Especificação do Trabalho Prático 2
Técnicas de Programação 2

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

Entrega: 17/11/2017, 23:59h

1. Introdução

Essa etapa do desenvolvimento do jogo Desert Falcon deve estender as funcionalidades da primeira especificação do trabalho onde os componentes a serem desenvolvidos nesta etapa do trabalho são novos *GameObjects*, sendo eles os *obstáculos* e os *inimigos*, assim como a contabilização da pontuação no jogo. Adicionalmente, deve ser implementado um *menu principal* permitindo a visualização do *placar* de outras partidas, registrado por meio das *pontuações* obtidas em cada partida.

Figura 1. Screenshot do jogo Desert Falcon no Atari 2600

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 biblioteca Gosu para desenvolvimento Ruby `https://github.com/gosu/gosu/wiki/ruby-tutorial`.

2. Dinâmica do Jogo

Nessa versão, o Desert Falcon possuirá obstáculos, sendo que esses obstáculos seguem uma lógica de movimentação parecida com a do Hiero. O obstáculo deve mover na diagonal, seguindo para o canto inferior-esquerdo da tela, ou seja, para a esquerda-baixo. O objetivo é gerar a sensação de movimentação do falcão. É necessário que os obstáculos estejam na mesma velocidade que os Hieros para o efeito ser adequado. Para não mantermos para sempre um obstáculo não mais visível em nossa estrutura de entidades, a

condição para o obstáculo ser excluído é sua origem estar em coordenadas fora da tela. Um obstáculo só aparece no nível de vôo(do falcão) mais baixo.

Enquanto isso, o inimigo possui uma lógica praticamente igual ao dos Hieros e obstáculos, exceto que sua velocidade é maior, assim gerando a sensação de que não somente o falcão está se movendo como o inimigo também, só que na direção contrária. Da mesma forma que o obstáculo, a condição para o inimigo ser excluído da estrutura de entidades é sua origem estar em coordenadas fora da tela. O inimigo pode aparecer em qualquer nível de vôo(do falcão).

Além disso, a colisão deve agora levar em conta o nível de cada entidade, de forma que uma entidade em um nível X não colide com uma entidade em um nível X+1 ou X-1.

Por sua vez, a pontuação deve ser implementada de forma que, quando o falcão colidir com um Hiero, o valor dessa pontuação irá ser incrementado. Após perder o jogo, o jogador pode digitar um nome de até três caracteres, e então o nome, em conjunto com a pontuação, será salvo em um arquivo texto. Esse arquivo texto deve ser lido quando a opção ”Placar”for selecionada no menu, e as dez maiores pontuações devem ser exibidas em ordem na tela.

Sempre que o falcão colidir contra um obstáculo ou inimigo, o jogador perde o jogo. Após ter sua pontuação salva como descrito acima, o jogador deve ser levado de volta ao menu principal.

Quanto ao menu principal, este deve possuir as seguintes opções: ”Jogar”, ”Placar”e ”Sair”. A organização destas opções na tela, assim como o método de seleção deles(mouse ou teclado) ficam a critério da equipe de desenvolvimento.

3. Componentes da Etapa 2 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

Além de possuir todas as funcionalidades implementadas no trabalho 1, a Janela do jogo deve ser capaz de processar a lógica de telas diferentes além da do jogo e de renderizar as mesmas. Uma vez que este componente encerra em si funções não muito complexas, não há necessidade de organizá-los em diferentes módulos. Segue o algoritmo sugerido para interação entre essas funcionalidades:

```
#Na criação da janela, defina os estados, começando no menu

typedef enum state {MENU, JOGO, PLACAR, PONTO};

state = MENU;

#No update coloque a lógica para cada estado e para suas transição
```

```
switch(estado){
 case MENU:
     // executa a logica do menu.
        if(ir para jogo)
                estado = JOGO;
        if(ir para placar)
                estado = PLACAR;
        break;
    case JOGO:
    // executa a lógica do jogo
        if(jogador perdeu)
                estado = PONTO;
        break;
    case PONTO:
    // executa a lógica de atualizar placar
        if(pontuação adicionada)
                estado = MENU;
        break;
    case PLACAR:
     // executa a lógica para exibir placar
        if(voltar ao menu principal)
                estado = MENU;
        break;
};
```

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. Deve possuir todas as funcionalidades já implementadas na etapa anterior do projeto e novas entidades devem ser introduzidas, no caso, Enemy e Obstacle.

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
```

Interfaces de Falcon, Hiero, Enemy e Obstacle:

```
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.

3.4. Componente 3 – Pontuação

Este componente deve ser capaz de ler e escrever de arquivos, além de possuir métodos para desenhar texto contendo as informações lidas e de pegar input do usuário.

3.4.1. Modularização

Esse componente deverá possuir os 2 seguintes módulos:

- **Módulo Escrita:** Este módulo tem a principal função de escrever em um arquivo o nome recebido por uma input do usuário seguido da pontuação da respectiva partida.
- **Módulo Leitura:** Este módulo deve ler o arquivo de pontuação e desenhar na tela as 10 maiores pontuações presentes no arquivo.

4. Análise Estática

Deve-se utilizar o Splint (para C) ou Rubocop (para Ruby) para avaliar as faltas de interface, dados, entrada/saída, controle e armazenamento. Deve-se assim aderir a padrões de documentação e eliminar faltas identificadas previamente por meio do processo de análise estática. Caso seja feito em C, seu programa deve compilar sem *warnings* com os comandos -weak +compdef +infloops +sysdirerros

4.1. Documentação do Código

1. Para tornar o código mais organizado e legível, o grupo deve-se adotar um padrão de documentação.
2. Elaborem um arquivo .pdf que contenha explicações e exemplos do padrão adotado pelo seu grupo.
3. Faça a documentação e comentários do código (funções, variáveis, TADs, módulos e suas interfaces) utilizando a ferramenta Doxygen e gere em formato .pdf ou .html.
4. Produzir o grafo de chamada de cada função de cada módulo utilizando as funções de geração de chamada de grafo (callgraph) no Doxygen. Para tanto, instalem o graphviz (sudo apt-get install graphviz). E, no arquivo de configuração gerado pelo Doxygen, configurem os seguintes parâmetros:

   HAVE_DOT = YES
   EXTRACT_ALL = YES
   EXTRACT_PRIVATE = YES
   EXTRACT_STATIC = YES
   CALL_GRAPH = YES

   Nas pastas html e latex geradas pelo Doxygen conterão, juntamente com a documentação do código gerado, os grafos produzidos automaticamente pelo Doxygen!

5. 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, usando a ferramenta CUnit 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.

6. 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 contém todas as bibliotecas necessárias para 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 += 0.75;
   case "GameObject Anteriores":
            nota += 0.75;
   case "Inimigos":
            nota += 0.75;
   case "Obstáculos":
            nota += 0.75;
   case "Pontuação":
            nota += 0.5;
};

 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 \& Doxygen":  //rep. do github & do doxygen
            nota += 1;
   case "Análise Estática Corrigida": //faltas identificadas e endereçadas
            nota += 1;
   case "Padrão de Documentação": //classes/módulos, métodos/funções, vars e const
            nota += 1;
};

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

BOM TRABALHO!!!!