Projeto de MAC-211 — 2011
River Raid

Marco Dimas Gubitoso

2 de maio de 2014

Sumário

O objetivo deste projeto é o desenvolvimento de um programa composto de várias partes (módulos), ao longo do semestre.

Os módulos serão construídos ao longo de algumas fases (provavelmente 3) com uma fase adicional para a "amarração" e construção do programa final.

Serão consideradas a documentação e a organização do código, além do seu funcionamento. Os módulos devem ser tão independentes quanto possível e a comunicação entre dois módulos só deverá ser feita através de uma interface bem definida.

1 Descrição do jogo

Este projeto é baseado em um jogo antigo, para o Atari 2600, mas agora faremos uma versão bem mais moderna e em 3 dimensões. A dinâmica é relativamente simples, basta avançar pelo cenário, destruir bases inimigas e evitar os ataques.

A forma de pontuação e o nível de dificuldade ficará em aberto para cada grupo ajustar, no entanto é importante que o jogo tenha uma boa jogabilidade.

Os comandos poderão ser emitidos pelo teclado ou pelo *mouse*, isso será visto nas fases finais.

1.1 Ciclo de jogo

O funcionamento básico do jogo é um laço central que controla o avanço do tempo. Cada iteração corresponde a um *tick* do relógio e é chamada de *timestep*.

Em um *timestep*, o programa deve fazer várias operações em sequência:

1. atualizar as posições da nave e dos projéteis (veja abaixo), além de eventuais objetos animados que você queira incluir.

2. verificar colisões e alterar o estado do jogo de acordo. Isto inclui atualizar a pontuação, verificar o final, etc.

3. desenhar o cenário visível de acordo com o novo estado.

Não faremos tudo isso de uma vez. Você deve ter notado, por exemplo, que não citei a entrada de comandos, ela será feita de modo independente, usando uma técnica de tratamento de eventos. Estes casos serão tratados por funções especiais[1] que serão chamadas a cada vez que o usuário quiser interagir com o sistema. Isto só será feito na terceira fase, quando incluirmos a parte gráfica.

[1]Estas funções são chamadas de *callback* e estão associadas a eventos específicos

2 Primeira fase

Nesta fase montaremos os elementos básicos do jogo. Não é necessário cuidar da dinâmica, apenas da construção da nave, do cenário essencial e das torres de ataque.

O seu programa deverá ter vários módulos, cada um cuidando de um tipo destes elementos. A nave é única, mas podem haver diversos canhões por exemplo. O número e variedade de tipos também é livre, mas pelo menos os seguintes devem estar presentes:

- **Nave** — é controlada pelo jogador, deve estar presente enquanto o jogo dura.
- **Canhão** — espalhados ao longo do cenário, tentam atingir a nave e destruí-la.
- `Projétil` — disparados pela nave ou pelos canhões, podem ser parametrizados para especificar tamanho, potência, velocidade, etc.

Outros elementos, como pontos de reabastecimento ou recuperação da nave, unidades inimigas diferentes e elementos passivos podem ser colocados e contarão bônus.

As seções seguintes descrevem os módulos que deverão ser programados nesta fase.

Estes elementos ganharão uma descrição gráfica mais à frente, mas isto não é importante neste momento. Apenas tenha consciência que os módulos correspondentes sofrerão alterações, para incluir estas características.

2.1 Nave.c

A nave possui alguns atributos básicos, que podem ser agrupados em uma `struct`:

- altura — é a distância com relação ao solo, como o nome diz.
- posição horizontal — contada a partir do plano central do cenário, valores negativos representam o lado esquerdo.
- posição ao longo do cenário — essencialmente a distância percorrida pela nave, considerando apenas o eixo que "entra" na imagem.

- velocidade — escalar, valor absoluto.
- orientação — indica a direção em que a nave aponta.
- quantidade de dano da nave

A velocidade e orientação serão utilizadas para calcular a nova posição da `nave`, no próximo ciclo de iteração.

Outros valores podem ser colocados para enriquecer o jogo, como por exemplo quantidade de combustível ou de armamentos.

2.2 Defesa.c

Os elementos de defesa podem ser de diversos tipos, mas todos possuem uma posição fixa em relação ao cenário. Os atributos são:

- posição em 3 dimensões
- precisão do tiro
- quantidade de dano sofrida
- parâmetros de ataque

Os parâmetros de ataque servirão para decidir quando um tiro deverá ser disparado. Eles podem indicar a frequência, a reação com a proximidade da `nave`, etc.

2.3 Tiro.c

Um `tiro`, ou projétil, possui uma descrição parecida com a da `nave`, mas não tem dano, nem recebe comandos. Ele apenas se desloca pelo cenário até atingir um alvo (incluindo o chão) ou sair do espaço representado.

2.4 Cenario.c

Este módulo representa o cenário central. O formato exato dependerá da sua decisão do jogo. Você pode escolher um rio ou um corredor.

O cenário será representado por uma lista de elementos passivos e/ou de defesa. Pode ser um vetor ou uma lista ligada. Lembre-se que o cenário

é dinâmico e deverá mudar à medida em que a nave avança. Os objetos ultrapassados pela nave deixarão de participar do jogo e sua representação pode ser abandonada, poupando espaço de memória.

Se usar um vetor, ele precisará ser atualizado a cada vez que a nave passar por um de seus elementos. Isto pode ser feito com o deslocamento de todos os elementos em uma posição, liberando uma posição no final que será preenchida por um novo item. Isto no entanto é caro, é mais fácil fazer um arranjo circular, como nos *buffers*.

No caso de uma lista, basta fazer a cabeça apontar para o segundo elemento, liberar o antigo primeiro elemento e adicionar um novo na cauda. É preciso tomar bastante cuidado com a manipulação de ponteiros e com a alocação e liberação de memória.

2.5 Programa parcial

Não se preocupe com desenhos para esta fase. Estamos apenas montando a estrutura. Você deve fazer um programa que monte o cenário, colocando os elementos e a nave. Cuide para que os elementos possam ser gerados e incluidos e removidos dinamicamente (em tempo de execução). Tome muito cuidado com `malloc`s e `free`s.

Seu programa principal deve testar todas as funções criadas e escreva um relatório sobre as decisões tomadas e dificuldades encontradas. Se você não conseguiu fazer alguma coisa, deixe claro quais foram as tentativas e o que aconteceu quando falhou. Isto garantirá uma recuperação da nota na fase seguinte, caso consiga arrumar.

Uma dica é ir escrevendo as decisões à medida em que forem tomadas,o mesmo vale para experimentos e resultados. Isso não apenas ajuda a entender melhor o processo, como facilita em muito a confecção do relatório final. Com isso, você tem um *log* do desenvolvimento que pode ser consultado a qualquer momento para tirar dúvidas ou avaliar seu planejamento. Este procedimento é muito usado na vida profissional de desenvolvedores, tanto no meio científico, como no meio empresarial.

3 Segunda fase

É nesta fase que montaremos a dinâmica do jogo. Essencialmente será implementado o laço de execução, que atualiza o estado e posição de todos os

objetos, verifica e trata colisões e, *para fins de testes*, tratar ações do usuário.

Em princípio, seria possível jogar ao final desta fase, mas ainda não teremos a visualização.

3.1 Laço de execução

O coração do jogo (e todos os jogos deste tipo) é o laço central de simulação. É um laço contínuo que é executado enquanto o jogo está ativo.

A cada passo, o cenário é atualizado em uma unidade de temo que chamaremos de *timestep*. O valor de um *timestep* determinará a velocidade de atualização e depende da potência do computador e, quando incluirmos a terceira fase, de sua placa gráfica. Ele pode ser uma variável de configuração, cujo valor é apresentado na linha de comando.

3.2 Atualização do estado

A cada *timestep*, o programa deve chamar uma rotina de atulização para cada elemento do jogo. Isto é, para cada tipo de elemento, precisamos escrever uma função que muda seu estado, de acordo com a situação do jogo. Esta função deve estar definida no módulo que define o objeto.

A função de atualização deve calcular e atualizar os seguintes valores, quando cabível:

- Nova posição do objeto, usando sua posição atual, orientação e velocidade.
- Nova orientação do objeto.
- Identificar colisão.
- Atualizar estragos (dano).

Caso o objeto tenha sido destruído, ele deverá ser removido do jogo. Para a nave, o jogador perde uma vida e "ganha" uma nave nova. Com 0 vidas o jogo é encerrado.

Veja alguns exemplos de atualizadores.

3.2.1 Nave

Todos os valores calculados devem ser atualizados na estrutura de dados.

- Calcula a nova posição
- Verifica a entrada do usuário para saber se houve mudança de orientação
- Verifica se deve disparar um tiro e toma as providências necessárias.
- Verifica se houve colisão e toma as providências necessárias.

Se quiser, pode colocar regiões especiais para recarregar munição, combustível e saúde. É relativamente fácil, depois de montado o mecanismo completo.

3.2.2 Projétil

É muito parecido com a nave, mas deve levar em conta o efeito da gravidade.

No caso de colisão, além de danificar o outro objeto, o projétil deve ser destruído.

Pode haver diversos tipos de projétil, fica a critério de cada grupo.

3.2.3 Unidades de defesa

São as torres e os outros objetos fixos. Como não se movem[2], as atualizações se referem a teste de colisão e á decisão se devem atirar ou não.

Para decidir se atira, a unidade verifica a distância da nave e usa um valor aleatório para determinar se o disparo ocorre.

Se uma unidade ficar atrás da nave (for ultrapassada) ela é eliminada do jogo, sem contar pontos.

3.3 Teste de colisões

O teste de colisões pode ser bastante complicado, se quisermos ser precisos.

Uma das formas de verificar se existe colião entre dois objetos A e B é verificar se existe um vértice de A dentro de B e vice-versa. Este procedimento é muito custoso e complicado.

[2]você pode incluir objetos móveis, se quiser. Vale bônus.

Uma forma mais simples é verificar se os objetos estão próximos o suficiente: se as esferas minimais que englobam cada objeto se tocam. Este método é muito simples. Como a maior parte dos objetos está arrumado por distância, podemos parar de fazer comparações mais cedo.

Para quem preferir algo mais preciso, ao invés de esferas, pode-se usar cilindros. Basta um pouquinho mais de álgebra linear (distância de ponto a reta, ou uso adequado de projeções) e a conta final é relativamente simples.

3.4 Ações do usuário

No modo normal de jogo, precisaríamos de *threads*: várias linhas de execução concorrentes. Isto porque a entrada de dados do jogador ocorre ao mesmo tempo em que o cenário é atualizado.

Este problema será resolvido automaticamente pelo *OpenGL*, por enquanto apenas leremos caracteres da entrada padrão para comandar a a nave e os disparos dos tiros.