jvo, hdaniel, jbarateiro

Lab 5: Game Engine, parte 1

Este trabalho vai ser o primeiro de uma série para efetuar em grupo e que levará ao projeto final de grupo. Primeiramente é necessário inscrever o grupo, de acordo com:

1. Use o link disponível na tutoria eletrónica ou:

   http://deei-mooshak.ualg.pt/~jvo/POO/Entregas

2. Depois, aceda ao mooshak através do link da tutoria ou: http://deei-mooshak.ualg.pt/~jvo/
3. Para obter User/Password clique em Register [for On-line Contest]

Em **Contest**, selecione POO 2024/25

Em **Group**, selecione a sua turma

4. Em **Name**, escreva POO2425p<i>g<j> onde <i> é a sua turma prática (1, 2, 3, 4 ou 5) e <j> é o seu número de grupo, usando os dados fornecidos em:

   http://deei-mooshak.ualg.pt/~jvo/POO/Entregas

5. Exemplo: **POO2425p1g1**
6. Em **Email**, escreva o endereço de correio eletrónico de um dos elementos do grupo para onde será enviada a password.

Submeta o seu código ao Mooshak http://deei-mooshak.ualg.pt/~jvo/ usando o login do seu grupo, ex: **POO2425p1g1**

Uma submissão ao problema permanecerá *pending* até que seja validada pelo professor durante a aula prática, **com todos os elementos do grupo**.
Só as submissões em estado *final* serão consideradas para avaliação.

Todas as submissões deverão ser feitas até: **30 de março 2025**

A validação poderá ser feita posteriormente, se necessário, até: **11 de abril de 2025**

NB: Nos problemas seguintes, sempre que necessário, considera-se que dois double *d* e *g* são iguais se $|d-g|<1e-9$

Restrições
Em todos os problemas devem seguir os princípios da programação orientada por objetos, estruturando o código segundo um conjunto de classes apropriadas.

Para validar cada problema é necessário que sejam implementados os interfaces indicados.

Para cada um dos problemas seguintes, antes de escrever qualquer código, **defina um conjunto de testes unitários** que reflitam o comportamento pretendido.

Comece por **esboçar um diagrama de classes inicial (de análise) UML** com as classes necessárias para implementar os problemas descritos. Este diagrama deve ser mostrado durante a validação, não devendo ser submetido ao Mooshak.

Adicionalmente, deve comentar o código, indicando:

i) Uma linha com a descrição da responsabilidade da classe ou do que o método faz

ii) @author (apenas para classes)
iii) @version (apenas para classes; inclua uma data)
iv) @inv (apenas para classes; inclua uma descrição da invariante usada)

v) @param (apenas para métodos e construtores)
vi) @return (apenas para métodos)

vii) @see (qualquer referência bibliográfica ou sítio web consultado para o desenvolvimento do código respetivo)

Desenvolvimento de um Game Engine 2D

Neste laboratório vamos começar o desenvolvimento de um *Game Engine 2D*, que continuaremos nos próximos. Vai ser reutilizado e estendido o que já foi desenvolvido sobre figuras geométricas nos laboratórios anteriores. Mas primeiro vamos dar um panorama geral do que se pretende.

O *game engine*, de futuro referenciado apenas por *engine*, suportará jogos em 2D, como por exemplo o que se encontra na figura.

Cada um dos elementos visíveis no ecrã é uma instância da classe **GameObject**:

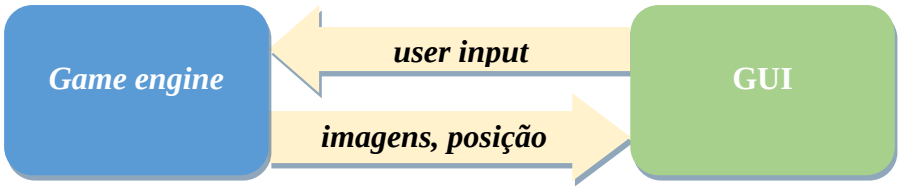
- Aeronaves, veículos
- projeteis
- explosões
- a imagem de fundo
- e mesmo a pontuação ou *score*

Assim, para este *engine* vamos ter 2 tipos de objetos:

- Imagens 2D
- Texto

que iremos descrever mais tarde quando abordarmos o *Graphics user interface* ou GUI. Para já vamos implementar o núcleo ou *core* do *game engine*, que não inclui o GUI.

Por agora apenas necessitamos saber que o GUI pode ser visto como um serviço externo, que fornecerá quando lhe for pedido pelo *engine* o input do utilizador e apresentará quando lhe for enviado, também pelo *engine*, as imagens dos **GameObjects** nas posições indicadas. Isto é o *engine* controla o GUI e não o contrário:



O *engine* mantém uma lista de **GameObjects** a apresentar no ecrã. Após inicializado o *engine* corre um *loop* com os seguintes passos para criar cada *frame*:

Loop forever:
Pede input do utilizador ao GUI
Atualiza as posições dos objetos
Envia a lista de objetos e posições para o GUI para criar uma *frame*

De notar que toda a informação para criar uma *frame* existe e é manipulada no *engine*. O GUI limita-se a colocar as imagens dos GameObjects na posição indicada, com a rotação e a escala também indicadas pelo *engine*.

Assim um **GameObject** é uma classe com vários atributos, necessários não só para o apresentar no ecrã, mas também para o funcionamento do jogo:

GameObject
- Transform
- Shape
- Collider
- Behaviour

0 º

Transform contém toda a informação sobre posição, *layer*, rotação e escala. Embora seja um *engine* 2D a existência de *layers* permite modelar **GameObjects** em vários níveis. Podemos assim evitar detetar colisões entre aeronaves e veículos terrestres. A rotação é indicada em graus no sentido anti-horário, onde zero graus representa o Norte. É efetuada em relação ao centroide do **GameObject**, isto é a posição no plano indicado na Transform.

Shape contém a imagem a apresentar no GUI.

Collider é usado para detetar colisões entre objetos. A deteção de colisões é das tarefas mais pesadas num *game engine*. Para tornar esta tarefa mais leve, imagens 2D complexas são envolvidas em figuras geométricas mais simples, que são usadas para detetar a colisão.

Na figura podemos ver os colisores a vermelho, centrados nas imagens dos **GameObjects**. Neste *engine* vamos suportar 2 tipos de colisores: Círculos e Polígonos.

Behaviour é uma classe que implementa os métodos de controlo de cada **GameObject**, sejam os mais elementares: mover, rodar, disparar ou mais complexos: fugir, patrulhar, atacar. A criação de comportamentos complexos pode ser efetuada decompondo as tarefas em outras classes de acordo com os princípios da programação orientada por objetos.

Nos problemas deste laboratório vamos focar-nos na criação e movimentação dos **GameObjects** e consequentemente dos seus colisores, utilizando e estendendo o que já desenvolvemos nos laboratórios anteriores, enquadrando já no contexto do *game engine*. A deteção de colisões ficará para o próximo laboratório.

Seguidamente apresenta-se a informação básica necessária para representar um **GameObject**. Esta está guardada na classe **GameObject**, que implementa a interface **IGameObject** e representa toda a informação necessária para cada objeto colocado no jogo. Para já vamos utilizar uma versão inicial, que será estendida posteriormente, que inclui apenas um nome, uma **ITransform** e uma **ICollider**.

```
public interface IGameObject {

   /**
    * @return the name of the GameObject
    */
   String name();

   /**
    * @return the Transform of the GameObject
    */
   ITransform transform();

   /**
    * @return the Collider of the GameObject with its centroid at this.transform().position()
    */
   ICollider collider();

   // ...
}
```

Todas as transformações dos **GameObjects** suportadas:

Translação
Rotação
Escala

são implementadas pela classe **Transform** que implementa os métodos descritos na interface **ITransform.** Esta classe guarda a posição do **GameObject** no plano cartesiano, coordenadas x, y reais tipo double. Guarda também em que layer se encontra, numa variável inteira com sinal. Guarda ainda a orientação em graus e o fator de escala, ambos tipo double.

```
public interface ITransform  {

   /**
    * Move this ITransform by dPos.x(), dPos.y() and dlayer
    * @param dPos     the 2D differential to move
    * @param dlayer  the layer differential
    */
   public void move(Point dPos, int dlayer);

   /**
    * Rotate this ITransform from current orientation by dTheta
    * @param dTheta
    * pos: 0 <= this.angle() < 360
    */
   public void rotate(double dTheta);

   /**
    * increment the ITransform scale by dscale
    * @param dScale the scale increment
    */
   public void scale(double dScale);

   /**
    * @return the (x,y) coordinates
    */
   public Point position();

   /**
    * @return the layer
```

```
    */
    public int layer();

    /**
     * @return the angle in degrees
     */
    public double angle();

    /**
     * @return the current scale factor
     */
    public double scale();
}
```

Os colisores são implementados pela classe **Collider** que implementa a interface **ICollider**. São suportados dois tipos de colisores:

Circulos
Poligonos

O colisor tem o centroide na posição do **GameObject** dada pela **transform.position()** do **GameObject** a que está associado.

```
public interface ICollider {

    Point centroid();

    // ...
}
```

Mover um GameObject
Para mover um **GameObject** basta somar o diferencial **dPos** à posição corrente na **Transform**.
Para mover o colisor, se for um círculo basta somar **dPos** ao centro. Se for um polígono soma-se **dPos** a todos os vértices.

Escalar um GameObject
Para escalar um **GameObject** a posição original mantém-se e o fator de escala na **Transform** é atualizado. O GUI irá escalar a imagem de acordo com esta informação.
Para escalar o colisor, se for um círculo basta multiplicar o raio pelo fator de escala. Se for um polígono, pode deslocar-se o polígono de forma que o seu centroide seja (0,0), multiplicar as coordenadas de todos os vértices pelo fator de escala e voltar a deslocar para a posição original.

Rodar um GameObject
Para rodar um **GameObject** basta atualizar o ângulo na **Transform**. O GUI irá rodar a imagem de acordo com esta informação.

Para rodar o colisor, se for um círculo não é necessário efetuar alguma operação.
Se for um polígono, rodam-se todos os vértices em torno do centroide de acordo com o seguinte algoritmo:

Rodar um ponto P(x_p , y_p) em torno de um centro C(x_c , y_c), no sentido anti-horário, por **g** graus. *(para rodar no sentido horário basta usar um ângulo negativo)*

#1
Mover o ponto C, o centroide, para a origem (0.0, 0.0) para simplificar o algoritmo:

$x_p = x_p - x_c$
$y_p = y_p - y_c$

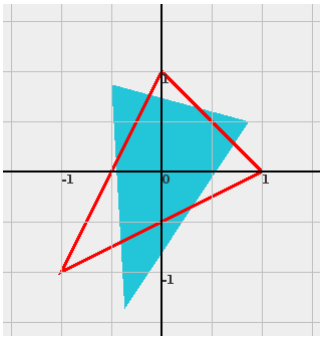
#2
Rodar o ponto P em torno da origem, em radianos:

rads = g * π / 180;

$x_n = x_p$ * cos(rads) - y_p * sin(rads);
$y_n = x_p$ * sin(rads) + y_p * cos(rads);

#3
mover o ponto P para a posição original:

$x_p = x_n + x_c$
$y_p = y_n + y_c$



Exemplo
Para o triangulo: { (-1.0, -1.0), (1.0, 0.0), (0.0, 1.0) }
Rodar 30 graus, vão-se obter os vértices:

{ (-0.37, -1.37), (0.87, 0.50), (-0.50, 0.87) }

Na figura a linha vermelha representa a posição original.

Calcular o centroide de um círculo
O centroide de um círculo é o seu centro.

Calcular o centroide de um polígono
Para n pontos ordenados no sentido horário: (x_0 , y_0), (x_1 , y_1), … (x_{n-1} , y_{n-1}), o centroide, ponto (x_c , y_c) é dado por:

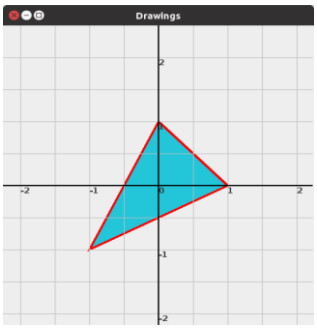
$$x_c = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{6A}(x_{n-1} + x_0)(x_{n-1} y_0 - x_0 y_{n-1})$$

$$y_c = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{6A}(y_{n-1} + y_0)(x_{n-1} y_0 - x_0 y_{n-1})$$

A parte a vermelho fecha o polígono.

A representa a área do polígono, que pode ser calculada com a fórmula de **shoelace** *(a parte a vermelho fecha o polígono)*:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}\left(x_i\, y_{i+1} - x_{i+1}\, y_i\right) + \frac{1}{2}\left(x_{n-1}\, y_0 - x_0\, y_{n-1}\right)$$



De notar que usando esta fórmula a área de um ponto ou linha é 0 como esperado.

Exemplo:
Para o triângulo: { (-1.0, -1.0), (1.0, 0.0), (0.0, 1.0) }
A área é: 1.50.
O centroide é: (0.0, 0.0).

Consulte agora os slides das Teóricas 8 e seguintes
antes de responder ao problema seguinte.

Problema L: Criação de GameObjects

Pretende-se implementar a criação de **GameObjects** como descritos acima. Primeiramente é necessário criar uma **Transform** e um **Collider**. Depois usando estes criar o **GameObject**, garantindo que o colisor tem o centroide na posição deste, dada por **transform.position()**. Se não o tiver terá de ser movido para esta posição. O colisor deverá também ser orientado de acordo a orientação na **Transform**, onde o ângulo zero representa o Norte e o ângulo é indicado no sentido anti-horário. Além disso, o colisor deverá ser escalado de acordo com o fator de escala na **Transform**. Para que o Collider possa responder desta forma, é necessário que o seu construtor receba uma referência para a mesma **Transform** que o **GameObject.**
Finalmente imprimir a representação do **GameObject** com **toString()** de acordo com a informação abaixo.

Entrada
A entrada do programa tem 3 linhas. A primeira tem uma string com o nome do **GameObject**.
A segunda linha tem a informação para criar a **Transform** com esta ordem: dois double para as coordenadas x e y, um int para a layer, um double para a rotação em graus e um double para o fator de escala.
A terceira linha tem a informação para criar o **Collider**. Se tiver 3 doubles representa um colisor circular com centro (x, y) nos 2 primeiros valores com raio igual ao terceiro. Se tiver 6 ou mais doubles o colisor será um polígono, e cada par de doubles representa um vértice, sendo estes ordenados no sentido horário.

Saída
Três linhas com a seguinte informação:

A primeira o nome do **GameObject**.
A segunda linha tem a informação guardada na Transform com esta ordem: dois double para as coordenadas x e y, um int para a layer, um double para a rotação em graus e um double para o fator de escala, com o formato: (x,y) layer angle scale.
A terceira linha tem a informação do **Collider** já centrado na posição do **GameObject**. Se for um círculo apresenta o centro e o raio no formato: (x,y) raio.
Se for um polígono apresenta os vértices por ordem horária com o formato: (x0,y0) (x1,y1) …
Todos os doubles devem ser impressos com 2 algarismos de precisão decimal.

Restrições complementares
Apresente o diagrama UML de todas as classes e interfaces desenvolvidos.
Para validar cada problema é necessário que sejam implementados os interfaces indicados acima.

Exemplo de Entrada 1
```
Alien01
1 2 1 0 1
2 2 3
```

Exemplo de Saída 1
```
Alien01
(1.00,2.00) 1 0.00 1.00
(1.00,2.00) 3.00
```

Exemplo de Entrada 2
```
Alien02
3 7 2 45.6 2
1 2 3
```

Exemplo de Saída 2
```
Alien02
(3.00,7.00) 2 45.60 2.00
(3.00,7.00) 6.00
```

Exemplo de Entrada 3
```
PlayerOne
5 9 0 90 2
2 2 2 6 4 6 4 2
```

Exemplo de Saída 3
```
PlayerOne
(5.00,9.00) 0 90.00 2.00
(9.00,7.00) (1.00,7.00) (1.00,11.00) (9.00,11.00)
```

Consulte agora os slides das Teóricas 8 e seguintes
antes de responder ao problema seguinte.

Problema M: Transformação de GameObjects

Pretende-se transformar, isto é mover, rodar e escalar, **GameObjects** como descrito acima. De notar que o ângulo zero representa o Norte e o ângulo é indicado no sentido anti-horário. Tanto a **Transform** como o **Collider** serão afetados pelas transformações.

Entrada
A entrada do programa tem 3 + n linhas, com n > 0
As 3 primeiras linhas definem um **GameObject** tal como a mesma sintaxe definida no problema anterior. As restantes linhas representam uma transformação a aplicar ao **GameObject** definido nas 3 primeiras linhas. As transformações podem ser de 3 tipos:

translação, iniciada pela string **move** e com um diferencial real para mover no eixo x e y e um diferencial inteiro para mover no layer:

move dx dy dlayer

rotação, iniciada pela string **rotate** com um angulo para rodar no sentido anti-horário a partir da orientação atual:

rotate dTheta

escala, iniciada pela string **scale** com um diferencial real para adicionar ao valor de escala corrente.:

scale dScale

Saída
Três linhas com a informação sobre o **GameObject** após transformado, tal como definido no problema anterior.

Restrições complementares
Apresente o diagrama UML de todas as classes e interfaces desenvolvidos.
Para validar cada problema é necessário que sejam implementados os interfaces indicados acima.

Exemplo de Entrada 1

```
Alien01
1 2 1 0 1
2 2 3
move 1 1 0
rotate 90
```

Exemplo de Saída 1

```
Alien01
(2.00,3.00) 2 90.00 1.00
(1.00,2.00) 3.00
```

Exemplo de Entrada 2

```
Alien02
0 0 2 0 1
1 2 3
move 3 7 2
scale 1
```

Exemplo de Saída 2

```
Alien02
(3.00,7.00) 4 0.00 2.00
(3.00,7.00) 6.00
```

Exemplo de Entrada 3

```
PlayerOne
7 9 0 0 1
2 2 2 6 4 6 4 2
move 3 7 2
rotate 90
scale 1
```

Exemplo de Saída 3

```
PlayerOne
(10.00,16.00) 2 90.00 2.00
(14.00,14.00) (6.00,14.00) (6.00,18.00) (14.00,18.00)
```