we, float
he, float de) {
	int randon = rand() % 10000 + 1;
	if ((randon >= 0)&&(randon <= ran)) {
		cout << "Gerando Produto " << productname << endl;
		this->generateProductOnFactory(productname,we,he,de);
	}
}
```

StationaryObject.h

```
#ifndef _STATIONARY_OBJECT_H_
#define _STATIONARY_OBJECT_H_

#include "GeoObject.h"

class StationaryObject : public GeoObject  {
       public:
              StationaryObject(char name[]) : GeoObject(name){}
              StationaryObject(float width, float height, float depth, float x,
float y, float z) : GeoObject(width,height,depth,x,y,z) { }
              ~StationaryObject() { }
};

#endif
```

StationaryPart.h

```
#ifndef _STATIONARY_PART_H_
#define _STATIONARY_PART_H_

#include "StationaryObject.h"

class StationaryPart : public StationaryObject {
      public:
            StationaryPart(char name[]) : StationaryObject(name) { }
            StationaryPart(float width, float height, float depth, float x, float
y, float z) : StationaryObject(width,height,depth,x,y,z) { }
            ~StationaryPart () { }
};

#endif
```

StoreEquipment.h

```
#ifndef _STORE_EQUIPMENT_H_
#define _STORE_EQUIPMENT_H_

#include "Equipment.h"

class StoreEquipment : public Equipment {

	public:

		StoreEquipment() : Equipment() { }
		StoreEquipment(char *equipmentName,float x,float y,float z) :
Equipment(equipmentName,x,y,z) {
			StationaryPart *sp = new
StationaryPart(0.5,0.5,0.5,0.0,0.0,0.0);
			this->registerStationaryPart(sp);
		}
		~StoreEquipment() { }

	protected:
		virtual void visualAction(long int t) {
			TranslateEvent *teProduct = new TranslateEvent (this-
>getNextProduct());
			teProduct->setTime(t,1);
			//teProduct->setDuration(50);
			teProduct->setInitialValues(0,0.3,0.0);
			teProduct->setFinalValues(0,0.3,0.0);
			sendEvent(teProduct);

		}
};

#endif
```

TranslateEvent.h

```
#ifndef _TRANSLATE_EVENT_H_
#define _TRANSLATE_EVENT_H_

#include "Event.h"
#include "MovingObject.h"

class TranslateEvent : public Event {

	public:

		TranslateEvent(MovingObject* o) : Event() {

			this->object = o;
			this->object->setBusy();

			this->startFlag = false;
			this->endFlag = false;
			this->timeFlag = false;

			this->startX = 0;
			this->startY = 0;
			this->startZ = 0;
			this->endX = 0;
			this->endY = 0;
			this->endZ = 0;
		}

		~TranslateEvent() {
			this->object->setFree();
		}

		virtual void setInitialValues(float x, float y, float z);
		virtual void setFinalValues(float x, float y, float z);

		virtual void setTime(long int itime, long int duration);
		virtual void action(long int t);

	private:

		void configureEvent();

		MovingObject* object;

		int sTime;

		bool timeFlag;
		bool startFlag;
		bool endFlag;

		float startX;
		float startY;
		float startZ;
		float endX;
		float endY;
		float endZ;

		float velX;
		float velY;
```

```
        float velZ;
};
#endif
```

TranslateEvent.cpp

```
#include "TranslateEvent.h"

void TranslateEvent::setInitialValues(float x, float y, float z) {
	this->startX = x;
	this->startY = y;
	this->startZ = z;
	this->startFlag = true;

	this->configureEvent();
}

void TranslateEvent::setFinalValues(float x, float y, float z) {
	this->endX = x;
	this->endY = y;
	this->endZ = z;
	this->endFlag = true;

	this->configureEvent();
}

void TranslateEvent::setTime(long int itime, long int duration) {
	this->sTime = itime;
	this->setStartTime(itime);
	this->setDuration(duration);
	this->timeFlag = true;

	this->configureEvent();
}

void TranslateEvent::action(long int t) {
	this->object->setPosition(velX*(t - sTime) + startX, velY*(t - sTime)+
startY, velZ*(t - sTime) + startZ);
}

void TranslateEvent::configureEvent() {
	if ((timeFlag)&&(startFlag)&&(endFlag)) {
		int d = getDuration();
		this->velX = (endX - startX)/d;
		this->velY = (endY - startY)/d;
		this->velZ = (endZ - startZ)/d;
	}
}
```

## TransportEquipment.h

```
#ifndef _TRANSPORT_EQUIPMENT_H_
#define _TRANSPORT_EQUIPMENT_H_

#include "Equipment.h"

class TransportEquipment : public Equipment {

	public:

		TransportEquipment() : Equipment() { }
		TransportEquipment(char *equipmentName,float x,float y,float z) :
Equipment(equipmentName,x,y,z) {
			StationaryPart *sp = new
StationaryPart(1.0,0.5,0.5,0.0,0.0,0.0);
			this->registerStationaryPart(sp);
			this->mp = new MovingPart (0.1,0.1,0.5,0.0,0.0,0.0);
			this->mp->setPosition(-0.45,0.3,0.0);
			this->registerMovingPart(this->mp);
		}
		~TransportEquipment() { }


	protected:

		virtual void visualAction(long int t) {
			TranslateEvent *teGo = new TranslateEvent (this->mp);
			teGo->setTime(t,10);
			teGo->setInitialValues(-0.45,0.3,0.0);
			teGo->setFinalValues(0.45,0.3,0.0);
			sendEvent(teGo);


			TranslateEvent *teBack = new TranslateEvent (this->mp);
			teBack->setTime(t+10,10);
			teBack->setInitialValues(0.45,0.3,0.0);
			teBack->setFinalValues(-0.45,0.3,0.0);
			sendEvent(teBack);

			TranslateEvent *teProd = new TranslateEvent (this-
>getNextProduct());
			teProd->setTime(t,10);
			teProd->setInitialValues(-0.45,0.3,0.0);
			teProd->setFinalValues(0.45,0.3,0.0);
			sendEvent(teProd);
		}

	private:

		MovingPart *mp;
};

#endif
```