Object-Oriented Programmed Snake

Data: 21/05/2024
Grupo: 3
Elementos: Diogo Fonseca nº 79858

Utilização

A maneira mais fácil de configurar os parâmetros de inicialização é usando o GameManagerBuilder e chamando o método play() do GameManager.

No modo stepped (definindo o update method como STEP), os updates são feitos iterativamente conforme o utilizador pressiona a tecla enter. Para parar a execução do programa basta digitar o input “stop” ou com Ctrl+C no modo gráfico.

Algo importante a notar é que nem todos os terminais interpretam códigos ANSI de cor, logo dependendo do terminal usar cores pode dar um output completamente desfigurado. Para evitar isto basta não definir nenhuma cor.

A classe Main tem algumas funções predefinidas com algumas configurações interessantes para o jogo, sendo apenas necessário descomentar a função desejada. Em seguida estão as funções na classe Main com uma visualização do output da configuração gerada por elas.

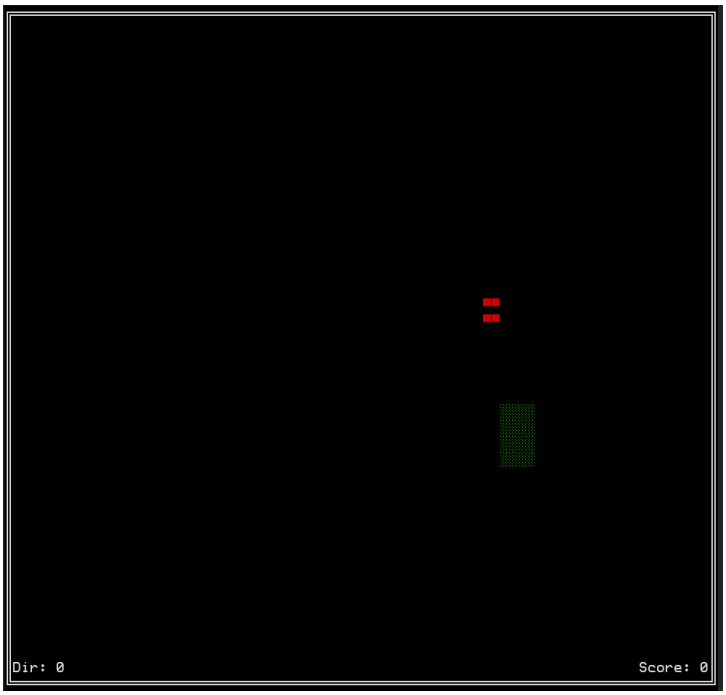
- **Default Example**

É gerado um mapa de 80x40 unidades, onde o tamanho da cobra é 4 e o tamanho da comida é 2. Cada comida é quadrada e dá 5 pontos quando consumida. O método de controle é manual, sendo o utilizador a controlar a cobra e a cobra poderá ser controlada pelas teclas ‘w’, ‘a’, ‘s’ e ‘d’ (outras opções de movimentação são VIM, ABSOLUTE, RELATIVE (e ARROW_KEYS para a representação gráfica)).
Tanto a cobra como a comida são geradas aleatoriamente.

```
private static void defaultExample(long seed) throws Exception
{
    new GameManagerBuilder()
        .setSeed(seed)
        .setTextual(true)
        .setFilled(true)
        .setMapWidth(80)
        .setMapHeight(40)
        .setSnakeSize(4)
        .setFoodSize(2)
        .setFoodScore(5)
        .setInputPreset(InputPreset.WASD)
        .setFoodType(GameManager.FoodType.SQUARE)
        .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
        .setControlMethod(GameManager.ControlMethod.MANUAL)
        .build();
}
```

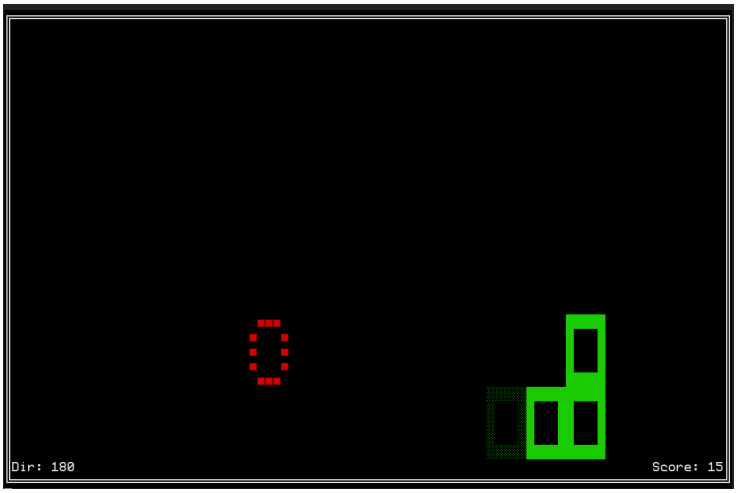
- **Colour Example**
  É gerado um exemplo análogo ao anterior, mas em cores e caracteres diferentes.

```
private static void colourExample(long seed) throws Exception
{
    new GameManagerBuilder()
        .setSeed(seed)
        .setTextual(true)
        .setFilled(true)
        .setMapWidth(80)
        .setMapHeight(40)
        .setSnakeSize(4)
        .setFoodSize(2)
        .setFoodScore(5)
        .setInputPreset(InputPreset.WASD)
        .setFoodType(GameManager.FoodType.SQUARE)
        .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
        .setControlMethod(GameManager.ControlMethod.MANUAL)

        // setting colors
        .setBackgroundColour(Colour.Background.BLACK)
        .setSnakeColour(Colour.Foreground.GREEN)
        .setFoodColour(Colour.Foreground.RED)
        .setObstaclesColour(Colour.Foreground.MAGENTA)

        // setting drawing characters
        .setMapChar(' ')
        .setSnakeHeadChar('▒')
        .setSnakeTailChar('▓')
        .setObstacleChar('▓')
        .setFoodChar('■')
        .build();
}
```



- **Circle Example**
  Neste exemplo é gerado uma comida circular, com o mesmo tamanho da cobra e nenhuma figura é preenchida.

```
private static void circleExample(long seed) throws Exception
{
    new GameManagerBuilder()
        .setSeed(seed)
        .setTextual(true)
        .setFilled(false)
        .setMapWidth(90)
        .setMapHeight(30)
        .setSnakeSize(5)
        .setFoodSize(5)
        .setFoodScore(5)
        .setInputPreset(InputPreset.WASD)
        .setFoodType(GameManager.FoodType.CIRCLE)
        .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
        .setControlMethod(GameManager.ControlMethod.MANUAL)

        // setting colors
        .setBackgroundColour(Colour.Background.BLACK)
        .setSnakeColour(Colour.Foreground.GREEN)
        .setFoodColour(Colour.Foreground.RED)
        .setObstaclesColour(Colour.Foreground.MAGENTA)

        // setting drawing characters
        .setMapChar(' ')
        .setSnakeHeadChar('▒')
        .setSnakeTailChar('▓')
        .setObstacleChar('▓')
        .setFoodChar('■')
        .build();
}
```



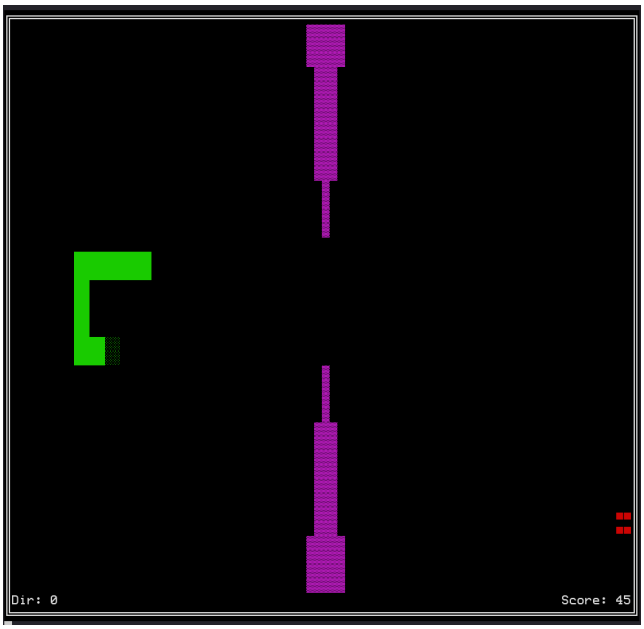
NOTA: com comida circular, em certas ocasiões (tamanho da comida par) pode dar o efeito visual de que a comida é maior do que ela realmente é, pois o renderer ao desenhar o círculo, vai obter valores reais para onde desenhar os pontos, e ao passar esses valores para um espaço discreto, pode “aumentar” o tamanho do círculo (mais especificamente, porque com tamanho par as coordenadas vão acabar em .5 e serão arredondadas para cima). Nestes cenários isto é só um efeito visual e a comida na realidade tem o tamanho correto. Estes casos são limitações tanto do algoritmo midpoint como do bresenham’s circle algorithm (que na verdade é só um midpoint modificado) (por outras palavras nenhum deles consegue desenhar um círculo com altura/comprimento par).

Normalmente isto não se notaria porque uma coordenada do raster é muito pequena, mas no caso do jogo da cobra representado textualmente, cada coordenada do raster é gigante e então o círculo poderá parecer ligeiramente maior do que as suas proporções reais nestes casos.

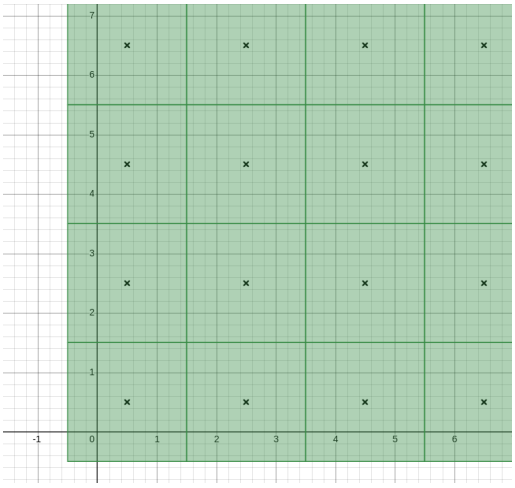
- **AI Example**

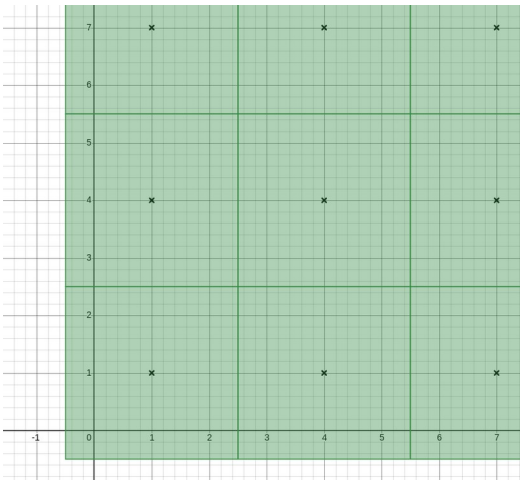
Este exemplo é mais complexo, instanciando dois obstáculos estáticos, usando o modo AUTO para a cobra ser controlada autonomamente, a cobra e a comida também são instanciadas com uma determinada posição (e a cobra com direção) ao invés de ser aleatório.

```
Triangle obstacle0 = new Triangle(new Point[] {
    new Point(38, 40),
    new Point(42, 40),
    new Point(40, 25),
});

Triangle obstacle1 = new Triangle(new Point[] {
    new Point(38, 0),
    new Point(42, 0),
    new Point(40, 15),
});

new GameManagerBuilder()
    // Global setup
    .setSeed(seed)
    .setTextual(true)
    .setFilled(true)
    .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
    .setControlMethod(GameManager.ControlMethod.AUTO)

    // Obstacle setup
    .addObstacle(obstacle0)
    .addObstacle(obstacle1)

    // Map setup
    .setMapWidth(80)
    .setMapHeight(40)

    // Snake setup
    .setSnakeSize(2)
    .setSnakePos(new Point(60.5, 20.5))
    .setSnakeDir(Direction.LEFT)

    // Food setup
    .setFoodSize(2)
    .setFoodScore(5)
    .setFoodPos(new Point(4.5, 4.5))
    .setFoodType(GameManager.FoodType.SQUARE)

    // Colour setup
    .setBackgroundColour(Colour.Background.BLACK)
    .setSnakeColour(Colour.Foreground.GREEN)
    .setFoodColour(Colour.Foreground.RED)
    .setObstaclesColour(Colour.Foreground.MAGENTA)

    // Character setup
    .setMapChar(' ')
    .setSnakeHeadChar('▒')
    .setSnakeTailChar('▒')
    .setObstacleChar('▒')
    .setFoodChar('■')

    .build();
```



NOTA: pode ser difícil escolher uma posição válida para a cobra pois a definição de posição válida varia com o tamanho da cobra de uma maneira não intuitiva. A primeira posição válida da cobra é dada pelo centroide do primeiro quadrado: eis dois exemplos visuais de posições válidas da cobra (marcado pelos pontos a preto).



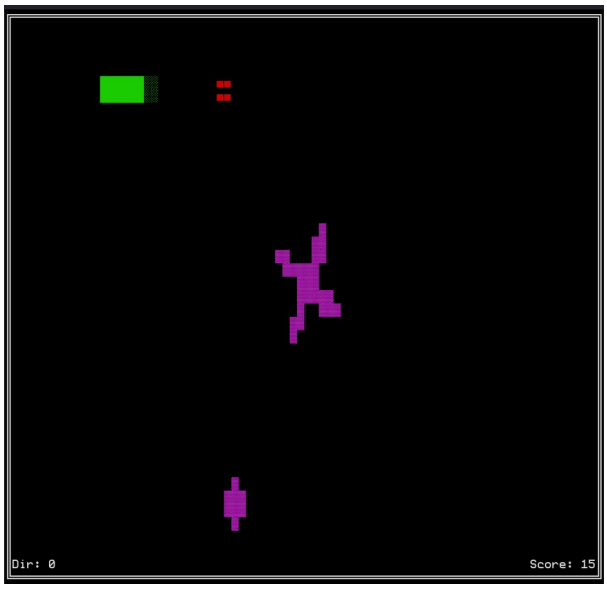


Cobra de tamanho 2 Cobra de tamanho 3

- **Dynamic Example**

Do ultimo exemplo para este, as diferenças são que foi definido um máximo de scores (10), e adicionou-se dois obstáculos dinâmicos ao invés de estáticos. Um vai rodar em torno de um centro, e o outro vai rodar em torno do seu centroide (pois o centro de rotação dado é nulo).

```
Polygon obstacle0 = new Polygon(new Point[] {
    new Point(40, 24),
    new Point(41, 21),
    new Point(44, 20),
    new Point(41, 19),
    new Point(40, 16),
    new Point(39, 19),
    new Point(36, 20),
    new Point(39, 21),
});
float speed0 = -0.3f;


Square obstacle1 = new Square(new Point[] {
    new Point(39, 38),
    new Point(39, 40),
    new Point(41, 40),
    new Point(41, 38),
});
Point rotationPoint1 = new Point(40, 20);
float speed1 = 0.25f;

new GameManagerBuilder()
    // Global setup
    .setSeed(seed)
    .setTextual(true)
    .setFilled(true)
    .setMaxScoresDisplay(10)
    .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
    .setInputPreset(InputPreset.WASD)
    .setControlMethod(GameManager.ControlMethod.MANUAL)

    // Obstacle setup
    .addObstacle(obstacle0, null, speed0)
    .addObstacle(obstacle1, rotationPoint1, speed1)
```



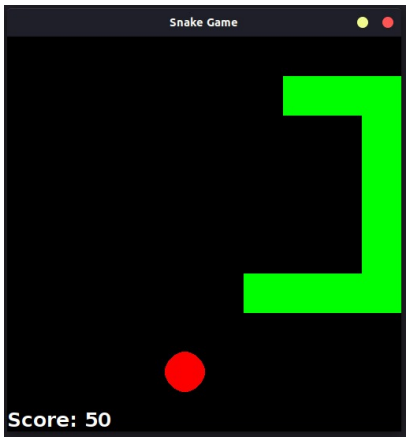
- **Graphic Example**

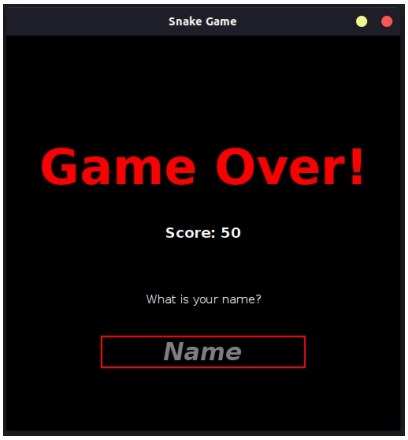
É gerado um mapa de 500x500, com uma cobra de 50x50 e uma comida (circular) de 50x50. O modo não é textual (é gráfico), as teclas para movimentação da cobra são as setas no teclado, e o modo de atualização é STEP. O modo STEP com interface gráfica atualiza o jogo **sempre que uma tecla é pressionada**, mesmo que a tecla não seja um input reconhecido pela cobra, neste exemplo, mesmo que o input não seja as setas no teclado, ao pressionar uma tecla, o jogo atualizará.

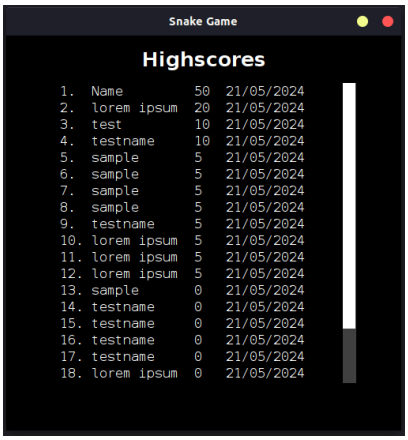
NOTA: as cores da janela gráfica estão separadas das cores definidas para o terminal.

NOTA: o modo de input ARROW_KEYS funciona somente para o modo gráfico.

```
new GameManagerBuilder()
    .setSeed(seed)
    .setTextual(false)
    .setFilled(true)
    .setMapWidth(500)
    .setMapHeight(500)
    .setSnakeSize(50)
    .setFoodSize(50)
    .setFoodScore(5)
    .setInputPreset(InputPreset.ARROW_KEYS)
    .setFoodType(GameManager.FoodType.CIRCLE)
    .setUpdateMethod(GameEngineFlags.UpdateMethod.STEP)
    .setControlMethod(GameManager.ControlMethod.MANUAL)

    // setting colors
    .setGraphicalBackgroundColour(Color.black)
    .setGraphicalSnakeColour(Color.green)
    .setGraphicalFoodColour(Color.red)
    .setGraphicalObstaclesColour(Color.magenta)

    .build();
```





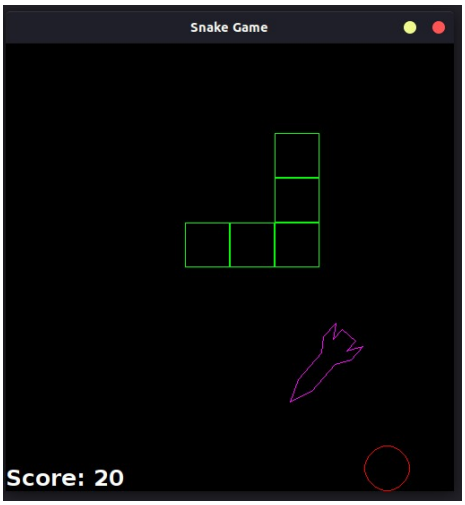


- **Autoupdate Example**

Neste exemplo nada é preenchido. Para além disto o update é feito autonomamente, 1 vez por segundo no máximo. É também introduzido um obstáculo dinâmico.

NOTA: o modo de update automático não foi feito para o modo textual, embora funcione pode haver comportamento inesperado por parte do programa, para além disto, nessa combinação não será possível introduzir input através do stdin, sendo só possível pôr a cobra também em modo automático.

```
new GameManagerBuilder()
    .addObstacle(poly, rotationPoint, speed)
    .setSeed(seed)
    .setTextual(false)
    .setFilled(false)
    .setMapWidth(500)
    .setMapHeight(500)
    .setSnakeSize(50)
    .setFoodSize(50)
    .setFoodScore(5)
    .setInputPreset(InputPreset.ARROW_KEYS)
    .setFoodType(GameManager.FoodType.CIRCLE)
    .setUpdateMethod(GameEngineFlags.UpdateMethod.AUTO)
    .setMaxFps(1)
    .setControlMethod(GameManager.ControlMethod.MANUAL)

    // setting colors
    .setGraphicalBackgroundColour(Color.black)
    .setGraphicalSnakeColour(Color.green)
    .setGraphicalFoodColour(Color.red)
    .setGraphicalObstaclesColour(Color.magenta)

    .build();
```



Entregável 1 e 2

Geometry

Este package contém todas as classes relacionadas com geometria e por consequência com matemática em geral.

Todas as classes neste package estão escritas para geometria euclidiana em duas dimensões, para além de serem todas imutáveis depois de instanciadas.

GeometricException: Representa uma exceção numa operação geométrica.

MathUtil: Classe estática para operações matemáticas comuns no projeto, com enfâse na comparação entre floating points, de modo a estabelecer um “standard” para o projeto. Representa a definição do que são dois floating points iguais para o projeto.

Line: Representa uma reta, contendo várias operações para manipular a mesma, tais como testar colinearidade, interseção, paralelismo, testar perpendicularidade e gerar reta perpendicular.

LineSegment: Representa um segmento de reta, tendo métodos principalmente para verificar interseções e colinearidade com pontos/verificar se um ponto está contido no segmento.

VirtualPoint: Representa um ponto cujas coordenadas são números reais. Contém métodos que ajudam a manusear pontos, tais como rotação, translação, distância...

Point: Representa um ponto cujas coordenadas estão no primeiro quadrante (são sempre positivas). (Subclasse de VirtualPoint)

Vector: Representa um vetor, contém funcionalidade para operações comuns com vetores, tais como produto escalar, multiplicação (por um quociente)...

Path: Representa um caminho.

BoundingBox: Representa os limites de um conjunto de pontos. É principalmente útil para saber o ponto mínimo ou máximo de um conjunto de pontos/polígono.

IGeometricShape: Representa uma figura geométrica, tendo esta uma série de operações geométricas possíveis, tais como rotação, interseção (tal como uma versão "inclusiva" da interseção, que conta como interseção quando as linhas das figuras estão sobrepostas), translação, e verificação de contenção.

Circle: Representa um círculo, este sendo uma IGeometricShape.

Polygon: Representa um polígono simples, este sendo uma IGeometricShape.

Rectangle: Representa um polígono de 4 lados com ângulos internos retos (um retângulo).

Square: Representa um retângulo cujos lados são todos do mesmo comprimento (um quadrado).

Triangle: Representa um polígono de 3 lados (um triângulo).

Game Engine

Este package contém todas as classes responsáveis pela funcionalidades necessárias para fazer um jogo (qualquer jogo, sendo este limitado a duas dimensões).

GameEngineException: Representa uma exceção relacionada Game Engine.

GameEngine: Classe singleton responsável por executar o jogo contido numa Scene, chamando os métodos de update e certificando-se de que as colisões são chamadas e a Scene é renderizada. Esta classe, a cada ciclo é responsável por acionar e coordenar os seguintes eventos (por ordem):

- **Start** – chamado uma única vez, no início. Usado para inicialização.
  - ➔ **Input –** input é lido do stdin (no caso de estar em modo de input).
  - ➔ **Early Update** – update chamado no inicio de cada ciclo. Usado principalmente para inicializações que têm de ser feitas antes do update principal.
  - ➔ **Update** – evento responsável por atualizar o estado atual dos GameObjects.
  - ➔ **Collisions** – as colisões são verificadas e acionadas.

- ➔ **Late Update** – update chamado no final de cada ciclo. Usado principalmente para "cleanup".
- ➔ **Rendering** – a Scene atual é renderizada.

- **Stop** – chamado uma única vez, quando o GameEngine para. Usado para "cleanup".

GameEngineFlags: Classe responsável por definir o comportamento do GameEngine, para além de conter valores *default.* Mais especificamente, define como o GameEngine vai dar update (através de código, automaticamente, através de input…) e define como o GameEngine vai escolher renderizar a Scene (textualmente no terminal ou através de uma interface gráfica).

Scene: Representa uma coleção de GameObjects. Esta pode estar ativa ou não dependendo de se está a ser usada pelo GameEngine ou não. A Scene é também responsável por categorizar os GameObjects pelas suas interfaces relevantes ao GameEngine.

GameObject: Representa um objeto no jogo a ser instanciado pela Scene. Contém um id o qual é definido pela Scene que o contém (não sendo possível este ser instanciado em mais que uma Scene). Contém também uma handle para a Scene que o instanciou, para que este tenha o poder de comunicar diretamente com outros GameObjects da Scene, ou mesmo para adicionar/remover GameObjects à Scene (ou até mudar a Scene). Esta classe tem os 5 métodos correspondentes aos eventos do GameEngine: start, stop, update, early update e late update. Esta classe está pensada para ser *inherited*.

CollisionManager: Classe estática responsável por detetar colisões entre IColliders na Scene e invocar o evento de colisão para as mesmas. Esta também detecta "deep collisions", quando algum dos respectivos IColliders assim estiver definido. Uma deep collision para além de testar interseção, também testa se um dos IColliders está contido dentro do outro.

ICollider: Interface responsável por representar um GameObject que poderá colidir. O GameObject deverá apresentar uma figura geométrica para testar colisão (popularmente conhecida como uma *hitbox*). Tem também o método "onCollision", chamado no evento de uma colisão com outro ICollider, passando como argumento o GameObject com que colidiu.

IInputListener: Interface responsável por representar uma classe que recebe input do utilizador quando o utilizador introduz algum input.

Renderer: Classe singleton responsável por renderizar a Scene (especificamente IRenderables usando RenderData). Usa o algoritmo de rasterização de linhas de Bresenham para rasterizar linhas, o algoritmo de rasterização de círculos de Bresenham para rasterizar círculos (e uma variante para os preencher) e usa o algoritmo Scanline para rasterizar polígonos (possivelmente côncavos) preenchidos. Renderiza também IOverlays, sendo estes renderizados de forma diferente: são "colados" por cima depois da Scene ser renderizada. Ao renderizar uma Scene, tem também atenção ao layer do objeto a renderizar (o maior layer fica por cima).

IRenderable: Interface responsável por representar uma classe a ser renderizada pelo Renderer, expondo o seu RenderData ao Renderer.

IOverlay: Interface responsável por representar um GameObject que será renderizado como um overlay, sendo este "colado" por cima da Scene renderizada.

RenderData: dados a serem expostos ao Renderer sobre como renderizar o objeto, nomeadamente a forma geométrica, se deve ser preenchido, o layer, o caractere representante e a cor (opcional). Esta classe implementa Comparable, pois deve poder ser ordenada por layer.

Colour: Enumerável que representa uma cor como uma string de um código ANSI.

TextOverlay: Classe com a utilidade necessária para criar um overlay de texto. Incluindo utilidade para gerar uma outline à volta do ecrã, preencher o ecrã, para escrever texto centrado, alinhado à esquerda e à direita numa dada linha, ou até para escrever um parágrafo (podendo possivelmente estender-se por várias linhas). Pode usar uma TextOverlayOutline para gerar a outline.

TextOverlayOutline: Classe a informação necessária para gerar uma outline, contendo também defaults para uma outline.

Logger: Classe responsável por fazer logging para o terminal quando ocorrem erros ou outro tipo de informação que poderá ser útil quando o programa não termina mas não funciona como esperado (ou até mesmo quando termina).

Snake Game

Este package contém todas as classes responsáveis pelas funcionalidades específicas ao jogo da cobra.

SnakeGameException: Representa uma exceção relacionada ao jogo da cobra.

Snake: Representa uma cobra, sendo esta capaz de se mexer, virar, comer, crescer e morrer. Esta só começará a andar quando for acordada (método awake() chamado). A cobra só crescerá no update asseguir ao em que comeu. Por questões de eficiência, a maneira como a cobra anda é usando uma queue, onde todas as SnakeUnits estão contidas, para andar simplesmente se remove a próxima SnakeUnit da queue e esta é colocada na direção em que a cobra é suposto mover-se, definindo essa SnakeUnit como a cabeça da snake e pondo a cabeça antiga da snake na queue.

SnakeUnit: Representa a unidade mais pequena de uma cobra (cada segmento). Esta extende a classe Unit, pode colidir (é um ICollider) e tem a propriedade de ser ou não a cabeça de uma snake (sendo a principal funcionalidade desta propriedade renderizar a cabeça de forma diferente). Se a SnakeUnit cabeça tiver uma “deep collision” com a comida em que a comida está estritamente dentro da cabeça, ela comerá a comida. Se qualquer SnakeUnit colidir com outra SnakeUnit, a cobra inerente morre.

Unit: Classe que representa uma unidade (quadrada) no espaço. Esta classe tem uma posição (implementa ISpatialComponent) e é renderizada. Uma unidade é especial pois tem uma propriedade tamanho. Os seus lados serão renderizados com o mesmo número de caracteres que o seu tamanho: se uma unidade tem o tamanho 3, esperamos que quando renderizada, vejamos 3 caracteres de comprimento e altura.

ISnakeStats: Interface para obter propriedades da cobra sem expor a cobra ou a classe responsável por saber o score (o GameManager no caso), tais como para ver qual o score atual e para ver a posição e direção atual da cobra.

SnakeStats: Classe responsável por obter as propriedades da cobra (implementa ISnakeStats). Vai buscar o score à instância atual do GameManager.

ISpatialComponent: Interface indicadora de algo que tem uma posição no espaço. (como por exemplo a cobra e a comida)

Direction: Enumerável que representa uma das 4 possíveis direções no jogo da cobra. Tem também um Enumerável que representa as possíveis direções em que se pode virar (pode-se virar para a esquerda, direita, ou não virar). Para além disto tem funcionalidade de converter uma mudança de direção absoluta numa direção relativa (por exemplo, se a direção absoluta é cima e a outra direção absoluta é esquerda, então a direção de viragem relativa será esquerda).

SnakeController: Classe responsável por controlar a cobra, seja o controle feito através de input do utilizador ou através de um AI. A classe simplesmente pede a cada ciclo a um ISnakeController que dê a próxima direção para onde virar a cobra.

ISnakeController: Interface que representa uma classe que diga para onde a snake deverá virar asseguir. (principalmente o InputSnakeController e o AISnakeController).

InputSnakeController: Classe responsável por controlar a cobra através de input do utilizador (implementa IInputListener e ISnakeController). Tem também uma série de “presets” para interpretar como input (nomeadamente WASD, VIM keybinds, direção absoluta e direção relativa).

AISnakeController: Responsável por calcular a próxima jogada da snake autonomamente. Usa o algoritmo de Dijkstra, o qual numa grelha discreta bidimensional é literalmente só uma BFS. Quando não é encontrado um caminho para a comida, é escolhido o caminho para a unidade mais longe da cobra. NOTA: Seria possível resolver facilmente o jogo da cobra calculando e seguindo um caminho hamiltoniano por todas as posições válidas do mapa, mas esta solução sem nenhuma modificação ao ciclo hamiltoniano é extremamente aborrecida de observar, sendo que a cobra passa maioria do tempo sem fazer nada de interessante a percorrer o ciclo, então foi tomada uma decisão de projeto de usar o Dijkstra, pois dá resultados mais divertidos de observar. (para além de também ser melhor no caso de existirem obstáculos dinâmicos).

GameMap: Representa o mapa de jogo: as suas dimensões, background e obstáculos à sua volta, parando a snake de sair do mapa. Esta classe não é um singleton para que, se desejado, futuras versões possam implementar uma versão (por exemplo “splitscreen vs”) onde poderá haver mais que um mapa no ecrã. O GameMap introduz uma série de funções extremamente úteis para lógica sobre Units, as quais são por um lado representações de posições discretas no espaço real. Um de inúmeros exemplos destas funções é um método para obter uma posição no mapa válida para instanciar uma Unit. O GameMap internamente (e dá as ferramentas para fazer o mesmo externamente) usa coordenadas relativas sobre o mapa, contando como origem o inicio do mapa. Isto é extremamente útil para inúmeros cálculos, tal como simplificação (através de abstração) do posicionamento de objetos com representação discreta em coordenadas reais.

IFoodStats: Interface para obter propriedades da comida sem expor a comida. A interface só expõe a posição da comida, mas está pensada de modo a que no futuro se quisermos expor mais informação sobre a comida, também seja possível com esforço mínimo.

FoodStats: Classe responsável por obter a informação sobre a comida instanciada atual.

IFood: Interface representante de um objeto de comida, podendo esta ser consumida. O objeto de comida é renderizado e pode colidir com outros objetos.

FoodSquare: Representa uma comida que é um quadrado (e mais especificamente uma Unit). Ao ser consumida é removida da Scene à qual pertence.

FoodCircle: Representa uma comida que é um círculo. Ao ser consumida é removida da Scene à qual pertence.

IObstacle: Representa um obstáculo o qual a cobra poderá colidir com.

StaticObstacle: Representa um obstáculo que não pode ser movido, por outras palavras, estático.

DynamicObstacle: Representa um obstáculo que está em constante movimento periódico, rodando à volta de um ponto fixo.

GameplayOverlay: (Classe representante do) Overlay usado quando o jogo está a decorrer e a cobra ainda está viva e em movimento. (mostra a direção da cobra e o score atual)

GameoverOverlay: (Classe representante do) Overlay usado quando o jogo atual termina, mostrando o score obtido.

HighscoresOverlay: (Classe representante do) Overlay usado para mostrar os highscores. Este overlay está encarregue de formatar os scores de modo a que estes sejam mostrados de forma uniforme e legível. A classe pode ser chamada de modo a mostrar todos os scores (que caibam no overlay) ou mostrar o top *n* scores. A classe obtém os scores de um IHighscoresReader.

IHighscoresReader: Interface que representa um objeto que lê e fornece os highscores gravados.

Score: Representa a pontuação de um jogo, correspondendo a um tuplo na classe Scoreboard. Pode ser ordenado por score e pode também ser serializado.

Scoreboard: Classe singleton que representa todos os highscores que foram obtidos a jogar o jogo. Tem funcionalidade para salvar scores para um ficheiro e também para os ler.

GameManagerBuilder: Classe responsável por inicializar um GameManager, por consequência inicializando também o jogo da cobra. Contém os métodos necessários para facilitar a construção do GameManager, abstraindo maioria dos conceitos do jogo para este poder ser inicializado com um conhecimento mínimo de como este está feito.

GameManager: Classe singleton responsável por coordenar o jogo da cobra, tratando da Scene no caso do jogador perder ou recomeçar o jogo (por exemplo reinstanciando a cobra com tamanho inicial após perder). Esta classe permite inicializar o jogo da cobra abstraindo-nos de grande parte das classes.

Entregável 3

NOTA: As classes com um asterisco representam classes que foram alteradas.

Geometry

Não foram feitas alterações a este package.

Game Engine

Clock: Classe responsável por chamar o evento tick() a todos os IClockListeners. Esta classe é instanciada com um determinado número de ticks máximos por segundo e quando o seu método start() é chamado ela invoca o evento tick. Ao acabar o evento, põe a thread atual a dormir pelo tempo restante, caso algum.

IClockListener: Interface que representa uma classe que ouve o evento tick da classe Clock.

GraphicWindow: Classe que representa uma janela gráfica de um jogo. A janela apresenta a mesma disposição lógica que é usada para o terminal, ela tem um Raster, onde o jogo é desenhado pelo Renderer, e tem um overlay, algo que é "colado" por cima do jogo renderizado. O overlay, sendo gráfico, em vez de um array de caracteres é um JPanel. Esta classe extende JFrame.

A classe contém também alguns métodos úteis para a sua utilização, como close() que fecha a janela e um método que desenha numa determinada coordenada/pixel no Raster.

Raster: Representa um raster: um array bidimensional de pixeis, tendo funcionalidade necessária para desenhar num determinado pixel e também para se desenhar. Extende JPanel.

GraphicOverlay: Classe análoga à classe TextOverlay, mas para janelas gráficas: Contém operações comuns a overlays gráficos. Atualmente tem somente um construtor que facilita a criação de overlays, sem ter de repetir as mesmas contas para todos os overlays gráficos.

InputManager: Classe singleton responsável por captar input do utilizador e ativar os eventos correspondentes do IInputListener. Funciona também como uma wrapper class para KeyListener.

***IInputListener:** Foram adicionados mais 3 eventos, correspondentes aos eventos do Java AWT keyPressed, keyReleased e keyTyped, sendo estes, respetivamente onKeyPressed, onKeyReleased e onKeyTyped. Com esta mudança, por outras palavras esta classe serve agora também de wrapper interface de KeyEvent (o uso da wrapper interface foi motivado pelo facto de que IInputListener já era usado para o input do terminal, e todas as classes que devem ser notificadas de input do terminal, devem também ser notificadas de input do teclado quando em modo gráfico. Outro argumento é de que se, no futuro, cessar o uso do Java AWT, todos os IInputListeners continuam a ter estes 3 eventos, eventualmente acionados por outro mecanismo abstraído a esta interface.)

***IOverlay:** Tem agora uma função para obter o overlay gráfico.

NOTA: embora esta classe tenha uma função para obter o overlay gráfico tal como o overlay textual, a implementação de ambos não é necessária, sendo apenas necessário retornar null caso não seja da intenção do overlay suportar o modo textual/gráfico. Tal como tudo o escrito neste relatório (e muito mais), isto está também descrito na documentação do código. Caso o GameEngine

instancie um overlay no modo textual, sendo que este não está implementado para esse modo, é gerada uma exceção com uma mensagem correspondente. Foi escolhida a abordagem de usar valores nulos em vez de gerar duas interfaces (e.g. ITextualOverlay e IGraphicalOverlay) pois o GameEngine onde os overlays são instanciados suporta modo textual **e** gráfico, sendo um overlay que não implemente um dos dois… uma *exceção*.

***GameEngineFlags:** Foi adicionado um parâmetro novo, representativo do número de updates máximos por segundo no caso do GameEngine estar em modo automático. Embora este parâmetro não seja uma flag como os outros, o nome GameEngineFlags manteve-se, pois acredito que continua a representar o intuito da classe.

***GameEngine:** Agora implementa a interface IClockListener, subscrevendo ao evento do clock, o qual ele usa no modo de update AUTO.

Coordena também o InputManager de modo a receber o input do utilizador (previamente o GameEngine só necessitava de receber input no modo STEP, onde simplesmente lia o input do stdin e atualizava após ler. Com a adição da componente gráfica, foi necessário adicionar um sistema ligeiramente mais sofisticado, dando origem ao InputManager).

Contém um método para aceder ao ficheiro "root" do projeto de onde está a ser executado, independentemente do sistema operativo.

***Colour → TerminalColour**: Foi apenas mudado o nome, pois esta representa apenas cores no terminal (através de códigos ANSI), sendo este sistema completamente distinto de cores numa interface gráfica (que neste projeto usam a classe Color, do Java AWT, a qual usa valores rgb).

***RenderData:** Capaz agora também de guardar uma Color, uma cor para o modo gráfico.

***RenderData:** Caso executado no modo gráfico renderiza agora tudo analogamente, mas em vez de desenhar num array o qual é depois imprimido para o stdout, desenha num array de pixeis o qual depois é desenhado no ecrã (nos algoritmos nada muda, no método draw() simplesmente desenha num sítio diferente no caso de ser gráfico). Responsável também por instanciar a janela no modo gráfico e de adicionar o componente overlay à mesma caso haja um na Scene que está a ser renderizada.

Snake Game

***GameManager:** Foi feita uma série de ajustes para acomodar e instanciar o jogo da cobra no GameEngine quando em modo gráfico.

***GameManagerBuilder:** Adicionou-se opções para construção de um GameManager com cores gráficas diferentes.

***GameoverOverlay:** Adicionou-se a componente gráfica do overlay, incluindo um campo de introdução de texto. Os tamanhos são completamente relativos ao tamanho da janela.

***GameplayOverlay:** Adicionou-se a componente gráfica do overlay, mostrando somente o score. Os tamanhos são relativos ao tamanho da janela e ao tamanho de uma unidade de cobra (para que o texto não tape uma unidade inteira).

***HighscoresOverlay:** Adicionou-se a componente gráfica do overlay, mostrando os scores. Caso haja mais scores do que é possível mostrar na janela, é também possível dar scroll pelos scores.

***InputSnakeController:** Contém agora um preset de input para as setas no teclado.

Notas de Projeto

(entregável 1 e 2)

Design Patterns utilizados:

- **Observer:** (em GameEngine, Scene e GameObject), sendo este útil para notificar os objetos do jogo. Sendo que desta forma ambos também estão completamente abstraídos um do outro.
- **Builder:** (em GameManager, GameManagerBuilder), extremamente útil para gerar o GameManager, já que este necessita de um alto número de argumentos para configurar o jogo como desejado.
- **Iterator:** (em Scene), para iterar sobre coleções de GameObjects, já que esta é uma operação comum.
- **Singleton:** (em GameManager, GameEngine, Renderer, Scoreboard), pois para estas classes não faz sentido ter mais que uma instância (podendo até ser problemático).

Decisões notáveis:

Devido a uma conversa com o prof. JVO, na altura, foi definido que só a cabeça da cobra poderá comer uma comida, logo o projeto foi feito com essa regra em mente, mesmo que não esteja descrita no enunciado.

Contudo esta regra tem implicações notáveis: Como só a cabeça da cobra pode comer, e como a comida só poderá aparecer em sítios onde a cobra a possa comer, então a comida não pode aparecer “entre duas células” (podemos assumir que o mapa está sub-dividido em células/units, do tamanho da cabeça da cobra, estas unidades representam todas as posições que uma unidade da cobra pode tomar). Por outras palavras, a comida pode nascer no mesmo sítio ou dentro de qualquer sítio válido que a cabeça da cobra possa estar. Por exemplo para uma cobra 3x3 e uma comida 1x1, haverá 9 posições válidas dentro de cada unidade que não esteja já ocupada para a comida aparecer. Se o mapa for 9x9 e não houver obstáculos nem cobra, existiriam (3*3*9=81) posições possíveis para a comida aparecer, que neste caso são todas as posições possíveis, o que é esperado sempre que a comida tem dimensões 1x1.

Outra decisão notável é que um obstáculo pode colidir livremente com uma comida, desde que a comida não nasça dentro de um obstáculo (sendo também impossível configurar o jogo para que comece com uma comida dentro de um obstáculo). Esta decisão foi tomada com a seguinte lógica em mente: Com estas regras não é possível uma comida estar dentro/a colidir com um obstáculo, mas é possível ter sido posta numa posição válida em que posteriormente foi preenchida por um obstáculo dinâmico. Sendo que os obstáculos dinâmicos têm movimento periódico, isto quer dizer que, eventualmente, **haverá** outro momento em que a comida poderá ser comida pela cobra. Para além disto a filosofia tomada é de que é da responsabilidade de quem construiu o mapa/disposição atual de por os obstáculos de maneira que não crie uma situação indesejável para o jogador (a não ser que assim seja o desejo de quem fez o mapa/”nível”).

Por fim uma nota sobre o UML gerado. O UML foi separado em packages. Esta decisão foi tomada pois, por exemplo, a classe ponto é extensivamente usada na package SnakeGame, o mesmo pode ser dito por outras classes (como por exemplo o GameObject que é extensivamente extendido no package SnakeGame). Com este tipo de relações, qualquer disposição do UML seria **completamente** ilegível por um humano, mesmo quando todas as inúmeras relações sejam postas de forma a que não pareça um mapa das trajetórias de todos os voos na Europa, ainda é impossível

de ler o UML, logo os UMLs gerados foram configurados de modo a só mostrar relações entre o próprio package (embora os atributos de objetos de outros packages nas classes continuem a aparecer).

(entregável 3)

(importante)

Durante o desenvolvimento do entregável 3, para motivos de teste de compatibilidade o projeto foi executado com outros compiladores populares no Windows, o qual teve vários problemas e exceptions que não tinham sido acionadas no ambiente de desenvolvimento do projeto (que no caso foi em Unix, com o compilador javac). Caso não tenha sido possível correr o entregável 2 no ambiente de desenvolvimento onde o código foi testado, o entregável 3 tem a parte textual idêntica e, em princípio, deve correr em qualquer ambiente. Caso o prof. não tenha conseguido correr o entregável 2, e não aceite a parte textual do entregável 3 como substituição, eis o comando usado para compilar o entregável 2 (e para o mesmo efeito o entregável 3):

```
javac -Werror -d $out_dir -cp $junit_dir $files
```

onde

$out_dir é o diretório onde o output deve ser gerado.

$junit_dir é o diretório onde reside a library do JUnit (standalone) (em princípio esta opção deverá ser opcional)

$files todos os ficheiros do projeto, separados por um espaço.

Deixo também anexado o script exato usado para compilar o programa, caso seja necessário (com o nome de java_compile).

Sob esta nota acrescento a informação que certos testes unitários, nomeadamente os testes que têm output gráfico do stdout, dão falsos negativos no Windows (e possivelmente noutros sistemas operativos), pois os valores esperados incluem “\n” o que corresponde no Windows a “\r\n”.

Não foi usada qualquer função do Java AWT/Swing para renderizar a cena. No projeto o AWT/Swing é responsável exatamente por: Instanciar a janela em modo gráfico, desenhar um array de pixeis na janela, desenhar a totalidade dos overlays em modo gráfico e por acionar os eventos de input de teclas em modo gráfico. Todo o jogo (exceto o overlay) é renderizado utilizando a classe Renderer, que é responsável por desenhar o jogo pixel a pixel.

```
436 tests successful
```

Foram feitos um total de 436 testes com cobertura quase perfeita, à exceção de UI testing, o qual não foi feito de todo por duas razões: UI testing não é abrangido por unit testing (o pedido no enunciado) e por falta de tempo, sendo a segunda razão o maior fator.

No UML atualizado, foi adicionado as parent classes como prefixo ao nome das classes no caso da parent class estar num package diferente. De modo a que a sua relação esteja explícita no UML.

Existe uma colisão com os ideais do projeto e os ideais do AWT, causando duas linhas de código extremamente desagradáveis. Foi escolhido AWT ao invés de OpenGL, mesmo apesar de estar tudo a ser renderizado pixel a pixel, devido a certas funcionalidades do AWT que não teriam sido possíveis implementar em OpenGL no tempo limite do entregável 3. Especificamente, a caixa de input do nome do jogador e a funcionalidade de scroll nos highscores.

Justificada a razão pela qual foi escolhido AWT, existem problemas quanto a esta decisão, não previstos inicialmente: O AWT não foi pensado para desenhar uma janela pixel a pixel. Tal foi possível alterando os valores dos pixeis de uma BufferedImage, à qual após a rasterização estar completa, é renderizada no ecrã. Isto é um problema por várias razões: Primeiramente o AWT não deteta que houve alterações à BufferedImage e por vezes não desenha automaticamente as diferenças. Para arranjar este problema basta após finalizar a rasterização, chamar o método redraw(), mas isto gera outro problema: O método redraw() pode não desenhar imediatamente a frame, adicionando apenas um evento à sua EventQueue simbolizando que terá futuramente de redesenhar a frame.

A biblioteca AWT/Swing (propositadamente) não tem nenhuma maneira de fazer a EventQueue disparar. Mesmo métodos como invalidate() geram apenas eventos para no futuro quando a EventQueue disparar, redesenhar. A razão porque isto é um problema é porque o tempo que demora a EventQueue a disparar consegue por vezes ser sentido pelo utilizador (depende muito das circunstâncias do computador em que está a ser executado). Benchmarks sob o projeto mostraram que no ambiente específico onde o programa estava a ser corrido, o GameEngine passava 98% do tempo idle, ou seja, o pequeno atraso sentido não é devido a computação excessiva (CPU bound). Outro parâmetro revelado pelas benchmarks realizadas sob o projeto, foi que de fato existia um pequeno delay de 50-100ms.

Após extensiva pesquisa na documentação do AWT e em fóruns online, para além de testes locais, foi descoberto que após um elevado número de eventos na EventQueue do AWT, este é forçado a dar flush na EventQueue, executando todos os eventos, incluindo o de redraw. Para este efeito existem duas linhas de código nada elegantes, como mencionado anteriormente, em GameEngine.GraphicWindow::paint(), as quais enchem a EventQueue suficientemente para esta dar flush instantaneamente. Após essas duas linhas, a benchmark feita deu um delay de 0ms (os delays eram desde o fim de renderização de tudo até ao inicio da BufferedImage ser desenhada na janela).

As duas linhas podem ser removidas e o projeto funcionará de igual modo, com a única diferença sendo que há a possibilidade de haver um pequeno delay entre o fim da rasterização e da lógica, e a imagem rasterizada ser mostrada no ecrã (pelo feedback de outros usuários do programa, o delay não se nota, embora ele exista).

Nos mesmos benchmarks realizados (mais uma vez lembrando que tais benchmarks são específicos ao ambiente onde o programa foi executado) os eventos extra não causaram *quaisqueres* perdas de performance, e embora não tenha sido testado, é provável que para qualquer computador com um processador dual core ou superior não haja quaisqueres perdas de performance com estas duas linhas, apesar da sua deselegância. Assumindo que o processador já não está sobre loads grandes.

Para além destas duas linhas, a incompatibilidade do AWT com rasterização pixel a pixel em conjunto de definir um valor elevado de fps máximos causa um problema de sincronização, o que por sua vez causa glitches gráficos. Embora o computador facilmente compute tudo necessário e, mais uma vez, passe maioria do tempo idle, o AWT não suporta tamanha taxa de atualização e causa inúmeros glitches visuais. O problema pode ser facilmente replicado pondo um número elevado de fps (e.g. 60), a cobra em modo automático e os updates do game engine também em modo automático. Infelizmente não é possível arranjar isto, sendo esta uma limitação do AWT, o qual não está a ser usado como pretendido, sendo rasterizado pixel a pixel, e sendo forçados os seus updates a uma taxa não suportada.