SCE0201 - Computação Gráfica
Rosane Minghim

P.A.E.: Danilo Medeiros Eler

Relatório do Projeto
Tiro ao Alvo

Arthur Filipe M. Nascimento - n°USP 5634455
Rafael Piovesan C. Machado - n°USP 5634945
Valter D. Moraes Junior - n°USP 5634820
Ulisses F. Soares - n°USP 5377365

1 Introdução

Como a implementação do projeto se tornou muito próxima do que foi especificado no início e como as diferenças foram, em geral, aditivas, vamos recolocar a especificação inicial do projeto e posteriormente indicar as diferenças.

1.1 Idéia Inicial do Projeto

A proposta inicial, de acordo com o nosso pré-projeto, era essa:

O cenário é uma barraca de tiro-ao-alvo de um parque de diversões. A cena mostra um lugar quase fechado, com paredes, teto e chão. Pendurados nas paredes e teto estão animais de pelúcia que são usados de prêmio.

Além disso também se vê a bancada da barraca (textura de madeira) logo abaixo da câmera e muito próximo dela. Sobre ela estão a munição restante à esquerda do usuário e os prêmios acumulados à sua direita. Os prêmios são alguns dos animais de pelúcia pendurados pelas paredes. Ao se acertar um alvo, um animal de pelúcia aleatório do cenário some da parede e reaparece sobre a bancada.

No centro da barraca se vê uma mesa em três níveis. Sobre essa mesa correm os alvos com velocidades constantes. Eles aparecem de um lado da mesa, correm sobre ela e somem ao chegarem no final. Sobre o primeiro e o terceiro níveis os alvos seguem em um sentido; no segundo nível da mesa os alvos correm no sentido contrário. O alvos são placas de metal em formato de coelho.

A arma usada será uma espingarda de pressão que atira rolhas. Ela é visível apenas como um cano (cilindro com textura de metal) com uma rolha (cilindro com textura de cortiça) na extremidade. Esse arranjo se move junto com a

câmera, como se o jogador estivesse sempre olhando através da mira da arma. A mira é uma cruz fixa no centro da tela e indica grosseiramente onde o tiro do usuário deve acertar.

A câmera terá restrições com relação aos ângulos de visão que ela irá permitir. Ela não permitirá que o usuário olha para fora da barraca, e portanto, para trás. Também não permitirá que ele mire diretamente para o chão ou para o teto.

1.2 O que foi feito

A implementação foi muito próxima do inicialmente especificado, mas difere nestes poucos pontos:

Prêmios: Os bichos de pelúcia distribuídos pela barraca foram substituídos por troféus sobre uma prateleira ao fundo da barraca.

Alvos: Os alvos agora se movem mais livremente sobre as mesas. Eles aparecem aleatoriamente no canto de uma mesa e seguem até encontrarem o final dela no outro extremo. As suas velocidades também são aleatórias, mas com pouca variação.

Mira: A mira não é mostrada, já que não achamos que ela se adequava à proposta. Em um jogo em um parque de diversões não se vê uma cruz flutuando à medida que se move a arma, então também não deveríamos ver isso no nosso programa.

Objetos na bancada: Colocar as rolhas disponíveis e os prêmios obtidos sobre a bancada seria trivial, já que precisaríamos mudar apenas poucas variáveis no nosso mundo 3D. Além disso, como não tínhamos mais a mira na visão 2D, resolvemos transportar essa funcionalidade. Ou seja, agora as rolhas disponíveis e os prêmios obtidos são mostrados em painéis 2D na tela. Assim continuamos usando objetos no mundo 3D junto com o mundo 2D.

Visão panorâmica: Através de experimentações, conseguimos melhorar consideravelmente a apresentação da barraca adicionando algum contexto. Inicialmente permitimos que a barraca não fosse mais fechada, mas sim vazada nas laterais, como é comum em barracas de parques de diversões. Como não havia o que se ver além das paredes da barraca, colocamos uma imagem panorâmica logo além dos limites da barraca. Mais sobre isso será falado na secção *Destaque de Detalhes*.

2 Estrutura Geral do Programa

O programa foi dividido em um programa principal (src/main.cpp) mais algumas classes separadas (pares .cpp/.h no diretório src). Essas classes implementam objetos reais que podem ou não serem vistos na cena.

Os objetos que têm componentes visuais derivam de uma classe-base *Visual* que simplifica o que seria a função '*display*' de cada um. Os objetos que se movem na cena derivam de uma classe-base *Movel* (que por sua vez também deriva de *Visual*) que dá a mesma vantagem sobre os objetos que precisam que

suas posições/animações sejam recalculadas a cada momento. Mais será dito sobre essas duas classes na próxima secção.

Quase todas as outras classes implementam diretamente objetos da cena, visual e funcionalmente, e podem ser facilmente identificadas no código-fonte. São elas: *Arma*, *Rolha*, *Alvo*, *Premio*, *Barraca* (envolve também todo o cenário em volta da barraca e partes fixas dentro dela), *Painel* (painéis 2D mostrados na frente mostrando a quantidade de rolhas), *Camera* (usada apenas para alterar a matriz ModelView básica, mas não mostra nenhum objeto adicional na cena), *Premio* (mostra os prêmios sobre a prateleira e a contagem de prêmios ganhos na base da tela à esquerda).

3 Destaque de Detalhes

3.1 Classes *Visual* e *Movel*

O primeiro destaque que queremos indicar é as classes *Visual* e *Movel* pela grande simplicidade que elas proporcionaram ao nosso código. Elas foram implementadas logo no início do projeto, junto com o início do programa principal. Depois que essas classes foram estabilizadas, uma boa parte do código principal ficou inalterada até o final, em particular as funções de 'display' dos objetos e uma outra função responsável por recalcular os movimentos dos objetos (através de glTimerFunc).

Entre outras coisas, a classe *Visual* permitiu que a função 'display' tivesse apenas quatro linhas. Essa classe faz com que os objetos que derivam dela que são instanciados sejam inseridos em uma lista geral de objetos visuais. Essa lista, então, é iterada na função 'display' do programa principal, que chama um método de todos os objetos que faz com que eles sejam mostrados na tela.

Algo similar foi feito com a classe *Movel*, com a única diferença de que o método acessado pela função de recálculo do movimento retorna um valor verdadeiro/falso. Este valor indica se o movimento do objeto já está terminado ou não. Se tiver terminado, o objeto é removido e não é mais iterado.

Um trecho de código para exemplo:

Arquivo *src/main.cpp, linhas 78-79*: núcleo da função 'display'

```
for(list<Visual *>::iterator obj = osObjetosVisuais.begin();
                          obj!=osObjetosVisuais.end(); obj++)
    (*obj)->desenha();
```

Além das linhas acima, a função '*display*' usa apenas uma chamada à função *glClear* e outra à função *glutSwapBuffers*. Isso mostra a simplicidade obtida com a classe *Visual*. A classe *Movel* provê a mesma simplicidade aos objetos dotados de mobilidade.

3.2 Cinemática do jogo

Uma classe chamada *vetor3d* havia sido criada por elementos do grupo para trabalhos de outras disciplinas e foi aproveitada neste projeto. Ela implementa uma forma transparente de se calcular equações vetoriais através da sobrecarga dos operadores adequados. Assim, os movimentos de objetos móveis puderam ser mapeados com equações simples e implementados diretamente. Os exemplos

seguintes comparam o código escrito com fórmulas da cinemática bem conhecidas.

Arquivo *alvo.cpp, linha50*: $s = s_0 + vt$

```
pos_atual = pos_inicial + velocidade*tempo;
```

Arquivo *rolha.cpp, linha50*: $s = s_0 + v_0 t + \frac{at^2}{2}$

```
pos_atual = pos_inicial + velocidade*tempo + aceleracao*tempo*tempo*0.5;
```

3.3 Visão Panorâmica

O último ponto que gostaríamos de destacar é o cenário distante que foi implementado.

Escolhemos a imagem panorâmica de um campo aberto e experimentamos várias formas de dispô-la ao fundo. A primeira tentativa envolveu colocar a imagem sobre um plano atrás da barraca. Mas o resultado visual foi insatisfatório pois não dava a idéia correta de distância. O resultado dava a impressão de se estar próximo de um outdoor, mas não em um campo aberto.

Então tentamos colocá-la sobre dois planos paralelos às paredes da barraca, junto com o plano anterior. Contudo, o resultado também foi menos do que o esperado, pois dava a impressão de se estar dentro de uma sala fechada com as paredes pintadas com uma paisagem.

A solução foi dispor a nossa imagem panorâmica sobre uma superfície curva distante da barraca. Essa superfície foi feita a partir de um meio cilindro elíptico facetado. Isso proporcionou o melhor resultado. Contudo, ainda era possível ver partes sem texturas abaixo e acima dessa superfície, e isso foi resolvido completando essas áreas em branco com texturas de grama e céu azul.

4 Manual do Usuário

O objetivo do usuário no jogo é acumular a maior quantidade de prêmios que puder com uma quantidade limitada de rolhas.

A jogabilidade do programa é bastante intuitiva, já que envolve apenas o mouse e da forma mais simples possível. Uma vez que o programa esteja em execução, o usuário deve mover o mouse para fazer com que a arma se mova. Quando ele achar que pode acertar um tiro em um pato, o usuário deve clicar uma vez com o botão esquerdo do mouse (e soltar). Isso irá fazer com que uma rolha seja atirada da arma na direção escolhida.

Na base da janela o usuário verá dois painéis: um mostrando a contagem de prêmios recebidos, à esquerda, e outro mostrando a contagem de rolhas que ainda pode atirar, à direita.

Quando o usuário ficar sem tiros ele pode clicar uma vez com o botão direito do mouse (e soltar) para reiniciar o jogo. Isso fará com que a arma se recarregue, mas também fará com que ele perca os seus prêmios acumulados.

Para sair do jogo, o usuário pode apertar a tecla ESC no teclado ou fechar a janela do programa.

5 Instruções de Compilação e Execução

5.1 Requisitos

1. Hardware com capacidade de processamento e renderização 3D, com os drivers devidamente instalados.

2. Um sistema com capacidade de compilar programas em linguagem C++. Recomenda-se o uso de alguma distribuição Linux devidamente capacitada para compilação, mas este programa também pode ser compilado em outros sistemas derivados de Unix ou no Windows sem maiores dificuldades.

3. A biblioteca OpenGL disponível para compilação.

4. A biblioteca DevIL[1] (Developer's Image Library), conhecida antigamente como OpenIL. Para esta biblioteca também é necessário suporte para compilação, isto é, instalar os pacotes de desenvolvimento (-dev) no Linux ou os pacotes que disponibilizem arquivos .lib no Windows.

5.2 Como Compilar

Primeiro é necessário descompactar o pacote com o código-fonte. No terminal do Linux use o comando:

```
tar xvjf GC.tar.bz2
```

No Windows é necessário um programa como o 7z ou WinRAR para descompactar o pacote.

5.2.1 No Linux

Em um terminal de linha de comando, entre na pasta criada e use o programa *make* para compilar automaticamente o programa.

```
cd CG/
make
```

Se o programa *make* não estiver disponível, então use a seguinte linha de comando (dentro da pasta descompactada) para compilar manualmente o projeto:

```
g++ -o CG src/*.cpp -lm -lglut -lILUT
```

Após estes passos, o programa deve estar compilado e o executável deve se chamar CG no diretório corrente.

Estes passos também devem funcionar em outros sistemas derivados de Unix e no MinGW/Cygwin no Windows com pouca ou nenhuma alteração.

[1] Disponível para várias plataformas em: http://openil.sourceforge.net/

5.2.2 No Windows

Se a interface de desenvolvimento Code::Blocks estiver disponível, abra o arquivo de projeto *CG.cbp* no diretório criado pela descompactação do pacote. Então pressione Ctrl+F11 para recompilar o projeto. Isso gera um arquivo CG.exe no diretório principal do projeto.

Caso seja usada outra interface de desenvolvimento (MSVC++, Dev-C++ etc) é necessário criar um projeto novo nessa interface e adicionar todos os arquivos de código-fonte do diretório *src*. Então configurar o projeto para fazer as ligações corretas com o OpenGL e DevIL e comandar a compilação. O programa gerado deve ser colocado no diretório principal do projeto para que o programa encontre as texturas adequadamente.

5.3 Execução

Para executar o projeto basta clicar no ícone do executável ou executá-lo da linha de comando a partir do mesmo diretório onde o executável se encontra (caso contrário ele não irá encontrar o diretório de texturas).

O programa não aceita argumentos de linha de comando, exceto possivelmente aqueles aceitos pela biblioteca *GLUT* através da sua inicialização *glutInit*.

Se o programa não mostrar nenhuma textura durante a sua execução, se assegure de que ele está sendo executado a partir do mesmo diretório em que se encontra o executável e o diretório das texturas. Se mesmo assim não funcionar, confira se a versão instalada dos drivers de vídeo suporta o hardware sendo usado (já que tivemos problemas com hardware antigo e drivers muito novos cujo suporte aquele hardware foi suspendido).

6 Comentários Finais

O primeiro comentário que gostaríamos de adicionar sobre o projeto é sobre a utilização de uma biblioteca adicional de manipulação de images. Usamos a biblioteca DevIL (Developer's Image Library), como salientado anteriormente na seção de requisitos.

A escolha dessa biblioteca facilitou e simplificou o código-fonte nas partes que envolvem imagens. Isso pois ela fornece um módulo (*ILUT*) que pode trabalhar completamente integrado com o OpenGL. Ao se carregar uma imagem, o DevIL se encarrega de enviá-la ao sistema gráfico e retorna apenas o descritor de textura do OpenGL. Ao criar esse descritor, também configura as propriedades da imagem adequadamente, como valores alfa sobre imagens que os possuem, entre outras coisas que aprendemos no próprio código-fonte da biblioteca.

Outra importante razão para a escolha dessa biblioteca é a impossibilidade de se usar a conhecida GLaux. De acordo com a documentação oficial do OpenGL, o seu uso é fortemente desaconselhado, já que a biblioteca foi abandonada muito tempo atrás e pode levar a problemas nos programas. Todas as suas funcionalidades são implementadas de forma muito mais adequada e eficiente em outras bibliotecas de utilitários gráficos, em especial a GLUT, ou em outras bibliotecas especializadas, como no DevIL, SDL, Allegro e outras.

Outro problema que encontramos foi ao tentar executar o nosso projeto em sistemas diferentes. Várias propriedades visuais mudavam de sistema para

sistema usando o mesmo código-fonte e imagens. Uma diferença foi na luminosidade da cena. Alguns mostravam a cena mais escura ou mais clara, mesmo sem usarmos qualquer iluminação na nossa cena. Também tivemos problemas menores de os painéis 2D aparecerem a distâncias diferentes da borda da janela, mesmo quando configuramos eles para terem posições e dimensões relativas ao tamanho da janela. E por último também tivemos o problema de que após termos desenvolvido todo o trabalho em Linux, ao tentarmos compilá-lo e executá-lo em ambiente Windows, todas as texturas apareciam invertidas sobre o eixo horizontal, ou seja, de cabeça para baixo. Exatamente o mesmo código-fonte e imagens foram testados em diversos sistemas diferentes e em todos os Linux o cenário era renderizado corretamente, mas em todos os Windows vimos o problema das texturas invertidas. Os problemas de luminosidade e dimensões dos painéis apareciam tanto nos Windows quanto nos Linux, mas em menor escala.

Por último, gostaríamos de lembrar do problema que tínhamos para acionar corretamente o teste de profundidade. Após alguns dias de experimentações frustradas, encontramos uma discussão em um fórum que dizia que o valor do plano próximo (z-near) não poderia ser zero. Não encontramos qualquer explicação sobre o motivo, mas alterando este valor para algo levemente maior (zero-ponto-um) fez com que o teste de profundidade funcionasse corretamente.

Antes dessa correção, a imagem era renderizada na ordem em que as primitivas eram enviadas ao OpenGL, o que gerava um resultado confuso e sem sentido, já que muitas das texturas dos planos escondidos eram mostradas na frente das texturas dos planos posteriores. Com a correção do valor z-near, a cena foi renderizada corretamente, escondendo as faces que deveriam estar atrás de outras e mostrando completamente aquelas que estavam na frente.