Moving Bob

Guilherme F. Otranto
Thomas R. Gomes

Dezembro de 2007

Universidade de São Paulo
Instituto de Matemática e Estatística
MAC 0499 – Trabalho de Conclusão de Curso
Orientador: Junior Barrera
Co-orientador: Marco Gubitoso

Sumário

1. Introdução

Ao final de 2006 começamos a considerar o que fazer como trabalho de conclusão de curso, desejávamos encontrar algo relacionado a uma iniciação científica que fazíamos juntos. Como a iniciação envolvia desenvolvimento de jogos, nós decidimos criar uma ferramenta genérica que auxiliasse nesse trabalho de criação.

Diversas aplicações atuais requerem recursos gráficos bastante sofisticados para funcionar propriamente, como simuladores ou jogos. Essa necessidade gráfica dificulta o desenvolvimento rápido ou a prototipação dessas aplicações, visto que a implementação da parte gráfica tende a ser entre as mais custosas e demoradas de todo o desenvolvimento.

Por esse motivo decidimos criar uma ferramenta que facilite o desenvolvimento da parte gráfica, permitindo que os programadores concentrem seus esforços em outras áreas do projeto, diminuindo o tempo gasto na criação de aplicações com recursos gráficos avançados. Essa ferramenta recebeu o nome de Moving Bob.

Nosso objetivo com o Moving Bob foi criar um sistema modular, simples de utilizar, que permita ao programador exibir em um aplicativo um mundo modelado por ele e definir interações com o usuário.

Assim, ao usar o Moving Bob para o desenvolvimento de aplicações, esperamos que o programador consiga reaproveitar o módulo gráfico quase sem mudanças e escrever o essencial para sua aplicação modificando os outros módulos.

2. Princípios e Ferramentas

2.1 O Model-View-Controler

Para facilitar o reaproveitamento do nosso código, modularizar o sistema foi uma de nossas maiores preocupações. Por isso decidimos usar uma linguagem orientada a objetos, escolhemos programar em Java por termos maior familiaridade com ela. Outro princípio que auxilia na modularização é o de *Model-View-Controler* - MVC. Essa técnica consiste em projetar o sistema dividido em três módulos principais, cada um com uma funcionalidade específica.

O *Model* fica responsável pela modelagem do mundo, contendo as classes dos objetos que serão representados no sistema. No nosso caso, o *Model* tem uma classe para o personagem (*Player*), uma para o mundo onde o personagem anda (*Arena*) e um

pacote contendo as classes das ações (*Actions*).

O *View* é a ponte entre o usuário e o programa, exibindo os objetos numa tela de aplicativo, recebendo entrada do usuário e repassando para o controlador, que faz o tratamento adequado.

O *Controler* serve como controlador de fluxo de dados. Ele mantém o aplicativo rodando e trata comandos do usuário adequadamente, em geral modificando os estado do *Model*.

2.2 As bibliotecas Gráficas

Para desenvolver a interface gráfica, escolhemos usar a Java Open Graphics Library (JOGL). Criamos usando essa biblioteca um componente que ficaria responsável por desenhar objetos gráficos na tela, ele será referido como *componente JOGL* pelo resto desse texto. Integramos o nosso componente JOGL com o Swing, biblioteca gráfica do Java capaz de criar interfaces de maneira rápida e simples, para facilitar a criação de uma interface para o programa.

JOGL é uma biblioteca de desenvolvimento gráfico gratuita sendo desenvolvida pela Sun que permite o acesso da famosa biblioteca OpenGL em um ambiente de programação Java. A OpenGL (Open Graphics Library) foi desenvolvida para a linguagem C e também é gratuita.

A JOGL funciona de forma muito semelhante à OpenGL, elas são ambas máquinas de estado que o programador controla a fim de obter resultados. Em essência a biblioteca fica responsável por manter correndo um ciclo infinito e chamar métodos específicos que o programador deve implementar. O mais importante desses métodos é o *display*, que é chamado sempre que o desenho na tela deve ser atualizado (em geral diversas vezes por segundo). Nesse método o programador deve especificar todos os objetos gráficos presentes na cena, a posição da câmera e da luz. Além disso a JOGL pode também chamar funções quando alguns eventos acontecem, como por exemplo movimento do mouse, redimensionamento da janela, ou mudanças no teclado.

O Swing é um *toolkit* de Java que permite a criação de interfaces gráficas para aplicações em Java. Ele fornece diversos tipos de componentes que podem ser usados na interface, como botões, sliders, caixas de texto, checkboxes e também permite que a JOGL faça uma renderização direta em um de seus paineis.

A seguir faremos uma explicação mais detalhada dos conceitos por trás da modelagem gráfica do Moving Bob, desde criação de objetos e animações até a inclusão desses elementos no código Java.

2.3 A Modelagem Gráfica

Exibir graficamente um ambiente, personagens e ações exige um modelo de cada um desses elementos, para isso uma estrutura de representação gráfica torna-se necessária.

A JOGL representa objetos tridimensionais através da união de suas faces, ou seja, um cubo nada mais é do que 6 quadrados posicionados adequadamente, onde cada face não passa de uma lista de seus vértices. Para criar objetos usando essa representação poderíamos criar cada uma de suas face separadamente e listá-las para a biblioteca gráfica, mas quando tratamos objetos complexos, com mais de 1000 faces, essa abordagem torna-se inviável.

Recorremos então ao uso de programas especializados em criar essas representações, programas de modelagem gráfica. Esses modeladores gráficos oferecem uma interface visual para a criação de elementos tridimensionais e facilitam muito a criação de objetos bastante complexos. A saída desses modeladores precisava agora ser adaptada a entrada pedida pela nossa biblioteca gráfica.

2.3.1 O Formato OBJ

O formato .obj é um padrão bastante direto e simples de se representar modelos 3D, ele pode ser usado pela maioria dos modeladores gráficos conhecidos (3DSM, Maya, Blender e outros) e contém todas as informações necessárias para que nossa biblioteca gráfica consiga criar sua representação do objeto.

Em essência o formato .obj é uma lista de todos os vértices do objeto seguido de uma lista de todas as faces (usando referência aos vértices definidos anteriormente). Com apenas isso seriamos capazes de desenhar o objeto em tela usando nossa biblioteca escolhida, mas com algumas limitações, o objeto não seria sombreado adequadamente e nem seria capaz de receber texturas.

Para eliminar essas limitações a JOGL precisa de mais algumas informações dos objetos, em especial ela precisa dos vetores normais às faces para ser capaz de sombrear o objeto adequadamente e das coordenadas de textura para cada vértice em cada face para conseguir mapear uma imagem no objeto. Um .obj pode conter todas essas informações, sua estrutura fica da seguinte forma:

- **Lista de todos os vértices:** A definição de todos os vértices que pertencem ao objeto. Um vértice é definido em cada linha, que começa pelo caractere v seguido de 3 números, as coordenadas x, y e z do vértice (a precisão pode ser *double*).
- **Lista das coordenadas de textura (opcional):** A lista que contém todas as

coordenadas de textura que serão usadas no objeto. Em geral cada vértice pode receber diversas coordenadas, uma para cada face que ele ajuda a definir. Uma coordenada é definida por linha, que começa pelo caracteres vt seguidos de 3 números (em geral a 3ª coordenada é nula, pois desejamos mapear uma textura bidimensional em nosso objeto tridimensional). Os números representam as coordenadas u, v e w da imagem que devem ser colocadas sobre aquele vértice naquela face. No caso de uma imagem, o valor de w é sempre nulo.

- **Lista de vetores normais (opcional):** A lista contém um vetor normal calculado para cada vértice, de forma que a normal das faces são a combinação das normais de seus vértices. Isso é usado para criar um sombreamento suave das faces, quando as normais de vértices estão inclinadas em relação a faces vizinhas, eles criam uma transição suave entre as faces, escondendo quinas. Novamente, os vetores são definidos um por linha, que começa pelos caracteres vn seguidos de 3 números, valores do vetor nas coordenadas x, y e z.
- **Lista de todas as faces:** A lista contém a definição de todas as faces do objeto, ela usa os índices das outras listas para referenciar quais vértices definem a face, quais suas coordenadas de textura e quais suas normais. A definição de uma face ocupa uma linha, que começa pelo caractere f seguido da definição dos vértices da face em ordem anti-horária seguindo a fronteira da face.
  Cada definição de vértice pode ser apenas um número, que representa o índice dele na lista de vértices (caso coordenadas de textura e normais não estejam definidas) ou pode ser 3 índices separados por /, o índice do vértice, seguido da coordenada de textura dele naquela face, seguido do índice de seu vetor normal. Note que se normais foram definidas mas coordenadas de textura não, o índice referente à coordenada é removido, deixando um espaço vazio.

2.3.2 Exemplo de OBJ:

A seguir um exemplo de um tetraedro definido em formato .obj, de 3 formas diferentes, inicialmente sem definição de normais ou texturas, a seguir adicionamos normais e finalmente adicionamos também coordenadas de textura.

Ao lado o .obj que define o mínimo possível, apenas as coordenadas dos vértices e as faces do tetraedro.

Note que os índices começam pelo 1, e não pelo zero, como é frequente na computação.

```
v 0.707107 0.707106 0.000000
v 0.258819 -0.965926 0.000000
v -0.965926 0.258819 0.000000
v 0.000000 0.000000 1.000000
f 2 1 4
f 4 3 2
f 4 1 3
f 3 1 2
```

No .obj mostrado ao lado temos o mesmo tetraedro, dessa vez as normais são definidas também. Note que na definição das faces, referencias são feitas à lista de vértices e também à lista de normais, nada aparece onde estariam as referências às coordenadas que não foram definidas.

```
v 0.707107 0.707106 0.000000
v 0.258819 -0.965926 0.000000
v -0.965926 0.258819 0.000000
v 0.000000 0.000000 1.000000
vn 0.863950 -0.231495 0.447214
vn -0.632456 -0.632455 0.447214
vn -0.231495 0.863950 0.447213
vn 0.000000 0.000000 -1.000000
f 2//1 1//1 4//1
f 4//2 3//2 2//2
f 4//3 1//3 3//3
f 3//4 1//4 2//4
```

Nesse último exemplo temos o .obj em sua forma mais completa, que define coordenadas de textura e normais para o tetraedro.

É importante notar que o tamanho do arquivo fica consideravelmente maior quando coordenadas de textura são definidas, por isso torna-se bastante interessante evitar ao máximo usá-las quando não são absolutamente necessárias. Veremos a seguir uma maneira de re-usar as coordenadas definidas para um objeto durante as animações dele, afim de economizar tempo e memória. Isso é possível pois apesar das posições de vértices e normais mudarem em uma animação, as coordenadas permanecem iguais.

```
v 0.707107 0.707106 0.000000
v 0.258819 -0.965926 0.000000
v -0.965926 0.258819 0.000000
v 0.000000 0.000000 1.000000
vt 0.000000 1.000000 0.0
vt 0.000000 0.000000 0.0
vt 1.000000 0.000000 0.0
vt 0.000000 1.000000 0.0
vt 0.000000 0.000000 0.0
vt 1.000000 0.000000 0.0
vt 0.000000 1.000000 0.0
vt 0.000000 0.000000 0.0
vt 1.000000 0.000000 0.0
vt 0.000000 1.000000 0.0
vt 0.000000 0.000000 0.0
vt 1.000000 0.000000 0.0
vn 0.863950 -0.231495 0.447214
vn -0.632456 -0.632455 0.447214
vn -0.231495 0.863950 0.447213
vn 0.000000 0.000000 -1.000000
f 2/1/1 1/2/1 4/3/1
f 4/4/2 3/5/2 2/6/2
f 4/7/3 1/8/3 3/9/3
f 3/10/4 1/11/4 2/12/4
```

Com um .obj completo, como o descrito acima, podemos abrí-lo dentro do nosso programa, ler e guardar as listas de vértices, texturas e normais e finalmente usar essas listas quando encontrarmos definições de faces para representar cada face como o pedido pela JOGL. Com isso conseguimos criar e trazer para nossa aplicação modelos gráficos bastante complexos sem grandes trabalhos de modelagem.

2.3.3 Animações

Um elemento importante para o Moving Bob foi o sistema de ações criado para a visualização de movimento. Decidimos usar animações para representar as ações realizadas por nossos personagens dentro do nosso mundo. Surgiu então a necessidade de representar essas animações de forma que a JOGL fosse capaz de desenhá-las corretamente. A seguir vamos analisar o conceito de animações usado pelo Moving Bob e como ele foi aplicado.

2.3.4 Animações Quadro-a-quadro

O conceito de animações quadro-a-quadro é bastante simples: criam-se diversas poses consecutivas para o objeto que se deseja animar, cada uma delas um pouco diferente da seguinte. Quando as poses são visualizadas seqüencialmente de forma rápida cria-se a impressão de um movimento fluído. Um exemplo clássico desse tipo de animação é obtido quando desenhamos figuras ligeiramente diferentes no canto de todas as páginas de um livro, ao folhearmos ele rapidamente, nossas figuras parecem ganhar vida e se animar.

Em nosso sistema computacional esse conceito não muda muito, para criar uma animação nos modelamos cada quadro dela, criamos um arquivo .obj para cada um e depois desenhamos eles seqüencialmente na tela, dando a impressão de movimento ao usuário.

É claro que criar cada quadro individualmente seria muito trabalhoso, mas programas de modelagem permitem a criação de animações desse tipo de maneira muito mais simples, com o uso de esqueletos e interpolação de quadros. Falaremos mais dessas técnicas a seguir.

2.3.5 Criação de Animações

Usando programas de modelagem gráfica podemos criar animações bastante sofisticadas sem muito trabalho e em pouco tempo. Algumas técnicas oferecidas por esses programas facilitam essa criação, vamos rever as duas principais a seguir:

- **Uso de Esqueletos:** Como o nome sugere, o esqueleto é um estrutura que move o objeto. O esqueleto não é visível na animação, ele é apenas uma estrutura auxiliar para mover o objeto de forma mais comportada e previsível. O usuário define um esqueleto como um conjunto de ossos e atribui a cada osso um conjunto de vértices do objeto, então quando ele move algum osso do esqueleto aquele

conjunto de vértices se move também. Essa técnica é excelente quando se anima um personagem humanóide, como por exemplo o Bob (um boneco de madeira semelhante a um humano).

- **Interpolação de quadros:** Muitas ferramentas de animação permitem essa técnica, que na verdade preenche o espaço entre dois quadros da animação com um estado intermediário, formando uma transição mais suave entre eles. Podemos então definir o primeiro e o décimo quadro de uma animação por exemplo, e o programa vai completar os 8 quadros intermediários com uma interpolação uniforme dos dois extremos. Dessa forma é possível criar uma animação de 40 quadros definindo apenas alguns deles quadros e permitindo que o programa preencha os espaços. Em geral algum ajuste fino se faz necessário nessa interpolação, mas o trabalho pesado já está feito.

Com uma modelagem do objeto pronta e com o esqueleto criado, o processo de criar novas ações para esse objeto é bastante rápido, graças a essas técnicas de animação.

2.3.6 Animações no Moving Bob

As animações criadas como descrito anteriormente são na verdade um conjunto de diversos arquivos .obj, cada um contendo um dos quadros da animação. Para trazê-las para nosso sistema escrevemos um interpretador que lê os arquivos .obj e extrai o objeto de cada um colocando-o em um vetor, possibilitando o acesso de cada quadro da animação separadamente.

Com as ações descritas em um vetor bastou criar uma inteligência capaz de informar o componente JOGL de qual quadro deve ser desenhado a cada instante e conseguimos visualizar as ações no sistema. Cada personagem (elemento capaz de realizar ações) é representado no sistema pela classe *Player*, e ele tem uma referência para a ação que ele está executando e em que quadro dessa ação ele está. A execução do programa é responsável por atualizar o quadro da ação, bem como trocar as ações que cada personagem está executando baseada em controle do usuário ou regras próprias.

Por exemplo, com o modelo do Bob pronto, criamos uma animação de 40 quadros que representam ele pulando. Uma vez registrada no sistema, os quadros do pulo são carregados para um vetor associado a ação pular. Se em algum momento durante a execução o usuário der o comando de pulo, o controlador vai fazer com que o Bob passe

a executar a ação de pulo. A cada passo da execução, o quadro da ação pulo será atualizado, e a posição no vetor onde o componente JOGL vai buscar a representação do Bob avança, logo sua representação muda, fazendo com que ele pareça pular.

Em termos de organização, em geral colocamos todos os .obj de uma ação em uma pasta com seu nome. Cada .obj recebe o nome da ação seguido do número do quadro que ele representa. Dessa forma fica fácil adaptar uma nova ação dentro do sistema, basta adicionar uma linha para ela na matriz de ações, e em cada posição daquela linha pedir para que o carregador de .obj leia o arquivo referente aquele quadro da animação.

2.4 O Blender

O Blender é o programa que usamos para fazer a modelagem gráfica do Bob e suas ações, outros programas podem ser usados junto com o Moving Bob, mas nós decidimos usar o Blender por ser um programa gratuito e excelente para fazer o que precisávamos. O programador usando o Moving Bob pode fazer sua modelagem gráfica em qualquer programa que exporte .obj, nós sugerimos o Blender por ser um modelador com ferramentas muito poderosas de edição gráfica e animação, além de ser gratuito.

A criação de animações quadro-a-quadro usando o Blender é bastante ágil e simples, o animador pode definir um esqueleto para seu objeto e usar as técnicas de interpolação de quadros para agilizar a criação.

O Blender oferece um sistema de copiar e colar poses, permitindo também que o animador copie uma pose e a cole invertida simetricamente. Ou seja, o animador pode copiar uma pose onde o osso do pé esquerdo está levantado e colar a pose inversa, onde o osso do pé direito vai estar levantado. Essa ferramenta facilita muito a criação de animações onde a segunda parte é simétrica a primeira, porém invertida. Como por exemplo um ciclo de caminhada ou corrida, onde o segundo passo é essencialmente o primeiro invertido.

O sistema de esqueletos na animação permite a criação de uma hierarquia de ossos, com isso podemos fazer um esqueleto onde o movimento do osso da coxa faz os ossos da canela e pé se moverem de acordo, isso é muito prático para animar um ser com esqueleto de verdade.

Ao exportar .obj, o Blender pode gerar as normais para as faces e também as coordenadas de texturas, caso definidas. Quando exportando uma animação podemos pedir para que ele exporte a animação completa, nesse caso o Blender cria automaticamente um arquivo para cada quadro, como usado no Moving Bob.

No site oficial do Blender (*www.blender.org*) pode-se encontrar manuais de uso, tutoriais escritos e em vídeos, galerias de imagens feitas com o programa, documentação e uma revisão de todas as ferramentas oferecidas por ele. Além disso o programa está disponível para download no site.

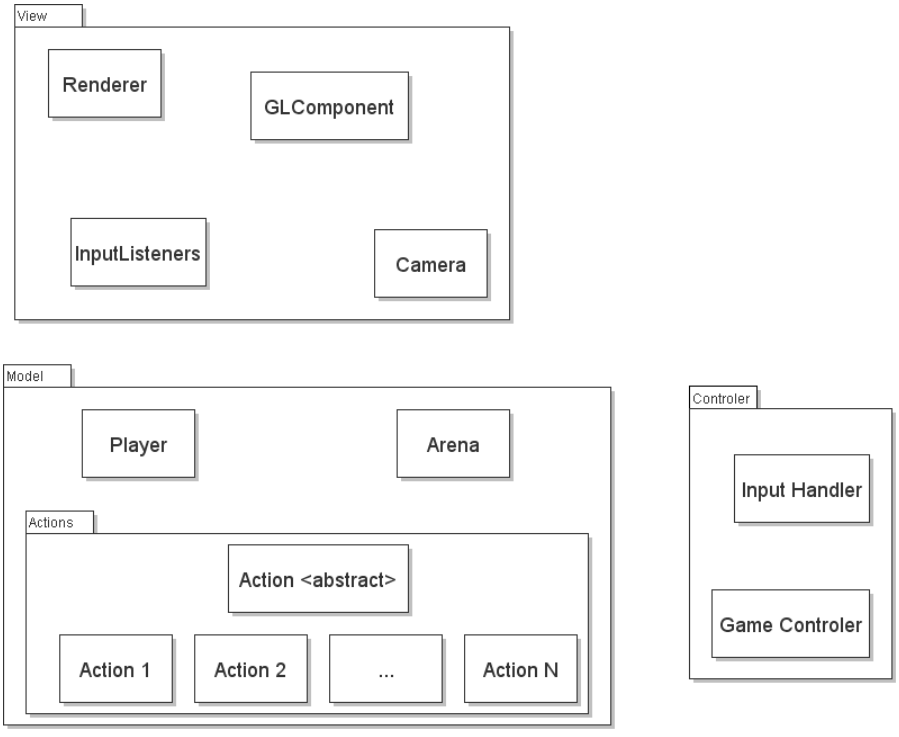
3. Desenvolvimento

3.1 Atividades Realizadas

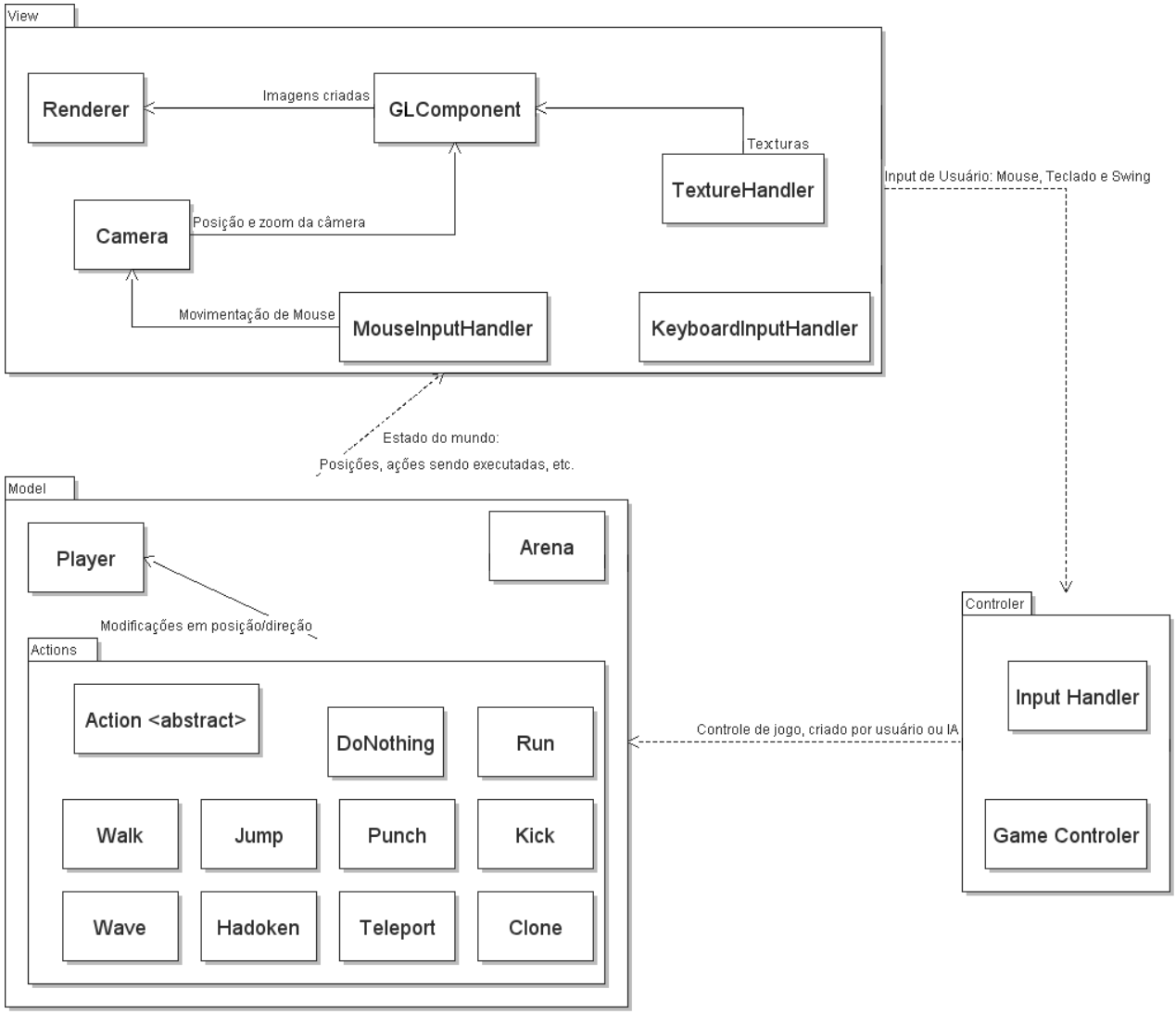
Seguindo os princípios de engenharia de software começamos o TCC planejando como seria o nosso sistema, a definição do que ele seria ser capaz de fazer gerou um certo debate. Chegamos a conclusão de que importar objetos e animações construídas em modeladores gráficos e permitir uma manipulação desses elementos gráficos dentro do nosso programa deveria ser nossa prioridade.

Desde o início estava claro que usaríamos uma linguagem orientada a objetos, e a escolha de Java foi fácil por diversos motivos. Precisávamos de uma linguagem que fosse bastante conhecida, para aumentar o número de possíveis usuários do Moving Bob, e também gostaríamos de usar alguma biblioteca gráfica poderosa, e a JOGL integrado ao Swing se encaixava perfeitamente.

Então começamos a modelagem do sistema. Como durante a iniciação científica tomamos contato com o modelo MVC, criamos um para representar a estrutura do sistema. O diagrama resultante segue abaixo.

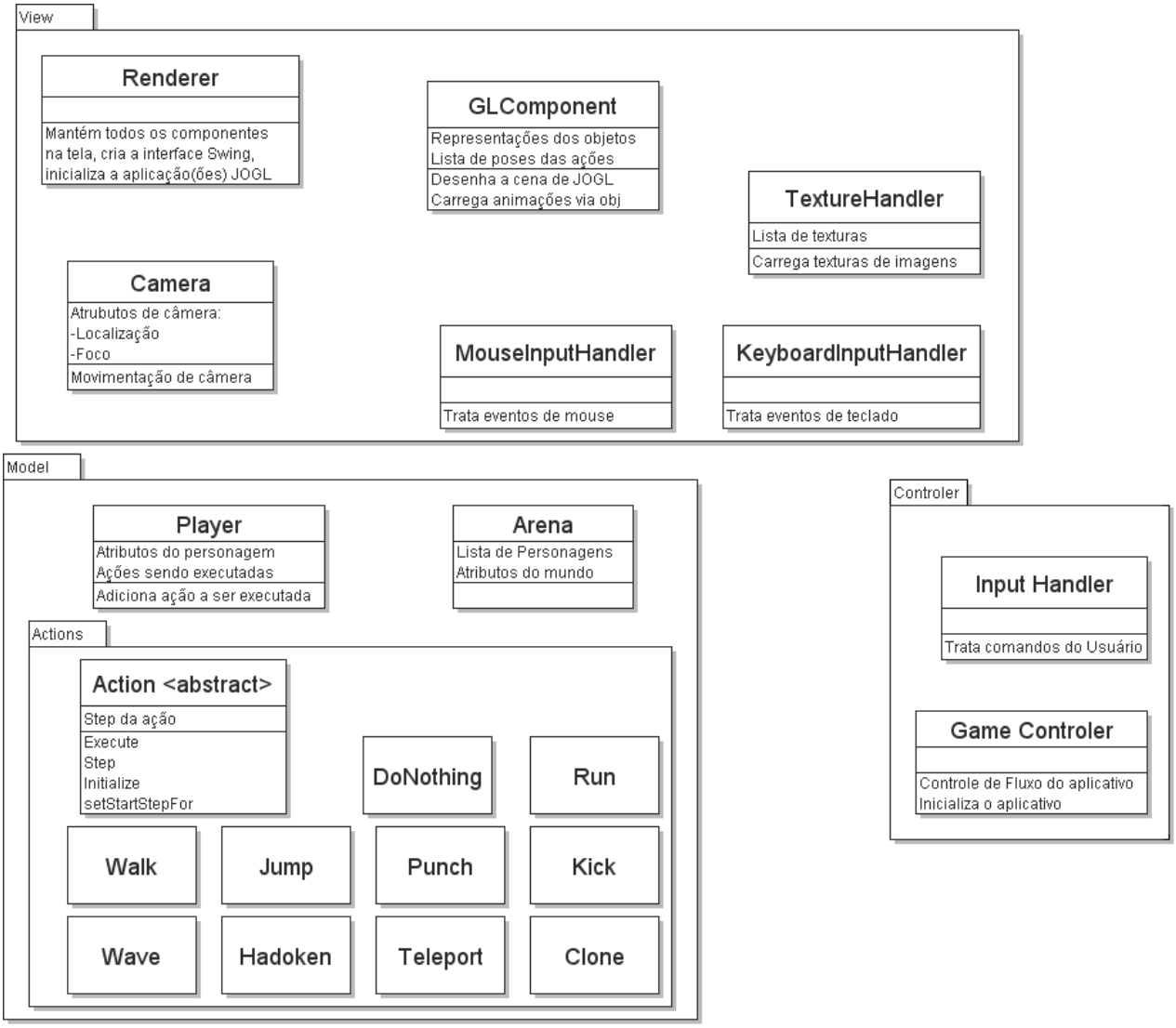


Como isso não bastava para a modelagem, tivemos de fazer diagrama de fluxo de dados. Esse diagrama contém uma visão geral das comunicações entre os módulos, deixando claro qual o fluxo de informações correndo no sistema.



Tivemos um certo trabalho arrumando o fluxo de dados, mas a ajuda dos nossos orientadores nos permitiu concertar os problemas rapidamente. Com esse diagrama pronto, já sabíamos o suficiente sobre o sistema para fazer um diagrama de classes. Esse último serve para deixar claro aos envolvidos no projeto quais as classes que pretendemos implementar e suas funcionalidades.

A seguir o diagrama de classes:



Esse planejamento todo nos ajudou muito no início da programação, sabíamos que classes o sistema possuía e como elas se encaixavam. Isso nos permitiu trabalhar separadamente sempre que necessário, aumentando muito o tempo total de dedicação ao projeto.

Para programar em Java usamos o Eclipse como ambiente de desenvolvimento. Ele facilitou a criação do código, uma vez que ele ajuda muito o programador, importando as classes automaticamente, gerando o código dos métodos herdados e cuidando de importar as bibliotecas externas como a JOGL.

Iniciamos o projeto com a criação de um protótipo para verificar a viabilidade do sistema e preparar uma base mais estável e robusta.

3.2 O protótipo

Como a modelagem lógica do sistema e o controlador eram mais simples que a parte gráfica eles seriam preparados quase inteiramente nessa fase, enquanto o visualizador e os modelos gráficos seriam, inicialmente, simplificados, bidimensionais e sem os tratamentos de textura e leitura de .obj.

Iniciamos a modelagem pelo mundo, ou seja a classe *Arena* e a classe *Player*. A *Arena* guarda referências ao personagens e a *Player* contém a posição, direção e alguns métodos de acesso do personagem. Criamos também o pacote *Actions* onde colocamos a classe abstrata *Action* e as classes *Move* e *DoNothing* que representando as ações andar e ficar parado.

No *Controler* modelamos apenas duas classes, a *GameControler* e *InputHandler*. A *GameControler* é responsável por inicializar o aplicativo e manter ele rodando e a *InputHandler* é a porta de comunicação do *View* com o *Model*, tratando adequadamente as chamadas de usuário.

Essa modelagem inicial é suficiente para representar um grande leque de ações (dado que cada uma seja modelada de acordo). Tendo um personagem num campo livre pode-se mandar o personagem andar, pular, fazer piruetas, e até voar. Essa era nossa base desejada, com ela concluída pudemos fazer alguns testes e verificar que era simples colocar mais recursos no mundo e criar outros objetos interativos.

A estrutura havia sido planejada com cuidado para permitir uma grande maleabilidade do sistema e ao mesmo tempo manter uma certa simplicidade para o programador. Percebemos que, uma vez que o modelo estava completo, criar uma ação nova era bastante rápido quanto a modificações no código, e, em termos de animação era possível construir novas animações de maneira bastante rápida, usando as técnicas descritas anteriormente.

O *View* inicialmente era apenas o componente JOGL, que desenhava objetos simples na tela, como um círculo para cada personagem. A câmera ainda era fixa, não haviam texturas, nenhum tratamento de mouse ou teclado e a integração com o Swing ainda não havia sido feita.

Esse foi o protótipo inicial do sistema. O que restou a seguir foi implementar todas as funcionalidades do View que nos comprometemos a fazer, criar um leque maior de ações, documentar o código e fazer testes constantes.

3.3 A Evolução do View

O *View* precisava ganhar diversas funcionalidades antes que o projeto estivesse completo. A primeira coisa a ser feita era aceitar inputs do usuário, para isso registramos duas classes na JOGL, uma para tratar eventos de mouse e outra para tratar eventos de teclado. O tratamento inicial era bastante simplificado, visto que a câmera ainda estava fixa e o mundo era bidimensional.

O próximo passo foi expandir o mundo para 3 dimensões e permitir o movimento da câmera. Os elementos gráficos ainda seriam bastante simplificados, mas agora eles seriam volumes, como esferas para os jogadores. A movimentação de câmera foi feita através de movimentos de mouse, o usuário pode rotacionar a câmera e mudar o zoom dela. Todo o tratamento para a movimentação de câmera é feito em uma classe própria, a *Camera*, nela implementamos métodos de controle usando álgebra linear.

A seguir integramos o componente JOGL ao Swing e criamos uma interface de bõtoes bastante simplificada. A renderização do JOGL em um pipeline direto para o Swing foi usada para manter a performance do Moving Bob o mais alta possível.

Com o sistema permitindo movimentação de câmera em um espaço tridimensional, foi necessário repensar o tratamento do clique do mouse. O método que faz esse tratamento foi um dos mais complexos do sistema, apesar de todas as informações necessárias estarem disponíveis, elas não eram facilmente obtidas ou manipuladas. A idéia para fazer o tratamento é que ao clicar na tela (em uma posição x, y qualquer) o usuário define uma trajetória reta no espaço 3D. Precisamos descobrir qual a equação dessa reta no nosso volume de visualização e com isso podemos calcular intersecções com os objetos do nosso mundo. Os passos executados quando o usuário clica na tela são os seguintes:

- Guarda a posição x, y onde o usuário clicou na tela, x é um valor no intervalo [0, largura da tela] e y está em [0, altura da tela].
- Descobre a posição da câmera, onde ela está olhando (ponto de foco), qual a abertura da câmera (zoom).
- Encontra o vetor que entra pela tela no espaço, ele é a subtração do foco da câmera pela posição da mesma.
- Encontra o vetor que anda para a direita no espaço em relação a tela, ele é o produto vetorial dos vetores (0, 0, 1) – que aponta para cima – e do vetor que entra no espaço. Isso é verdade pois o vetor que procuramos é o único perpendicular ao que entra na tela e ao que sobe no espaço 3D.
- Encontra o vetor que vai para cima no espaço 3D em relação a câmera. Esse vetor

não é o (0, 0, 1) porque a câmera pode estar inclinada, mas podemos encontrá-lo fazendo o produto vetorial dos vetores que entram no espaço e que anda para a direita no espaço, visto que ele é perpendicular aos dois.

- Normaliza a posição x, y de clique na tela para o intervalo [0,1], usando o tamanho da janela. Subtrai 0.5 desse valor, deixando-o no intervalo [-0.5,0.5] ele representa em termos normalizados onde o usuário clicou na tela, considerando o centro da tela como origem.
- Anda o valor normalizado x do clique vezes o tamanho lateral do espaço de visualização no vetor que vai para a direita. Anda o valor y vezes a altura do espaço no vetor que vai para cima. Com isso achamos o ponto no espaço referente ao inicio da trajetória definida pelo clique do usuário. Se andarmos para dentro do espaço, de acordo com o vetor que entra, encontraremos a trajetória completa!

Com tratamento de mouse sendo feito apropriadamente, faltava criar um leitor de texturas, que foi feito de maneira bastante rápida. Agora as esferas podiam receber imagens. O próximo passo era criar o leitor de .obj para tornar as representações mais complexas.

O leitor de .obj foi feito em partes, inicialmente ele reconhecia apenas .objs do tipo mais simples, sem definição de normais ou texturas. Após um certo estudo do formato, fomos capazes de adaptar o nosso sistema para entender .obj com ou sem a definição desses itens opcionais. Ao ler um arquivo, nosso interpretador já desenha o objeto em uma lista que será chamada quando aquele objeto for necessário.

Finalmente criamos o sistema que lê automaticamente todos os .obj que representam quadros de uma ação e cria a matriz de ações. Com ela criada já podíamos ver nossos elementos gráficos executando animações modeladas no Blender. Com esse animador atingimos os objetivos estipulados no início do trabalho.

3.4 O Model e o Controler

Inicialmente o main do programa é bem simples:

```
public static void main(String[] args) {
    new Renderer();

    (new Thread(GameControler.instanceOf())).run();
}
```

Iniciamos poucas coisas, e chamamos o construtor do *Renderer*. Ele faz todo o tratamento pesado de interface e gráficos, cria os modelos do personagem, inicializa o

mundo, prepara a interface com os botões e registra os *EventListeners*.

É importante de avisar aos programadores de Swing, ou outro aplicativo gráfico do Java, que cada *EventListener* usa sua própria thread para pegar e tratar os eventos. Ou seja, se algum usuário chamar algum método demorado, em especial os que fazem a thread dormir, o *EventListener* para de ouvir eventos pois ele fica ocupado na chamada de método.

Em nosso sistema encontramos esse problema logo no começo, e resolvemos isso de uma maneira simples, criamos uma thread para o controlador do programa, que fica responsável por executar as ações do*(s)* personagem*(ns)*, enquanto os *EventListeners* apenas fazem chamadas simples para os controladores que alteram pequenos elementos.

Uma vez que as threads estão todas organizadas o *Run* fica pedindo que todos os personagens executem suas ações.

Não inicializamos explicitamente a *Arena*, ou o *InputHandler*, o mesmo acontece com algumas classes do *View*. Isso se deve pois não faz sentido (no nosso aplicativo) múltiplas instâncias dos controladores de fluxo de informações, então utilizamos o padrão de design *Singleton.* Esse padrão garante que exista apenas uma instância das classes de controle e todas as chamadas para elas são feitas através de um método estático que retorna a referência daquela instância.

O *View*, durante a execução, fica ouvindo por eventos de mouse e teclado (isso inclui os eventos internos do *Swing* como clique de botão). Sempre que um desses eventos acontece o *View* verifica se deve tratá-lo ou não. Caso seja uma movimentação de câmera ou o botão de fechar, o tratamento é feito internamente. Caso seja um comando para o personagem, o *InputHandler* é chamado para que tratar adequadamente aquele comando. Com isso deixamos encapsulado dentro do *View* tudo referente à interface e deixamos que o *GameControler* cuide do fluxo de dados.

O *InputHandler* por sua vez cria a ação que foi pedida pelo usuário. No caso de um clique na tela, ele recebe o ponto clicado e cria uma ação de movimentação para o personagem ir até o lugar desejado. Com isso ele encerra o seu trabalho, deixando que a thread do *GameControler* cuide de mandar o personagem se mover, e libera a thread do *EventListener* para que ela volte a ouvir eventos.

3.5 A modelagem da ação

Em paralelo com a criação do *View*, definimos o que era importante para representar uma ação. Com isso definimos uma classe abstrata que conteria a estrutura necessária para uma ação. Depois disso o *Model* foi pouco modificado, apenas ações foram adicionadas e algumas refinadas para ficarem mais naturais. A seguir tomaremos um tempo para explicarmos como ficou a modelagem das ações e como elas devem ser usadas no sistema atual.

A seguir o código da classe abstrata *Action*.

```
public abstract class Action {
    protected Player active;
    protected Player target;
    protected String name;
    protected String flag = "";
    protected int step;

    /* Métodos abstratos que caracterizam uma ação e
     * devem ser implementados pelas subclasses */
    protected abstract void execute();
    public abstract void initialize();
    public abstract void step();
    public abstract void setStartStepFor(Action action);

    /* Getters */

    public String getName(){
        return this.name;
    }

    public int getStep(){
        return step;
    }

     * Construtor genérico de uma ação qualquer.
    public Action (String name, Player active, Player target){
        this.name = name;
        this.active = active;
        this.target = target;
    }

     * Método que avisa a ação que uma próxima ação está na fila.
    public void setFlag(String flag) {
        this.flag = flag;
    }
}
```

A classe abstrata *Action* serve de super classe para todas as ações. Dessa forma uma ação pode ser executada sem que se saiba sua classe, sabe-se apenas que ela

possui os métodos definidos na *Action*. Esse é o princípio de uma interface, porém escolhemos implementar alguns métodos e manter algumas variáveis de classe na *Action*, então ela não foi feita como uma interface. A seguir veremos quais métodos devem ser implementados por todas as ações e o que eles fazem.

- **step():** Corre um passo de iteração. Usada para controlar a velocidade da animação e o momento adequado de executá-la. Nela existe um tratamento sobre a próxima ação a ser executada, que determina quando é o momento adequado da ação acabar, permitindo que se implemente uma transição suave entre ações.
- **execute():** O execute altera o estado do *Model* para refletir o efeito da ação, como alterar posição ou direção de personagens. Ele pode também executar uma ação mais complexa, por exemplo, durante um tiro ao alvo o execute determinaria a precisão e onde acertou o tiro, isso aconteceria no momento determinado pelo step().
- **setStartStepFor(Action action):** Esse método foi criado para dar uma idéia de fluidez na transição de ações. Em alguns sistemas mais avançados, como alguns jogos tridimensionais, queremos que a movimentação do personagem pareça a mais fluida possível, evitando interrompê-la no meio durante a transição de ações. Para isso criamos esse método, que determina a partir da última ação sendo executada qual será o passo na animação que iremos começar na próxima ação. Isso simula uma transição direta de uma ação para outra, sem uma pausa.
- **initialize():** Algumas ações precisam armazenar informações internas referentes a sua execução. Essas informações precisam ser inicializadas em algum momento, então quando a ação se torna ativa, esse método inicializa os valores necessários.

A seguir um exemplo do Move, responsável por locomover o personagem.

```
public class Move extends Action{
    private boolean ending = false;
    private double speed;

     * Construtor da classe Move.
    public Move(String name, Player active, Player target) {
        super(name, active, target);
    }

     * Função que executa a ação apropriadamente.
    protected void execute() {
        if (this.active.getX() != this.target.getX() || this.active.getY() != this.target.getY()){
            this.active.setMoving(true);
            if (this.ending){
                if (this.step == 0){
                    this.active.setMoving(false);
                    this.active.endAction();
                }
                this.active.setLocation(
                        this.active.getX() + (this.target.getX() - this.active.getX())/(40 - this.step),
                        this.active.getY() + (this.target.getY() - this.active.getY())/(40 - this.step));
            }
            else{
                this.active.setLocation(
                        this.active.getX() + this.active.getDirX()*this.speed,
                        this.active.getY() + this.active.getDirY()*this.speed);
            }
        }
        else{
            this.active.endAction();
            this.active.setMoving(false);
        }
    }

     * Inicializa a direção do personagem, sua velocidade e poe uma aura no ponto desejado.
    public void initialize(){
        double distance;
        distance = Math.sqrt((target.getX() - active.getX()) * (target.getX() - active.getX()) +
                (target.getY() - active.getY()) * (target.getY() - active.getY()));
        active.setDirection((target.getX() - active.getX())/distance, (target.getY() - active.getY())/distance);
        this.speed = 1.5/40.0;
        Arena.instanceOf().setCliked(true);
        Arena.instanceOf().setAuraPoint(this.target.getX(), this.target.getY());
        if (distance < 1.5){
            this.ending = true;
        }
    }
     * Faz um passo de ação.
    public void step(){
        double distance;
        this.step = (step + 1)%40;
        this.execute();
        if (this.step == 0){
            distance = Math.sqrt((target.getX() - active.getX()) * (target.getX() - active.getX()) +
                    (target.getY() - active.getY()) * (target.getY() - active.getY()));
            if (distance < 1.5){
                this.ending = true;
            }
            if (!this.flag.equals("")){
                this.active.setMoving(false);
                this.active.endAction();
            }
        }
    }
     * Determina qual o passo inicial da animação da ação.
    public void setStartStepFor(Action action) {
        this.step = 0;
    }
}
```

A classe acima serve como exemplo de implementação de ação. Escolhemos usar o Move pois sua complexidade é grande o suficiente para ilustrarmos o que pode ser feito porém não grande o bastante para atrapalhar na compreensão.

Move, como seu nome indica, faz o personagem se deslocar para um lugar de destino. Armazenamos o destino no campo *target* da classe *Action*, esse campo representa o alvo da ação, ele é do tipo *Player*. Para o move bastaria armazenar as coordenadas de destino, mas armazenando o alvo em um *Player* o sistema fica mais genérico, permitindo ações cujo alvo é um outro personagem, como um tiro por exemplo.

Ao terminar sua execução, o move chama um método que atribui o valor nulo para a ação ativa do personagem que estava se movendo. A seguir temos um trecho de código que mostra como o *Player* troca ações ativas e executa elas.

```
 * Adiciona no next action do personagem a próxima ação a ser executada.
public void setNextAction(Action nextAction) {
    this.nextAction = nextAction;
    if (this.currentAction == null){
        this.currentAction = new DoNothing("DoNothing", this, null);
    }
    this.nextAction.setStartStepFor(this.currentAction);
}

 * Método que verifica se o jogodor está executando uma ação e se existe uma próxima
public void performAction(){
    if (this.currentAction == null && this.nextAction == null){
        this.currentAction = new DoNothing("DoNothing", this, null);
        this.currentAction.initialize();
    }
    else if (this.currentAction == null){
        this.currentAction = this.nextAction;
        this.currentAction.initialize();
        this.nextAction = null;
    }
    if (this.nextAction != null){
        this.currentAction.setFlag(this.nextAction.getName());
    }
    this.currentAction.step();
}

 * Finaliza a ação corrente.
public void endAction(){
    Arena.instanceOf().setCliked(false);
    this.currentAction = null;
}
```

O código a seguir está na classe Player, os métodos mostrados são os responsáveis por fazer a transição de ações sendo executadas pelo personagem, adicionar uma nova ação ou executar a ação atual.

Para modelar uma ação o programador usando o Moving Bob deve fazer o seguinte:

- Criar uma sub-classe de *Action*, que implementa os métodos de acordo com a ação sendo modelada.
- Adicionar no *View* e no *InputHandler*, um meio do usuário acessar a ação através de um comando. Isso pode significar adicionar um botão na interface Swing, criar

um atalho de teclado ou criar a inteligência para disparar a ação automaticamente.

- Modelar graficamente a ação, criando uma série de .obj, um para cada quadro da animação que representa o personagem executando a ação.
- Adicionar no *View* chamadas para o interpretador de .obj que lê a animação e desenha cada quadro dela na matriz de ações.

A representação gráfica das ações é apenas um dos elementos necessários na criação delas, especialmente se elas devem causar efeitos no estado do mundo. É necessário cuidar da parte lógica também, ou seja, o personagem não deve correr mais no seu ciclo de movimentação que sua animação da a entender, além disso, se uma ação causa mudanças ao seu redor, essas mudanças devem ser programadas.

4. Resultados

Ao fim de 2007 concluímos nosso objetivo de criar uma ferramenta de visualização de ações que agilizava o desenvolvimento de aplicações com componentes gráficas.

Testamos a praticidade do sistema ao modelar uma pequena demonstração para a apresentação do projeto. Em pouco tempo modelamos diversas ações, editamos ações antigas e testamos a conexão entre elas. Criamos um pequeno mundo para nosso personagem e fizemos uma interface para o usuário bastante sofisticada muito rapidamente.

Terminamos o projeto com uma ferramenta com muito potencial para ser utilizada em outros sistemas. Nós inclusive planejamos alguns projetos pessoais onde Moving Bob será a base para o desenvolvimento dos gráficos. A ferramenta permanecerá no site da disciplina e poderá ser baixada por qualquer usuário interessado em experimentar e aprimorar o sistema. Nós estaremos disponíveis para qualquer tipo de ajuda que um novo usuário do Moving Bob possa precisar. Vamos também disponibilizar os arquivos do Blender usados para fazer as animações, caso o usuário queira fazer algum ajuste fino sobre a movimentação do Bob.

O uso do projeto é bem simples. Três passos precisam ser seguidos para criar e exibir uma nova ação.

- Criar o modelo gráfico de quadros seqüenciais que representam uma animação. Isso poder ser feito com qualquer ferramenta de animação e modelagem tridimensional como o Blender, 3DStudioMax, Maya, etc. Ao final da animação os quadros devem ser exportados em arquivos do formato .obj, cada um em seu

próprio arquivo. Usamos uma convenção de nome de arquivo *"<nome da ação><número do quadro>.obj "* , outras convenções podem ser usadas, mas para isso uma adaptação pequena do código deve ser feita.

- Criar o modelo lógico da ação. Criar a classe correspondente à ação como subclasse da classe Action e implementar os métodos adequadamente.
- Carregar a ação dentro do View, e programar para que a ação aconteça quando desejado, por exemplo, quando o usuário aperta uma determinada tecla.

5. Referências

- **JAVA –** É a linguagem de programação desenvolvida pela Sun. Usamos funcionalidades da versão 5.0 para frente, então o código não funcionaria para versões mais antigas.
  http://java.sun.com/
- **Eclipse –** Ambiente de desenvolvimento gratuito para a plataforma JAVA.
  http://www.eclipse.org/
- **JOGL –** Biblioteca gráfica gratuita para a linguagem JAVA, baseada em OpenGL e sendo desenvolvida pelo Game Technology Group na Sun Microsystems.
  https://jogl.dev.java.net/
- **Swing –** Um toolkit para a plataforma JAVA destinado ao desenvolvimento de interfaces gráficas, incluso nas bibliotecas padrão do JAVA.
  Ver o site da Sun, referenciado em JAVA.
- **Blender –** Um programa voltado para a modelagem gráfica, distribuído sobre GNU GPL (GNU General Public License) desenvolvido para múltiplas plataformas.
  http://www.blender.org/ - Site oficial.
  http://wiki.blender.org/index.php/Manual – Wiki do manual.
  http://download.blender.org/documentation/htmlI/ - Documentação oficial, ver seção 4 para uma introdução excelente às técnicas de animação usando Blender.
- **Formato Wavefront (.OBJ) -** Formato usado para armazenar objetos 3D, lido pelo Moving Bob.
  http://www.fileformat.info/format/wavefrontobj/ - Especificação do formato.
- **Violet –** Ferramenta escrita em JAVA, que permite a criação de diagramas UML de forma simplificada. Disponível em formato WebStart ou .jar executável. Também disponibilizado sob GNU GPL.
  http://horstmann.com/violet/

- **Material online –** Agradecemos aos seguintes sites pelo maravilhoso material disponível online que nos ajudou a completar o Moving Bob
    - **Mayang´s Free Textures –** Site que contém milhares de texturas gratuitas para download.
      http://mayang.com/textures/
    - **NEHE Tuturiais –** Site com tutoriais para OpenGL, com código das lições disponíveis em diversas formas. Um bom site para adquirir uma base teoria em computação gráfica.
      http://nehe.gamedev.net/
    - **Tutoriais JOGL –** Site com tutoriais em JOGL bastante úteis para pessoas novas a biblioteca.
      http://jerome.jouvie.free.fr/OpenGl/Tutorials1-5.php

6. Parte Subjetiva

6.1 Guilherme Otranto

Sempre gostei muito de desenvolvimento gráfico, em especial quando aplicado a jogos. Essa foi a principal razão que me levou à iniciação científica que mais tarde seria responsável pela escolha do Moving Bob como projeto de conclusão de curso.

Além de aplicar diversos conhecimentos adquiridos durante o curso de BCC, aprendi muitas coisas novas ao projetar e desenvolver o Moving Bob, desde técnicas de modelagem e animação gráfica até o uso de programas e bibliotecas voltadas a parte gráfica do desenvolvimento de aplicações.

A preparação para o projeto foi possivelmente mais demorada e trabalhosa de que a programação em si, mas com toda certeza foi de enorme ajuda para que fizéssemos uma ferramenta melhor. Considero que a preparação para o projeto possa ser dividida em duas partes de igual importância, o planejamento da estrutura e a pesquisa de bases e conceitos necessários para a realização do Moving Bob.

A parte de planejamento estrutural consistiu em criarmos um esqueleto que serviria de base para toda a programação. Esse esqueleto foi essencial para a criação o sistema sem encontrar em algum problema estrutural e nos obrigasse a repensar o sistema. Acredito que ganhamos muito tempo graças a um bom esqueleto e um planejamento sólido. Os professores Junior Barrera e Marco Gubitoso ajudaram muito durante toda

essa fase de preparação, sua experiência em desenvolvimento de sistemas nos salvou de muitos problemas futuros devido a erros estruturais e nos ajudou muito a criar um esqueleto estável e robusto para o Moving Bob.

A pesquisa de bases foi bastante complicada de início, era necessário um estudo de conceitos e ferramentas que permitissem atingir uma meta que eu considerei bastante ambiciosa desde o começo. Apesar de todas as frustrações de ferramentas mal documentadas e técnicas pouco desenvolvidas, gostei muito de fazer a pesquisa, aprendi com ela inúmeras coisas úteis, desde como usar ferramentas interessantes até como encontrar material pouco divulgado.

Com toda a preparação feita, nos restava programar o Moving Bob e checar os resultados. A programação foi muito mais rápida e suave do que eu havia antecipado, atribuo isso a um esqueleto bem feito e uma escolha de ferramentas adequada. Alguns problemas surgiram durante a programação, conseguimos localizar e resolver a maior parte deles de forma rápida, nesse sentido um sistema modularizado se mostrou incrivelmente útil para um diagnóstico preciso e ágil. Minha maior frustração durante essa fase foi a documentação incompleta e muitas vezes pouco precisa da biblioteca gráfica que escolhemos. Apesar de se encaixar perfeitamente em nosso projeto, o que justificava seu uso, o JOGL deixa a desejar em termos de documentação. Grande parte do que aprendi em relação ao uso biblioteca foi lendo tutoriais e abstraindo conceitos deles, habilidade que acredito ser bastante útil para minha vida futura.

Matérias e Conceitos aprendidos no BCC:

Quando revejo as diversas fases do projeto vejo com facilidade quais matérias e conceitos aprendidos nos meus anos de BCC foram fundamentais para o desenvolvimento do Moving Bob:

- **Planejamento estrutural:**

Os conceitos aprendidos em Programação Orientada a Objetos (MAC0441) foram fundamentais para a criação de uma estrutura orientada a objetos bem feita, que usava padrões aprendidos nessa matéria para resolver diversos problemas encontrados e modelava o sistema com um método *Model-View-Controler* estudado também em POO. Além disso acredito que matérias como Laboratório de Programação I e II (MAC0211 e MAC0242 respectivamente) e Engenharia de Software (MAC0332) nos deram experiência em desenvolvimento e planejamento de sistemas que nos permitiu evitar erros passados e manter conceitos que funcionaram de maneira satisfatória.

Contamos muito com a ajuda de nossos orientadores e de conceitos aprendidos no

BCC nessa fase, sem isso acredito que nosso planejamento não seria bom o suficiente para o projeto e teríamos de replanejá-lo durante a fase de programação, o que atrasaria bastante o desenvolvimento.

- **Pesquisa de Bases:**

Diversas matérias do BCC nos colocaram em contato com ferramentas ou conceitos que viemos a usar durante o projeto. Em especial devo mencionar Computação Gráfica (MAC0420) que usou a biblioteca gráfica OpenGL, apresentando vários conceitos e técnicas que são aplicados na versão em Java da mesma biblioteca que usamos o JavaOpenGL (JOGL). Além de CG os laboratórios de programação nos ajudaram a escolher ferramentas auxiliares ao desenvolvimento do Bob, como o Swing e o Eclipse.

- **Programação:**

Nossa base conceitual em programação foi adquirida nas matérias de introdução a programação (MAC0110 e MAC0122) e Estruturas de Dados (MAC0323), essa base foi fundamental para conseguirmos programar o que havia sido planejado. Além dessas matérias, Computação Gráfica ajudou muito em termos de conceitos e maneiras de resolver problemas usando o JOGL. Programação Concorrente (MAC0432) foi bastante útil quando encontramos problemas de concorrência entre as threads do nosso sistema e na própria programação usando Java. E devo a Geometria Computacional (MAC0331), Álgebra Linear (MAT0139) e Métodos Numéricos (MAC0300) toda a base matemática que foi usada para fazer o tratamento de mouse em espaço 3D e movimentação de câmera.

- **Geral:**

  Não posso deixar de mencionar matérias como:

  - Programação Linear (MAC0315)
  - Métodos Numéricos da Álgebra Linear (MAC0300)
  - Álgebra II (MAT0213)
  - Cálculo III (MAT0211)
  - Conceitos Fundamentais de Linguagens de Programação (MAC0316)

Essas matérias, que considero entre as mais complicadas ou difíceis do curso, aumentaram muito a nossa capacidade e paciência de atacar um problema que não sabemos como resolver até chegarmos a uma resposta. Nos mostraram como procurar um material de estudo pouco disponível, como contornar problemas ainda não solucionados e nos deram uma nova perspectiva do que pode ser considerado complexo,

difícil ou até mesmo trabalhoso.

Seguindo o Caminho do Moving Bob

Pretendo continuar o desenvolvimento do Moving Bob. Gostei muito do resultado que obtivemos durante esse ano e já planejei alguns projetos que usarão o Moving Bob. Também planejo melhorar a própria ferramenta, aumentando suas funcionalidades e melhorando as existentes. Acredito que a documentação possa ser adaptada conforme eu receba retorno de usuários para que ela fique cada vez mais completa e intuitiva. Pretendo também criar alguns tutoriais para usuários iniciantes que mostrem como o Moving Bob pode ser usado, o que precisa ser feito e o que pode ser modificado.

Acredito que o Moving Bob seja uma base fantástica para a criação de protótipos rápidos, o que para mim é ótimo, visto que estou sempre me envolvendo em projetos com idéias novas que podem ou não funcionar. É importante ser capaz de testar idéias de maneira ágil e decidir quais merecem mais dedicação.

Gosto também da idéia de divulgar o Moving Bob, aumentando o número de usuários com certeza aumentaremos o número de pessoas sugerindo e implementando melhorias no sistema.

6.2 Thomas Gomes

Desafios

Acredito que o maior desafio no nosso trabalho foi aprender os detalhes de uso do JOGL, que apesar de ser uma excelente ferramenta, possui uma documentação pouco informativa.

Ao longo do curso programamos diversas vezes em Java, isso foi uma ótima experiência pois sempre que eu não entendia um método, ou não sabia as capacidades de uma classe, eu pudia consultar a documentação dela e aprender funcionalidades que eu não conhecia. Isso infelizmente não foi plenamente possível com o JOGL, cuja documentação deixou a desejar. Algumas vezes tivemos que recorrer à documentação do OpenGL para procurar certas informações.

Outra complicação foi a modelagem do sistema. Por um lado, como programador, eu sinto sempre vontade de começar a programar o quanto antes, especialmente para testar idéias, e eu tendo a focar em código quando discutindo os conceitos e requisitos que o projeto deve ter, pois eu me preocupo se serei capaz de implementar tudo. Porém

isso não é bom para a fase de desenvolvimento conceitual e levantamento de requisitos. Tive que lutar contra essa minha vontade e focar no modelo abstrato do projeto. Apesar das discussões geradas por causa disso, tenho em mente que nessa fase inicial é plenamente importante se desvincular do código e modelar corretamente os conceitos.

Essa modelagem também não foi fácil. Tivemos que fazer diversos diagramas, diversas vezes, discutir a validade das metas e qual estrutura usaríamos. A hierarquia de classes e seus métodos deveria ser bem planejada para garantir um sistema encapsulado e modular. Precisávamos criar um sistema robusto e genérico que não usava de soluções temporárias ou restritas.

Disciplinas relevantes:

- MAC0323 Estruturas de Dados: Acredito que uma boa base de estrutura de dados permite uma generalização para qualquer projeto, é e de extrema importância escolher a estrutura adequada.
- MAC0211 Laboratório de Programação I: Como uma matéria que envolveu um projeto com interface gráfica, ao menos para nós, foi importante para aprender como desenvolver projetos maiores que os exercícios programas que vemos o tempo todo. Aprendi que se não houver planejamento e dedicação o projeto sofre bastante.
- MAC0332 Engenharia de Software: Apesar da matéria não ter tido muita influência, imagino que noções de engenharia de software sejam importantes para criar um software decente.
- MAC0300 Métodos Numéricos da Álgebra Linear: Sempre importante para lidar com pontos e aplicar rotação, translação e reflexão, funções que apesar de prontas em módulos gráficos como o JOGL, são importantes para manipulação de câmera e movimentação de pontos em aplicativos gráficos.
- MAC0438 Programação Concorrente: Foi uma matéria bem interessante do curso, e apesar no nosso sistema não ser um sistema diretamente concorrente, ele possui as diversas threads dos event listeners do java. Como a matéria forneceu uma excelente base sobre concorrência, pudemos lidar com essas threads sem problemas, garantindo a consistência.
- MAC0331 Geometria Computacional: Qualquer aplicativo que lide com polígonos e poliedros fará bom uso dessa matéria. Os diversos conceitos de geometria envolvidos são importantes para a manipulação dos dados.
- MAC0421 Computação Gráfica: Sem dúvida e matéria mais importante para o

nosso projeto. Nela desenvolvemos e aplicamos conceitos de computação gráfica e tivemos experiência direta com o OpenGL que foi a base do JOGL, que usamos no projeto. Não só aprendemos os conceitos do pipeline gráfico, como também pudemos sentir as capacidades do OpenGL no projeto que fizemos.

- MAC0441 Programação Orientada a Objetos: Acho que não só a escolha de linguagem orientada a objetos bem como a estrutura de classes foi influenciada por essa matéria. Eu considero ela muito importante para o curso, uma vez que atualmente orientação a objetos vem se tornando cada vez mais popular.

Aplicação de conceitos

Muitos dos conceitos aprendidos no IME são importantes para o desenvolvimento de aplicativos e softwares. Porém como esse conceitos são muito abrangentes, poucos são aplicados em uma área específica, como a de aplicativos gráficos. Ainda mais que a maioria das matérias que envolvem estudos da área gráfica, são opcionais no instituto.

Continuação na área

Nosso projeto pertence à área de jogos na minha opinião. Primariamente criação de ferramentas gráficas. Gosto muito da área de jogos, porém eu prefiro um pouco mais a criação de sistemas, de modelagem, que da parte gráfica. Mas mesmo assim é uma área de interesse que eu pretendo me manter próximo. Pretendo futuramente estudar como os projetos são modelados e quais ferramentas e funcionalidades eles usam, tanto da área gráfica como das outras.