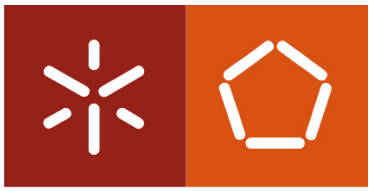

Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Sistemas de Representação Conhecimento e Racocínio

Exercício 1

Registo de eventos numa instituição de saúde

Grupo 19

Célia Natália Lemos Figueiredo
Aluna a67367

Gil Gonçalves
Aluno 67738

José Carlos Pedrosa Lima de Faria
Aluno a67638

Judson Quissanga Coge Paiva
Aluno E6846

Braga, 20 de Março de 2016

Conteúdo

Resumo

O presente relatório documenta o primeiro trabalho prático da Unidade Curricular de Sistemas de Representação Conhecimento e Racocínio. Nesta primeira fase o objetivo foi construir um mecanismo de representação de conhecimento para o registo de eventos numa instituição de saúde. Em termos gerais, foi usada a linguagem PROLOG, esta que utiliza um conjunto de fatos, predicados e regras de derivação de lógica. Uma execução de um programa é, na verdade, uma prova de um teorema, iniciada por uma consulta. Neste relatório pretende-se apresentar a forma como a aplicação foi construída bem como explicar algumas decisões tomadas.

O relatório encontra-se organizado em 3 partes. Na primeira parte apresentam-se as classes usadas, explicando qual a função de cada uma e das suas respectivas funções. Na segunda parte apresenta-se o gerador. Indica-se quais as figuras para as quais é possível gerar pontos bem como de que forma os pontos de cada figura são gerados. Na última parte apresenta-se o motor, nomeadamente a forma como os ficheiros de pontos são lidos, como os pontos são desenhados, bem como algumas considerações sobre a câmara e menus da aplicação.

1. Introdução

Nesta secção apresentam-se as classes usadas. Tanto o motor como o gerador recorrem extensivamente a estas classes para desempenhar as suas funções. Algumas destas classes são recorrentes e usadas mais que uma vez em diferentes contextos. Assim, antes de introduzir como funciona o motor e o gerador é importante primeiro perceber quais são as classes que ambos usam, bem como a função de cada uma.

O objectivo de construção destas classes foi simplificar a construção do motor e gerador apenas. Nesta primeira etapa, foram deixadas para segundo plano questões como a modularidade e encapsulamento de dados, embora seja algo a considerar em etapas posteriores do trabalho.

1.1 CoordsPolares

Esta classe pretende abstrair os cálculos relacionados com coordenadas polares. A ideia é poderem ser criadas instâncias desta classe indicando os parâmetros correspondentes a coordenadas polares e as correspondentes coordenadas cartesianas poderem ser obtidas facilmente a partir da instância.

Designando o ponto central como C(cx,cy,cz) e um ponto P(px,py,pz) que se pretende localizar, temos que as coordenadas px e py podem ser obtidas da seguinte forma:

$$px = cx + r \times sin(\alpha)$$
$$py = cy$$
$$pz = cz + r \times cos(\alpha)$$

A classe *CoordsPolares* representa apenas uma forma fácil de trabalhar com este sistema de coordenadas. Esta classe possui variáveis de instância relacionadas com as suas coordenadas polares e cartesianas:

```
//Centro a partir do qual se
//considera as coordenadas polares
Ponto3D centro;
//Coordenadas polares
float raio, azimuth;
//Coordenadas rectangulares correspondentes
//às coordenadas polares
Ponto3D cCartesianas;
```

O ponto chave desta classe é que as variáveis das coordenadas polares e cartesianas referem-se sempre ao mesmo ponto. Sempre que um dos parâmetros é alterado (através de uma das funções disponibilizadas), todos os restantes são actualizados em conformidade.

A classe tem um único construtor:

```
CoordsPolares(Ponto3D c, float r, float az);
```

Neste construtor são pedidos os parâmetros referentes às coordenadas polares. O primeiro parâmetro *c* corresponde ao ponto central, o segundo parâmetro r corresponde ao raio e o último parâmetro corresponde ao ângulo azimuth (α). Ao receber estas coordenadas polares, o construtor seguidamente atualiza as variáveis de instância em conformidade, nomeadamente coloca na variável de instância *cCartesianas* as coordenadas cartesianas correspondentes às coordenadas polares passadas como argumento. Essa actualização é feita pela função *refreshCartesianas()*:

```
void refreshCartesianas(){
        cRectangulares.x = centro.x + raio * sin(azimuth);
        cRectangulares.y = centro.y;
        cRectangulares.z = centro.z + raio * cos(azimuth);
}
```

Como se pode ver, esta função implementa apenas as fórmulas apresentadas anteriormente. Esta função é privada à classe, por isso não pode ser chamada em qualquer parte do código. É a própria classe que se responsabiliza por chamar esta função sempre que necessário.

Assim, dada uma instância desta classe é sempre possível saber as coordenadas cartesianas correspondentes através da função *toCartesianas()*:

```
Ponto3D toCartesianas() {
        return cCartesianas;
}
```

1.2 CoordsEsfericas

Esta classe pretende abstrair os cálculos relacionados com coordenadas esféricas. A ideia é poderem ser criadas instâncias desta classe indicando os parâmetros correspondentes a coordenadas esféricas e as correspondentes coordenadas cartesianas poderem ser obtidas facilmente a partir da instância. É também possível fazer o inverso, ou seja, indicar um ponto com coordenadas cartesianas e obter as respectivas coordenadas esféricas.

As coordenadas esféricas são constituídas por um raio r, por um ângulo θ (também designado por ângulo azimuth) e um ângulo ϕ (também designado por ângulo polar), conforme apresentado na figura **??**:

É possível saber as coordenadas cartesianas de um ponto P(px,py,pz) a partir das suas coordenadas esféricas através das seguintes fórmulas:

$$px = r \times cos(\theta) \times sin(\phi)$$
$$py = r \times sin(\theta) \times sin(\phi)$$
$$pz = r \times cos(\phi)$$

Por outro lado, consegue-se saber também as coordenas esféricas de um ponto a partir das suas coordenadas cartesianas de acordo com as seguintes fórmulas:

$$r = \sqrt{px^2 + py^2 + pz^2}$$
$$\theta = \arctan(py \backslash px)$$
$$\phi = \arccos(pz \backslash r)$$

De referir que ao contrário da classe *CoordsPolares*, nesta classe optou-se por não se considerar um centro. Assume-se o centro como sendo (0.0,0.0,0.0) sempre. Considerou-se esta simplificação razoável na medida em que responde aos requisitos do motor e gerador nesta primeira fase.

A classe *CoordsEsfericas* tem as seguintes variáveis de instância:

```
// Coordenadas Esfericas
float raio, azimuth_ang, polar_ang;
// Coordenadas cartesianas
Ponto3D cCartesianas;
```

O ponto chave desta classe é que as variáveis das coordenadas esféricas e cartesianas referem-se sempre ao mesmo ponto. Sempre que um dos parâmetros é alterado (através de uma das funções disponibilizadas), todos os restantes são atualizados em conformidade.

Uma instância da classe *CoordsEsfericas* pode ser criada indicando os parâmetros das coordenadas esféricas (para se saber as suas coordenadas cartesianas), ou indicando um ponto em coordenadas cartesianas (do qual se pretende saber as coordenadas esféricas), através dos seguintes construtores:

```
CoordsEsfericas(float r, float az, float polar);
CoordsEsfericas(Ponto3D pto);
```

Caso a instância seja criada a partir de coordenadas polares, o construtor calcula as coordenadas cartesianas correspondentes através da função *refreshCartesianas()*:

```
void refreshCartesianas() {
        cCartesianas.z = raio * sin(polar_ang) * cos(azimuth_ang);
        cCartesianas.x = raio * sin(polar_ang) * sin(azimuth_ang);
        cCartesianas.y = raio * cos(polar_ang);
}
```

Caso a instância seja criada a partir de coordenadas cartesianas, o construtor calcula as coordenadas polares correspondentes através da função *refreshEsfericas()*:

```
void refreshEsfericas() {
        raio = sqrt(pow(cCartesianas.x, 2) +
                pow(cCartesianas.y, 2) +
                pow(cCartesianas.z, 2));
        polar_ang = acos(cCartesianas.y / raio);
        azimuth_ang = atan2(cCartesianas.x, cCartesianas.z);
}
```

Estas duas funções correspondem à implementação das fórmulas apresentadas anteriormente e garantem que as variáveis de instância correspondentes às coordenadas esféricas e cartesianas se referem sempre ao mesmo ponto. Ambas são funções privadas, pelo que é a própria classe que tem a responsabilidade de as chamar sempre que é necessário atualizar valores.

A qualquer momento, é possível saber as coordenadas cartesianas de uma instância pela função *toCartesianas()*

```
Ponto3D toCartesianas() {
        return cCartesianas;
}
```

Além destas funções, a classe possui ainda funções adicionais que permitem mudar a posição do ponto representado por cada instância. Sempre que uma destas funções é chamada tanto as variáveis de instância das coordenadas polares e cartesianas são atualizadas quer pela função refreshEsfericas() ou refreshCartesianas(). Estas funções que permitem mudar a localização do ponto, garantem ainda que $0 \leq \phi \leq \pi$ e que $0 \leq \theta \leq 2 \times \pi$

1.3 Figura

Esta classe representa uma figura que será desenhada pelo motor. Representa por isso apenas um conjunto de pontos numa determinada ordem que correspondem a triângulos, que por sua vez formam uma figura.

Por representar um conjunto de pontos, sem surpresa, a sua única variável de instância é um vector de pontos:

```
std::vector<Ponto3D> pontos;
```

A utilidade desta classe revela-se pelas funções que disponibiliza. Em primeiro lugar, disponibiliza um conjunto de funções que quando chamadas colocam no vector *pontos* os pontos necessários para o desenho de uma figura em concreto. Dessa forma, estas funções permitem criar planos, caixas, círculos, cilindros e esferas. Estas funções serão explicadas em mais detalhe quando for apresentado o gerador onde será mostrado de que forma estas funções podem ser chamadas bem como de que forma geram os pontos.

Além das funções que permitem criar figuras destaca-se ainda a função *toFicheiro()*, que permite guardar os pontos da figura num ficheiro cujo nome é passado como argumento.

```
void toFicheiro(std::string filePath)
```

É ainda possível obter os pontos da figura pela função *getPontos()*:

```
std::vector<Ponto3D> getPontos()
```

1.4 TinyXML-2

Esta biblioteca foi usada para auxílio à leitura de ficheiros XML por parte do motor e pode ser encontrada no endereço: http://www.grinninglizard.com/tinyxml2/

2. Conclusão

O trabalho apresentado cumpre todos os requisitos propostos. Foi feito um gerador de figuras com capacidade de gerar pontos para um plano, caixa, cone e esfera como pedido. Além destas figuras foram ainda desenvolvidas funções adicionais que permitem criar planos sem ser no eixo XZ bem como foram desenvolvidas funções para a criação de círculos e cilindros, algo que não era expressamente pedido.

No lado do motor, a aplicação consegue ler ficheiros XML e .3D e a partir dai desenhar as figuras com os pontos especificados nos ficheiros. Adicionalmente ao sugerido, incluiu-se também uma câmara colocada sobre uma esfera que permite ao utilizador ver a figura desenhada de vários ângulos. Incluiu-se ainda um pequeno menu para alterar o modo de visualização da figura.

O código produzido recorreu ao uso de classes, o que evitou bastantes repetições de código e sobretudo gerou um código fácil de ler, perceber e manter. Por estes motivos, considera-se como bastante sólido o trabalho desenvolvido.

Não obstante, existem aspectos em que o trabalho que poderiam ser melhorados e serão alvo de atenção no futuro.

Em primeiro lugar, destaca-se a questão da câmara. Nesta fase implementou-se a câmara usando coordenadas esféricas. Decidiu-se que a camâra seria por isso apenas uma instância da classe *CoordsEsfericas*. Deste modo mover a câmara corresponde apenas a chamar as funções definidas na classe. Este aspecto facilitou imenso a implementação da câmara, no entanto trouxe também algumas desvantagens. Sendo a câmara uma instância de *CoordsEsfericas* significa que a câmara só pode ter coordenadas esféricas. Isto dificulta a adição de funcionalidades extra à câmara. Além disso, enquanto que é perfeitamente válido que as coordenadas esféricas possam referir um ponto "no polo norte" da esfera, tal não é verdade para a câmara, pois nesse caso o objeto pode deixar de se tornar visível. Isto deixa a entender que no futuro a câmara terá que pertencer a uma classe própria e muito provavelmente será esse o caminho a seguir.

Em segundo lugar, refere-se a modularidade e encapsulamento de dados, aspectos que foram deixados um pouco para segundo plano. A prioridade desde cedo foi ter código simples, funcional e fácil de ler. Tal implicou que muitas vezes quando confrontados com a decisão de manter algumas variáveis como públicas ou privadas a decisão tenha sido manter públicas. Exemplos disso são as classes Ponto3D (note-se a falta de getters e setters) e as classes das coordenadas esféricas e polares. Enquanto que ter variáveis públicas em classes como a Ponto3D seja relativamente irrelevante, tal já não é verdade para as classes das coordenadas. Como o objetivo principal do projeto não é ter bons módulos de dados nem bom encapsulamento dos mesmos, estes aspectos foram deixados para segundo plano, no entanto serão alvo de uma atenção mais cuidada no futuro.