Especificação do Trabalho Prático 1
Técnicas de Programação 2

[1]Departamento de Ciência da Computação
Universidade de Brasília (UnB)

Entrega: 07/10/2017, 23:59h

1. Introdução

Desert Falcon é um jogo do console Atari, presente tanto em sua versão 2600, 7800 e nos computadores 8-bit da Atari. É um jogo relativamente simples, um dos primeiros a, de fato, simular um ambiente tridimensional. Dentro do jogo, o jogador controla um falcão, cujo o objetivo é coletar hieróglifos e desviar de inimigos. Para o que não conhecem o jogo, podem ver uma execução dele clicando nesse vídeo do jogo no YouTube

Figura 1. Screenshot do jogo Desert Falcon no Atari 2600

No projeto a ser desenvolvido, deve-se implementar uma versão simplificada do jogo, seguindo os conceitos apresentados na disciplina. De acordo com a linguagem de programação escolhida, deve utiizar ou a biblioteca SDL para desenvolvimento em C ou C++ `http://www.sdltutorials.com/sdl-tutorial-basics`, ou a biblioteza Gosu para desenvolvimento Ruby `https://github.com/gosu/gosu/wiki/ruby-tutorial`. Os componentes a serem desenvolvidos são: *a janela do jogo* e o *GameObject* – a representação de uma entidade no jogo. Além disso, devem haver especializações de GameObject, sendo essas a entidade controlada pelo input do jogador – **Falcon** – e a entidade que representa o objeto que deve ser coletado – **Hiero**.

2. Dinâmica do Jogo

Nessa versão, o Desert Falcon deve ter uma estrutura capaz de guardar qualquer quantidade de entidades (GameObjects) dentro do mundo do jogo. Dentro do loop principal do jogo, o qual deve ser repetido até a saída ser requisitada (apertar a tecla ESC, por exemplo), deve-se percorrer a estrutura atualizando cada entidade, para então a percorrer novamente checando se houve colisão entre duas entidades. Se sim, executar o que deve acontecer ao haver uma colisão entre os tipos específicos de entidade. Deve-se também percorrer a estrutura desenhando(ou renderizando) as imagens de cada entidade.

Por sua vez, o falcão, entidade controlada pelo jogador, deve-se mover na diagonal, em duas direções: cima-esquerda e baixo-direita. O falcão não pode ultrapassar os limites da tela, ou seja, sempre que fosse se mover para uma posição com coordenadas negativas ou maiores que o tamanho da tela, sua posição deve se manter a mesma. Vale lembrar que no plano cartesiano tanto da SDL quanto da Gosu a cordenada y cresce para baixo. O falcão deve também ser capaz de mudar sua altura, sendo que começa na altura mediana, podendo subir ou descer uma altura. Ou seja, devem-se haver três níveis de altura, representada por uma alteração na coordenada y do falcão. O falcão nunca deve ser excluído da estrutura que contém as entidades.

Já o Hiero deve mover na diagonal, seguindo para o canto inferior-esquerdo da tela, ou seja, para a esquerda-baixo. Dessa forma, haverá a sensação de que o falcão está se movendo, e que os Hieros estão passando por ele. A condição para um Hiero ser excluído da estrutura que contém as entidades é: colidir com o falcão, ou ao estar em coordenadas que estejam fora da tela.

3. Componentes da Etapa 1 do Jogo

Como mencionado, nessa primeira entrega do trabalho, dois componentes deverão ser implementados: Janela e GameObject, detalhados a seguir. Lembrem-se que para cada componente e seus respectivos módulos, devem-se criar suas interfaces, implementado por meio dos modulos de declaração e de implementação, evitando alto acoplamento e baixa coesão. Além disso, devem ser respeitados os demais conceitos de encapsulamento quanto ao escopo das variáveis.

3.1. Componente 1 – Janela do Jogo

Essa componente será responsável pelo início, execução, renderização e finalização do jogo. Assim, esse componente deve possuir: (1) um método de inicialização, (2) um método a ser executado em cada iteração do loop de execução e (3) um método para desenhar (renderizar) o que está sendo processado no loop principal e (4) um método para finalizar o jogo.

Uma vez que este componente encerra em si funções não muito complexas, não há necessidade de organizá-los em diferentes módulos.

3.1.1. Funções a serem implementadas

```
public:
  construtor(largura : int, altura : int);
  destrutor();
  update();
  render();
```

3.2. Componente 2 – GameObject

Esse componente representa todos os objetos presentes no jogo. Nessa etapa do trabalho, serão implementados como objetos do jogo o *Falcon* e o *Hiero*. Dessa forma, este componente possuirá métodos e atributos relacionados a propriedades de um objeto num mundo virtual, como posição X e Y (largura e altura), um nome, e por fim, uma imagem.

3.2.1. Modularização

Esse componente deverá possuir os 3 seguintes módulos:

- **Módulo Sprite:** Este módulo deve ser guardar a imagem e de manipulá-la quando necessário. Cada entidade do jogo que herda de GameObject terá sua própria imagem.
- **Módulo Box:** Este módulo deve guardar a posição X e Y da entidade, além de sua largura e altura. Deve possuir métodos para checar a intersecção entre duas entidades.
- **Módulo GameObject:** Este módulo deve possuir métodos que sejam comuns a qualquer entidade, como, por exemplo, ser capaz de desenhar a imagem associada a si próprio e detectar quando há uma colisão.

3.2.2. Entidades

Caso esteja utilizando orientação a objetos, Falcon e Hiero deve ser uma especialização da classe do módulo Objeto. Caso esteja utilizando C, os módulos Falcon e Hiero devem incluir a interface de Sprite e Box, e implementar como módulo Objeto cada um.

3.3. Interfaces a serem implementadas

Interface do Sprite:

```
public:
  construtor(nomeDoArquivo : string)
  destrutor()
  render()
```

Interface do Box:

```
public:
  construtor(x : int, y : int, w : int, h : int)
  destrutor()
  overlapsWith(other : GameObject) : bool
```

Interface do módulo GameObject:

```
public:
  construtor(x : int, y : int, z : int)
  destrutor()
  update()
  render()
  isEqual(other : GameObject) : bool
  isDead() : bool
  notityCollision(other : GameObject) : bool
```

Interface de Falcon:

```
public:
  construtor(x : int, y : int, z : int)
  update()
  notityCollision(other : GameObject) : bool
```

Interface de Hiero:

```
public:
  construtor(x : int, y : int, z : int)
  update()
  notityCollision(other : GameObject) : bool
```

Lembre-se de utilizar as funções de acesso (obter/get e atribuir/set) para a manipulação das variáveis encapsuladas.

4. Controle de Qualidade das Funcionalidades

Depois de pronto, deve-se criar um módulo controlador de teste (disciplinado) usando a biblioteca do **bdd-for-c** (para C), **RSpec** (para Ruby) ou framework do **Google Test** (para C++ em `https://github.com/google/googletest`) para testar se as principais funcionalidades e restrições dos módulos de armazenamento, tratamento e persistência dos dados atendem a especificação. O teste disciplinado deve seguir os seguintes passos:

1. Antes de testar: produzir um roteiro de teste.
2. Antes de iniciar o teste: estabelecer o cenário do teste.
3. Criar um módulo controlador de teste para testar as principais funcionalidades de cada módulo.
4. Ao testar: produzir um laudo em que todas as discrepâncias encontradas são registradas. Esse laudo pode ser uma saída da execução da suíte de teste. Somente termine o teste antes de completar o roteiro, caso observe que não vale mais a pena continuar executando o roteiro, uma vez que o contexto para o resto está danificado

Após a correção: repetir o teste a partir de 2 até o roteiro passar sem encontrar falhas.

5. Documentação e Entrega do Trabalho

Visando viabilizar o trabalho em grupo, deve-se utilizar o repositório GitHub para todo o ciclo de desenvolvimento, para interação e gerenciamento de todo o desenvolvimento. Vide slides introdutórios sobre comandos do git no Aprender da disciplina de Técnicas de Programação 2 (tópico 5).

A submissão do trabalho deve estar compactada com o seguinte conteúdo:

1. Todos os arquivos necessários para a compilação e execução do programa, incluindo os módulos controladores de teste.
2. Um ReadMe.txt contendo:
    - As instruções para a compilação e execução correta do trabalho. Lembrem-se que o programa será avaliado em uma distribuição Linux.
    - O link do projeto no GitHub.
3. Uma modelagem conceitual e outra modelagem física contemplando todos os componentes e respectivos módulos.
4. Os gráficos gerados pelo GitHub com as estatísticas do desenvolvimento do projeto contendo as horas trabalhadas por cada membro do grupo e as respectivas descrições das tarefas que cada membro realizou.

Observações importantes:

- Entregas que não contiverem todas as bibliotecas/dependências necessárias para a compilação e execução não terão o programa avaliado.
- Cada dia de atraso na entrega corresponde a um ponto a menos da nota final do trabalho.
- Trabalhos plagiados serão atribuídos sumariamente nota 0.
- O algoritmo da correção é o seguinte:

```
float nota = 0;
if (plágio)
    nota =0;
    exit(1);

if (compila)
    nota += 0.5;
if (funciona)
    then if (funciona_com_erros)
        nota += 1;
    else
        nota  +=2;

  switch (modularização){
   case "Janela":
            nota += 1.0;
   case "GameObject":
            nota += 1.0;
   case "Falcon":
            nota += 0.75;
   case "Hiero":
            nota += 0.75;
};

 for (modulo = 0; modulo < 4; modulo++)
        if (teste(modulo))
            nota += 0.25;

 assert(nota==7); //nota relativa à parte de implementação

 switch (documentação){
   case "GitHub \& ReadMe \& relatorioTarefas":
            nota += 1;
   case "ModeloConceitual":
            nota += 1;
   case "ModeloFisico":
            nota += 1;
};

assert(nota == 10); //nota total incluindo a parte de documentação
```

BOM TRABALHO!!!!