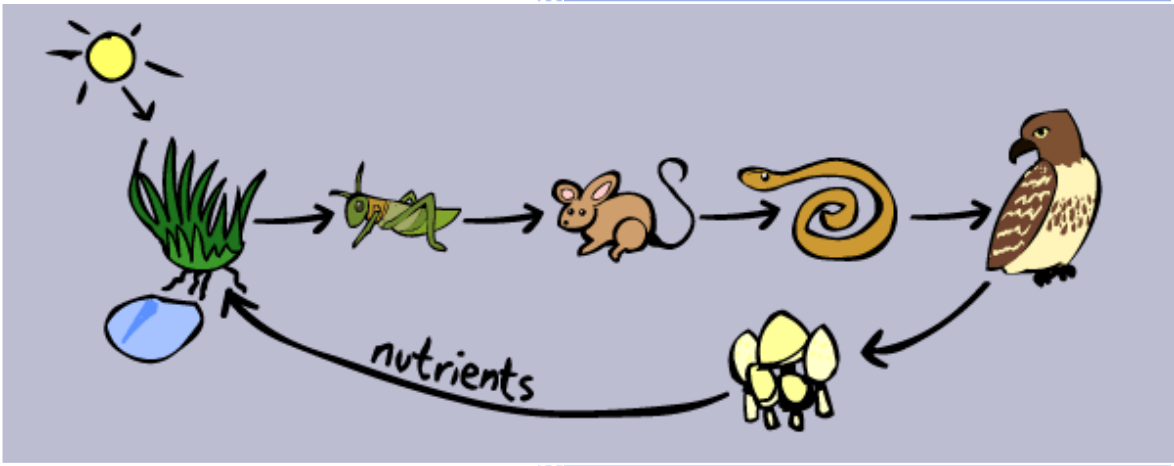
MSSN

Simulação de Ecossistemas

A Cadeia Alimentar



Discente Jorge Fernandes a39372

Docente Arnaldo Abrantes

Instituto Superior de Engenharia de Lisboa

Fevereiro de 2014

Índice

Índice de Ilustrações

Objetivos

É objetivo deste projeto a consolidação dos conteúdos lecionados na Unidade Curricular de Modelação e Simulação de Sistemas Naturais, assim como a compilação da execução prática de diversos exercícios e módulos desenvolvidos A priori em aula.

Tendo em vista o desenvolvimento de um modelo de simulação de um sistema natural, apresentam-se seguidamente as bases para o desenvolvimento do mesmo assim como os principais focos de estudo:

- Modelação de sistemas do tipo presas-predadores;
- Simulação de movimento, baseada nas leis da dinâmica de Newton;
- Modelação dos comportamentos de agentes autónomos;
- Simulação de processos dinâmicos através de autómatos celulares;
- Programação em python usando o paradigma orientado a objetos;
- Utilização da API pygame para a construção de jogos/simulações

Pretende-se ainda que o utilizador possa ser agente ativo na simulação, podendo assim interferir nas populações das diferentes raças, de modo a provocar ou evitar a extinção.

Modelo Base

O modelo base e todas as suas disposições estão expostos no enunciado deste trabalho prático. É assim dispensável a abordagem detalhada dos mesmos. Expõe-se assim o diagrama de classes do modelo sobre o qual se basearão todas as extensões desenvolvidas nas posteriores versões.

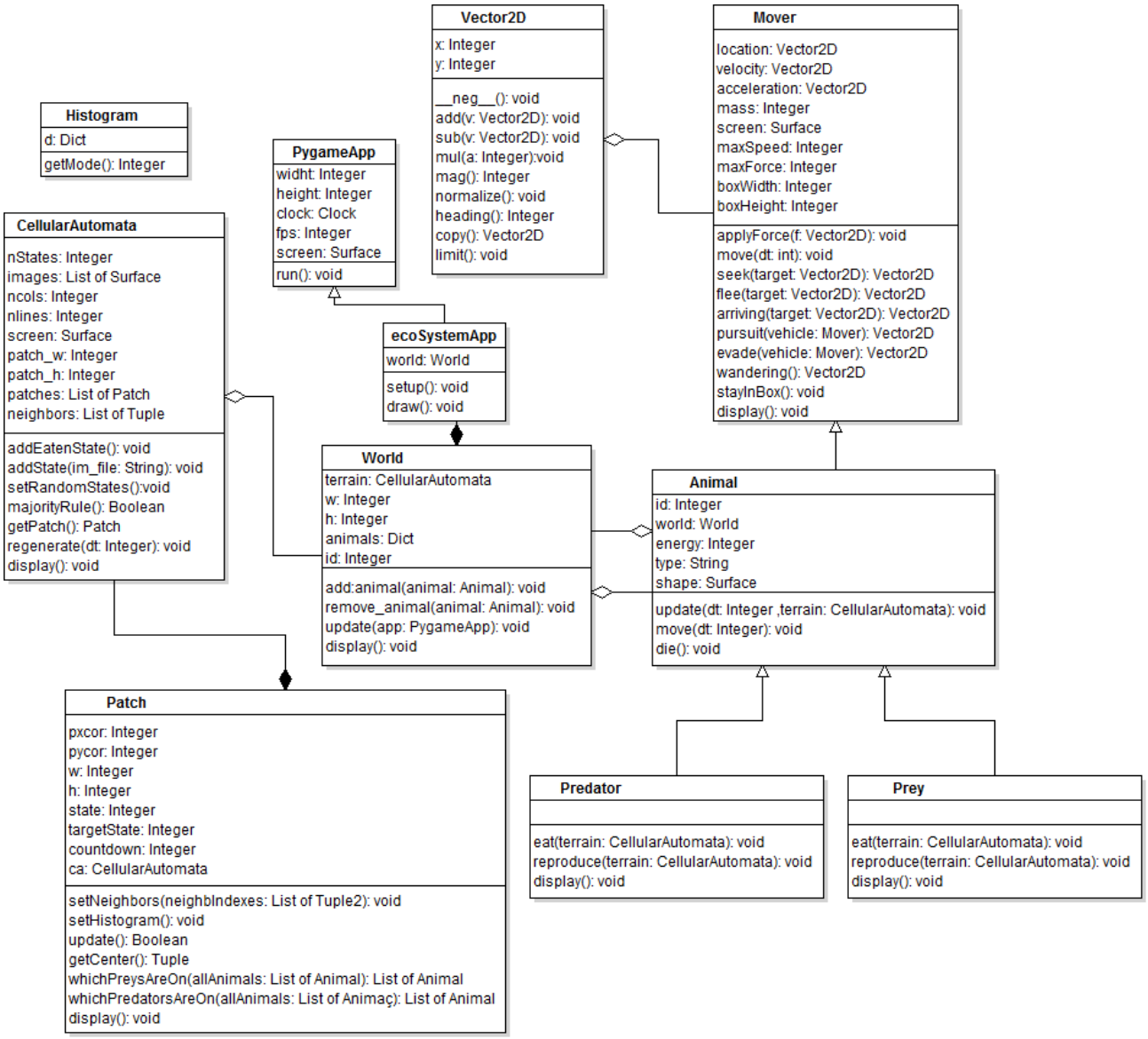


Ilustração 1 - UML da versão base

Versão 1

Na versão base existem apenas dois tipos de animais, sendo estes presas e predadores. Nesta nova versão faz-se a extensão do número de populações distintas, criando-se duas subclasses de presas e duas subclasses de predadores.

Com esta alteração, tornou-se significativo efetuar a substituição dos métodos whichPreysAreOn(self,allAnimals) e whichPredatorsAreOn(self,allAnimals) por um método global whichAnimalsAreOn(self, allAnimals, races), que devolve a lista de todos os animais de determinada(s) raça(s). Adicionou-se assim à classe animal o atributo race. A definição do método encontra-se exposta abaixo.

Foi necessária ainda a criação de uma nova classe que compilará todas as funcionalidades relativas à criação e gerência de dados para a criação de um gráfico de atualização em tempo real que facilitará a visualização e estudo da evolução das populações. Com esta classe permitiu-se a criação de um nível de abstração que facilita a criação do gráfico na aplicação já desenvolvida.

Adicionando ainda complexidade a nível dos movimentos baseados nas leis da dinâmica de Newton, integrou-se o conceito de zonas nas quais os animais evitam de entrar, baseado no estado dos Patches para os quais o animal se desloca. O algoritmo implícito no método criado assim como a função serão dilucidados nesta secção.

```
 def obstacleAvoidance(self, nPixels, intPixels, states):
    a = arange(0, nPixels, intPixels)
    vAux = Vector2D(self.velocity.x, self.velocity.y)
    vAux.normalize()
    xBase=vAux.x
    yBase=vAux.y
    for i in a:
       patch = self.world.terrain.getPatch(int(self.location.x+i*xBase),int(self.location.y+i*yBase))
       if patch.targetState in states:
          break
    if i < a[-1]:
       v2 = Vector2D(-vAux.y, vAux.x)
       v2.mul(0.5)
       v2.add(vAux)
       v2.normalize()
       xBase2=v2.x
       yBase2=v2.y
       patch =
self.world.terrain.getPatch(int(self.location.x+i*xBase2),int(self.location.y+i*yBase2))
       if patch.targetState in states:
          vector = Vector2D(self.velocity.y, -self.velocity.x)
       else:
          vector = Vector2D(-self.velocity.y, self.velocity.x)
       vector.normalize()
       vector.mul(max((1.-(1.*i/nPixels))*self.maxForce*2, self.maxForce))
       return vector
    else:
       return Vector2D( 0, 0)
```

Detalhando o funcionamento do método anterior, nesse verifica-se o estado dos Patches que se situam em frente ao animal em linha reta, com um step de intPixels e até uma distância de nPixels. Caso se encontre algum Patch em algum dos estados contidos na lista states a procura termina, passando-se à verificação de qual o sentido mais conveniente para alterar o movimento. Pode verificar-se pela imagem-exemplo seguinte o raciocínio implícito no algoritmo, no qual se considerou que fazendo uma verificação do patch equidistante ao patch encontrado, mas segundo um angulo diferente, caso esse Patch esteja no estado pretendido, o desvio deverá de ser feito no sentido inverso ao mesmo, seguindo-se também o raciocínio inverso.

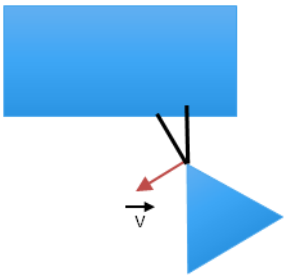


Ilustração 2 - Imagem exemplo do algoritmo obstacleAvoidance()

Apresenta-se de seguida o método de obtenção dos animais presentes em determinado Patch, cuja race esteja na lista races:

```
def whichAnimalsAreOn(self,allAnimals, races):
    return [animal for animal in allAnimals
        if (self.ca.getPatch(animal.location.x,animal.location.y) == self and animal.race in races)]
```

Apresenta-se finalmente na página seguinte o UML alterado em consonância com o exposto.

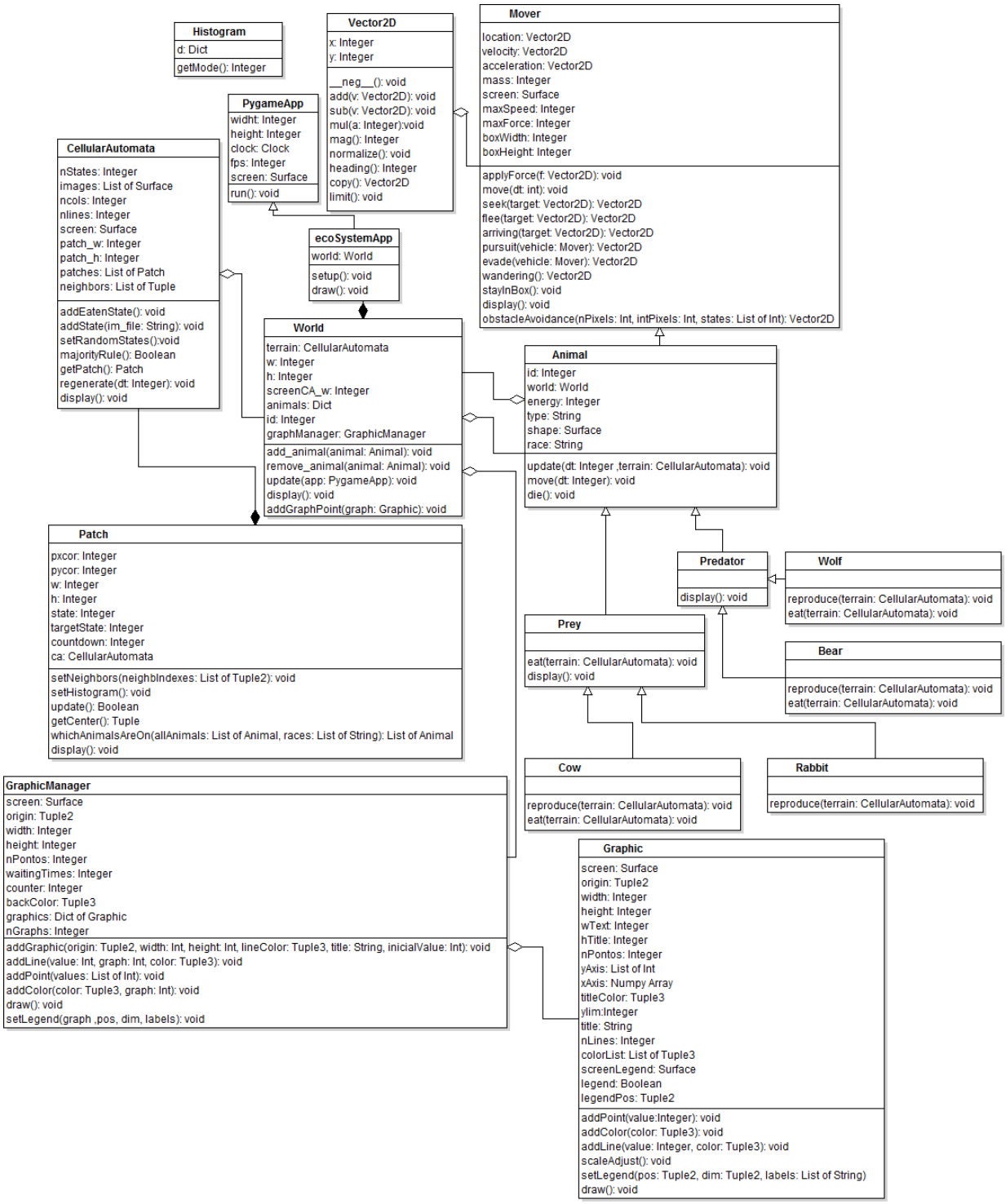


Ilustração 3 - UML da versão 1

É de relevante importância neste estudo a cadeia alimentar adotada, a qual segue o seguinte diagrama:

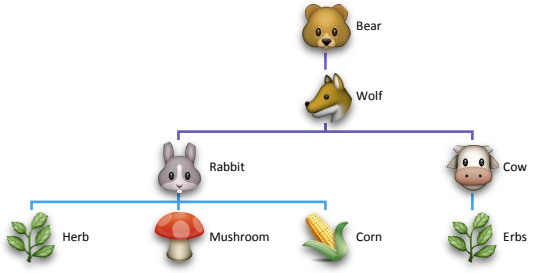


Ilustração 4 - Cadeia alimentar

Versão 2

Nesta segunda versão fez-se uma grande alteração e de relevante importância na visão da dinâmica de sistemas naturais, dado que se tem em conta o comportamento animal em função das suas necessidades básicas.

Neste ponto é adicionada a noção de procura de alimento. Assim, quando o animal atinge um valor de energia menor que um limiar de fome pré-estabelecido para cada subclasse de Animal, este deixa de se movimentar segundo o comportamento wandering() e passa a procurar uma presa para caçar caso seja um Predador, ou um Patch que possa constituir alimento, caso seja uma presa.

O comportamento de movimento de busca de alimento é diferente para presas e predadores. Achou-se conveniente que a classe dos Predadores caçasse movimentando-se segundo o movimento pursuit, dado que capturará o alvo em movimento, e que as presas seguissem o comportamento arriving. Deste modo, o predador consegue prever qual a localização futura da sua caça e em contrariedade, dado que o alimento dos animais da classe presa se encontra em repouso o animal, este imobilizar-se-á ao atingir a posição do Patch, a fim de se alimentar.

Achou-se pertinente ainda que aquando atingido o limiar da fome, o valor de energia do animal se tornasse visível ao utilizador, conforme se pode verificar na secção Resultado Gráfico. É nesta situação que o animal busca alimento.

Para tal, foi necessário definir ainda como fazer a procura do alimento e a afetação do target. Para ambas as classes, presas e predadores, é efetuada uma pesquisa nos Patches vizinhos, verificando se nesse existe alimento disponível. Caso exista, a posição desse alimento é afetado como target do movimento. Denota-se que se tomou em consideração a procura ser efetuada primeiramente nos Patches mais próximos.

Apresenta-se nas páginas seguintes dois exemplos do método de decisão e afetação do alvo a seguir para uma presa e um predador, respetivamente (neste caso vaca e urso), assim como o diagrama de UML com as alterações.

```
    def decide(self):
        if self.energy < self.ungry_Limit and self.targetPatch== None:
            (col,line) = self.world.terrain.getPatchCoor(self.location.x, self.location.y)
            for (x,y) in FOOD_SEARCH_NEIGH:
                col += x
                line+=y
                if col<0 or line<0 or col>=self.world.terrain.ncols or line>=self.world.terrain.nlines:
                    continue
                patch = self.world.terrain.patches[col][line]
                if patch.state==2 and ENERGY_FROM_FOOD[patch.state]>0:
                    self.targetPatch = patch
                    break
        if self.energy < self.ungry_Limit and self.targetPatch!= None:
            f=Vector2D(0,0)
            if ENERGY_FROM_FOOD[self.targetPatch.state] ==0:
                self.targetPatch=None
            else:
                f = self.arriving(Vector2D
((self.targetPatch.pxcor*self.targetPatch.w)+10,(self.targetPatch.pycor*self.targetPatch.h)+10))
        else:
            f = self.wandering()
        return f
```

```
    def decide(self):
        if self.energy < self.ungry_Limit and self.targetPrey== None:
            (col,line) = self.world.terrain.getPatchCoor(self.location.x, self.location.y)
            for (x,y) in FOOD_SEARCH_NEIGH:
                col += x
                line+=y
                if col<0 or line<0 or col>=self.world.terrain.ncols or line>=self.world.terrain.nlines:
                    continue
                patch = self.world.terrain.patches[col][line]
                preys = patch.whichAnimalsAreOn(self.world.animals.values(),('wolf'))
                if any(preys):
                    self.targetPrey = preys[0].id
                    break
        if self.energy < self.ungry_Limit and self.targetPrey!= None:
            try:
                f=Vector2D(0,0)
                prey = self.world.animals[self.targetPrey]
                f = self.pursuit(prey)
            except KeyError:
                self.targetPrey=None
        else:
            f = self.wandering()
        return f
```

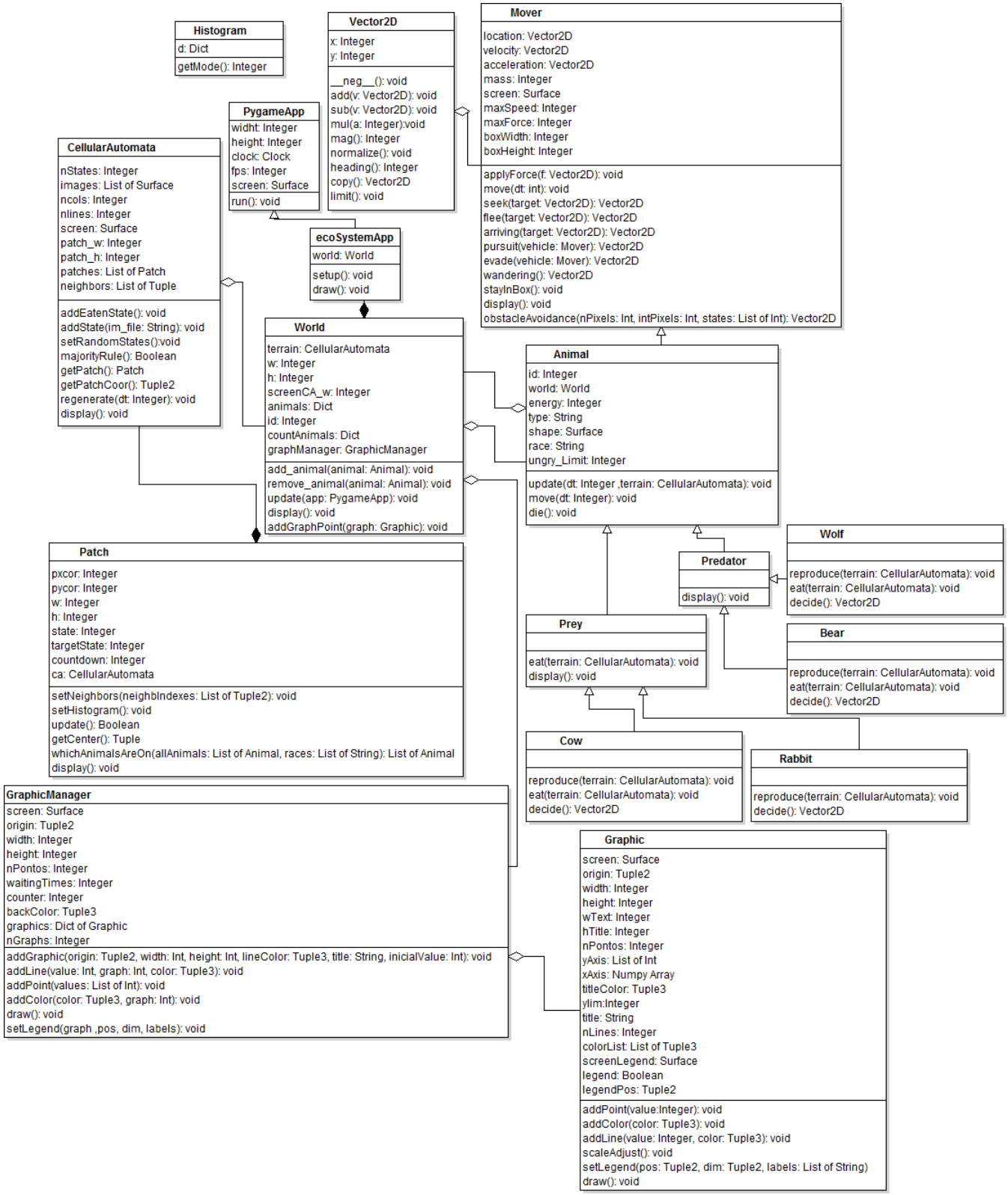


Ilustração 5 - UML da versão 2

Versão 3

Nesta terceira versão adiciona-se outro comportamento instintivo animal, a fuga dos predadores. Naturalmente, qualquer animal tende a afastar-se dos seus predadores. O mesmo comportamento foi adicionado na simulação. Considera-se ainda que em situações de fome o animal considera prioritário alimentar-se, correndo nessa situação o risco de ser apanhado pelo seu predador.

Para efetuar esta implementação, foi necessário alterar apenas algumas das classes derivadas de Animal. Considerou-se ainda que o movimento de fuga seria do tipo flee.

Deste modo, nos métodos decide já definidos e explícitos na secção anterior faz-se a chamada a um novo método denominado escape, aquando o animal se encontra a vaguear. Este método retorna uma força, a força correspondente ao somatório do comportamento flee para com todos os seus predadores presentes na vizinhança, que será adicionada à força resultante do método wandering, e que será a responsável pelo movimento.

Apresenta-se de seguida o método referido que busca todos os predadores na vizinhança e calcula a força resultando de repulsão a todos esses predadores:

```
    def escape(self, animalsRace):
        (col,line) = self.world.terrain.getPatchCoor(self.location.x, self.location.y)
        EscapeAnimals=()
        for (x,y) in FOOD_SEARCH_NEIGH:
            col += x
            line+=y
            if col<0 or line<0 or col>=self.world.terrain.ncols or line>=self.world.terrain.nlines:
                continue
            EscapeAnimals = set(EscapeAnimals).union(set(self.world.terrain.patches[col][line].
whichAnimalsAreOn(self.world.animals.values(),animalsRace)))
        force=Vector2D(0,0)
        for animal in EscapeAnimals:
            f = self.flee(animal.location)
            force.add(f)
        return force
```

Dado que a única alteração ao programa foi a adição deste método não se considera significativo redemonstrar o UML da aplicação.

Versão 4

Nesta última versão fez-se a compilação dos comportamentos da versão anterior com o comportamento concebido na primeira versão, na qual os animais evitam entrar em determinados Patches.

Desenvolveu-se ainda uma nova classe que permite a apresentação de um menu inicial no qual o utilizador tem a oportunidade de definir os valores iniciais das diversas populações.

Nesta versão o utilizador torna-se também parte atuante na simulação, na medida em que pode adicionar animais de cada espécie e extinguir também cada uma das espécies. É-lhe permitido a cada momento também começar uma nova simulação, tendo como ponto de partida o menu inicial com a decisão dos valores iniciais.

Considerou-se ainda, há semelhança do que acontece com a alimentação das presas, definir diferentes níveis nutritivos para cada raça de animal ingerido. Estes valores de nutrição serão definidos no ficheiro params.py, ficheiro este onde se pode fazer também a alteração dos metabolismos dos diversos animais.

Nesta ultima instância de desenvolvimento da aplicação de simulação do sistema verificou-se algumas potenciais melhorias, as quais foram efetuadas, mesmo estando afetas a versões anteriores. Passa a enumerar-se:

- Alteração do método decide com vista a tirar melhor proveito da herança de métodos nas classes que estendem de Prey e Predator – Melhoria na redução do número de linhas de código, evitando a definição repetida de métodos muito idênticos;
- Alteração do método escape, que após a redefinição passa a verificar a existência de animais da raça a fugir, através do contador dos mesmos, evitando assim a iteração sobre todos os Patches vizinhos caso não existam predadores – Melhoria na redução do número de tarefas e consequentemente na redução do uso de recursos da máquina.

```
    def escape(self, animalsRace):
        (col,line) = self.world.terrain.getPatchCoor(self.location.x, self.location.y)
        EscapeAnimals=()
        n=0
        for race in animalsRace:
            print int(countAnimals[race])
            n += int(countAnimals[race])
        if n==0:
            return Vector2D(0,0)
        for (x,y) in FOOD_SEARCH_NEIGH:
            col += x
            line+=y
            if col<0 or line<0 or col>=self.world.terrain.ncols or line>=self.world.terrain.nlines:
                continue
            EscapeAnimals = set(EscapeAnimals).union(set(self.world.terrain.patches[col][line].
whichAnimalsAreOn(self.world.animals.values(),animalsRace)))
        force = Vector2D(0,0)
        for animal in EscapeAnimals:
            f = self.flee(animal.location)
            force.add(f)
        return force
```

```
def decide(self):

    return self.decidePredator( ["bear"], ('cow','rabbit'))
```

```
def decide(self):

    return self.decidePrey(["wolf"], range(6))
```

Apresenta-se assim o UML dos Sistema de Simulação Terminado:

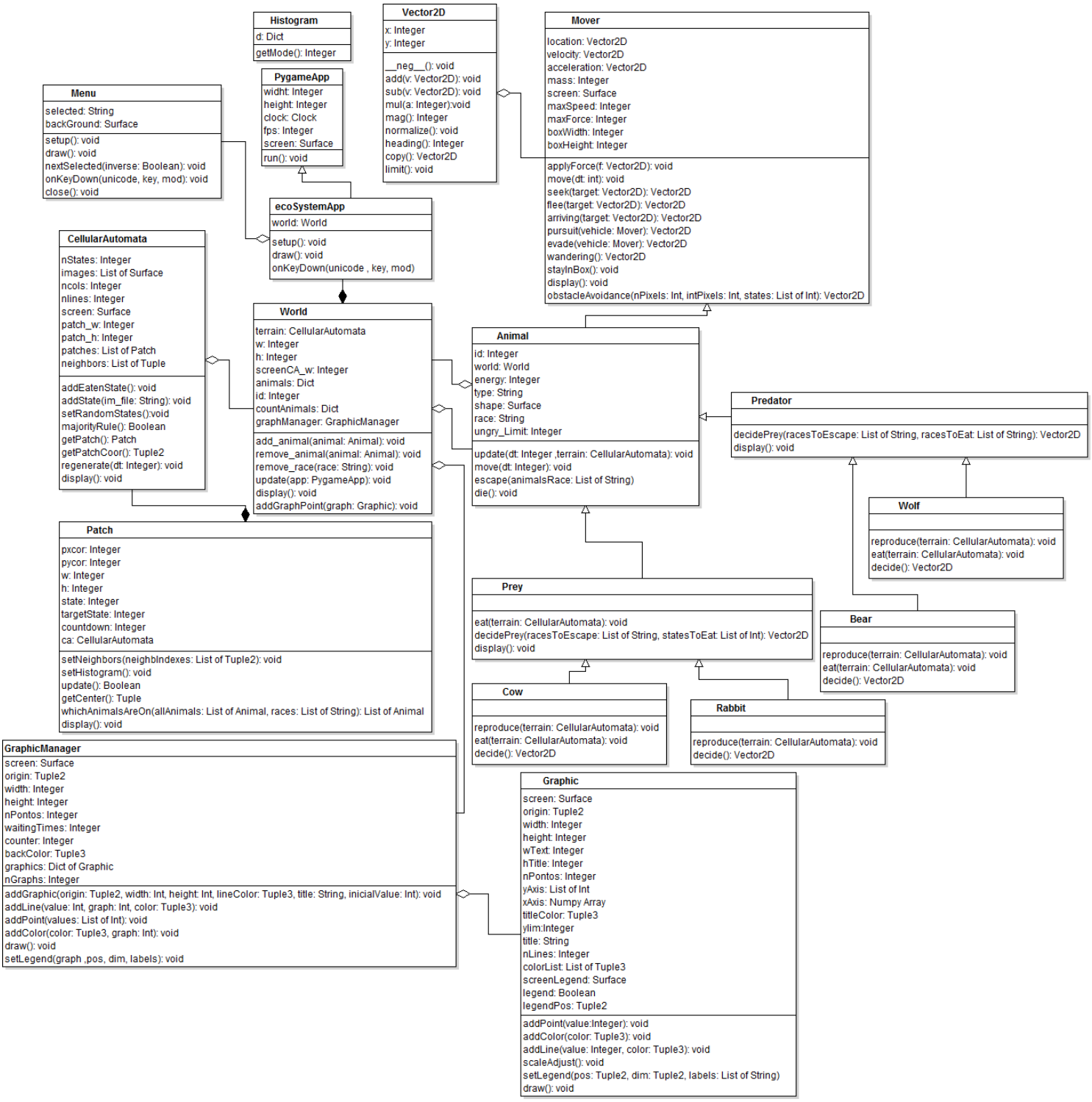


Ilustração 6 - UML da versão 4

Resultado Gráfico

Apresenta-se na presente secção o resultado gráfico da aplicação de simulação desenvolvida.

Nas primeiras duas figuras, relativas á versão base do projeto, pode verificar-se o carácter pseudo-aleatório da inicialização do ambiente. Verifica-se ainda a existência de apenas uma raça de animais da casse Prey e Predator.

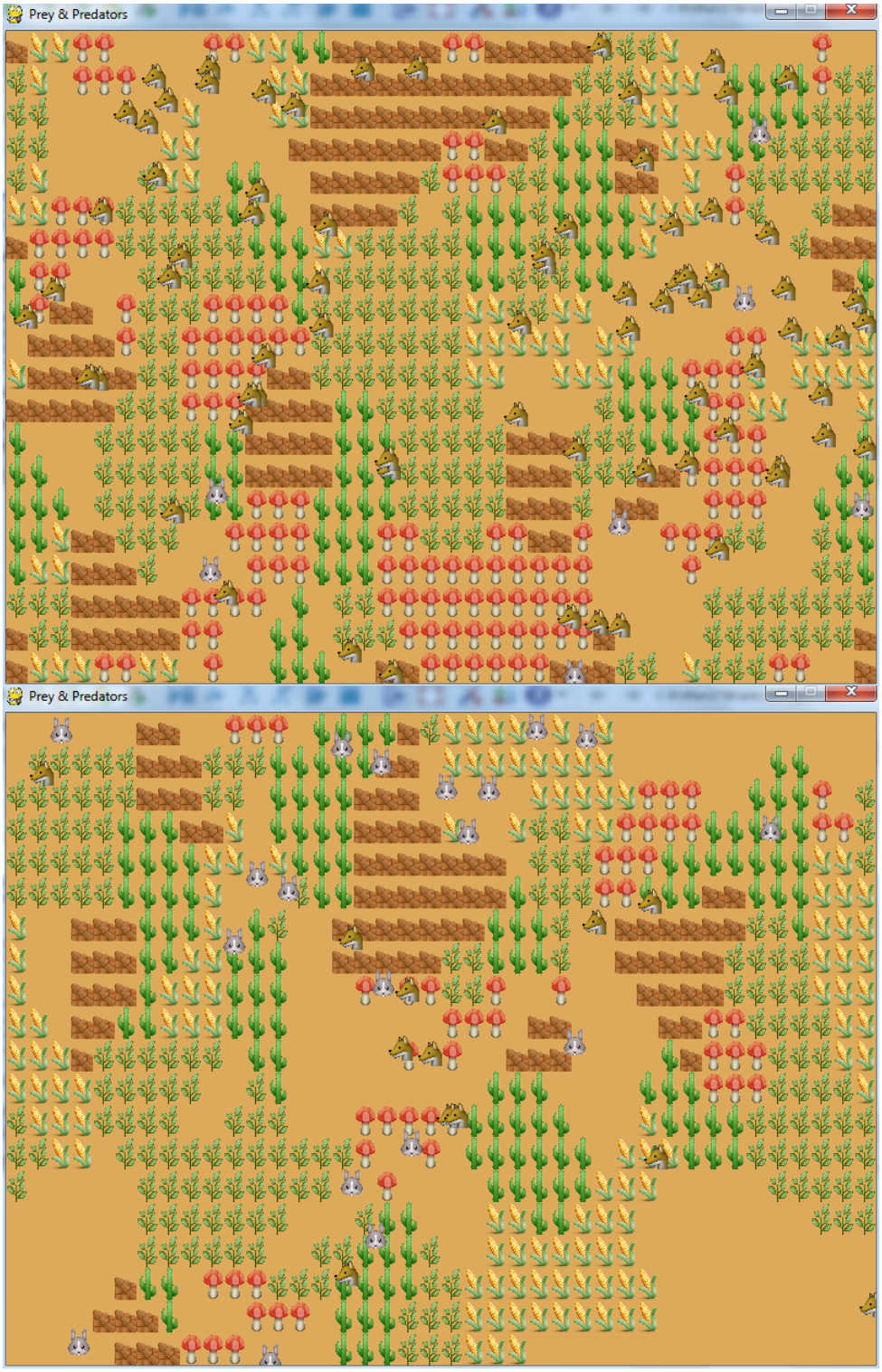


Ilustração 7 - Diferentes inicializações do Ecossitema

As seguintes ilustrações são relativas à versão final no trabalho. Primeiramente demonstra-se a interface gráfica relativa ao menu, seguida do ambiente de simulação com o respetivo gráfico. Apresenta-se ainda o pormenor da informação de energia do Animal quando este atinge o limiar da fome e passa a procurar alimento.

Ilustração 8 - Menu inicial

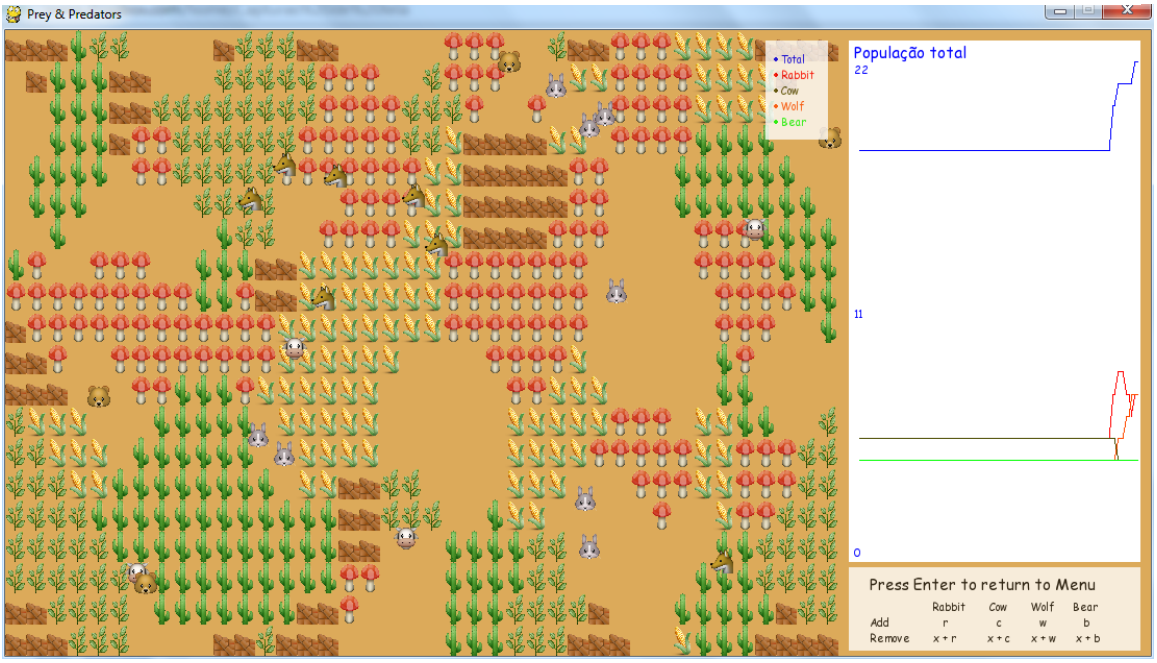


Ilustração 9 - Ambiente de simulação

Ilustração 10 - Pormenor de animal em busca de alimento

Simulações

Simulação tendo em conta iguais metabolismos

Numa primeira simulação todos os animais concediam o mesmo ganho energético ao seu predador, assim como todos os animais tinhas os mesmos valores de energia inicial, energia de reprodução e limiar de fome. Estes factos tornavam o sistema pouco fiel à realidade, dado que na natureza as diferentes espécies têm diferentes metabolismos.

Observações registadas:

- Domínio da população de coelhos, devido à abundância de alimentos para esta espécie;
- Mesmo que próximo da extinção, basta ao coelho uma situação de fraca população de predadores para que este num curto período de tempo passe a predominar;
- É bastante comum que a população de coelhos tenda para um ponto fixo de média 220 elementos, devido à limitação de alimentos, aquando a extinção de todas as restantes espécies;
- O aumento da população de coelhos é acompanhado pelo aumento da população de raposas, quando a população de coelhos é considerável para que não fique distribuída de modo rarefeito no ecrã;
- Acontece a extinção de todas as espécies quando a repousa predomina sobre o coelho, pois esta leva à extinção os coelhos e fica sem alimento.
- Caso ocorra a extinção de vacas e coelhos, todas as espécies extinguem;
- É comum a extinção da população de vacas dada a pouca quantidade de alimento disponível, que por sua vez é partilhada com a população de coelhos, muito mais abundante. Sempre que a população de coelhos domina a população de vacas tende para a extinção;
- Logicamente, o aumento de uma determinada espécie causa a diminuição das populações das espécies das quais esta primeira se alimenta.

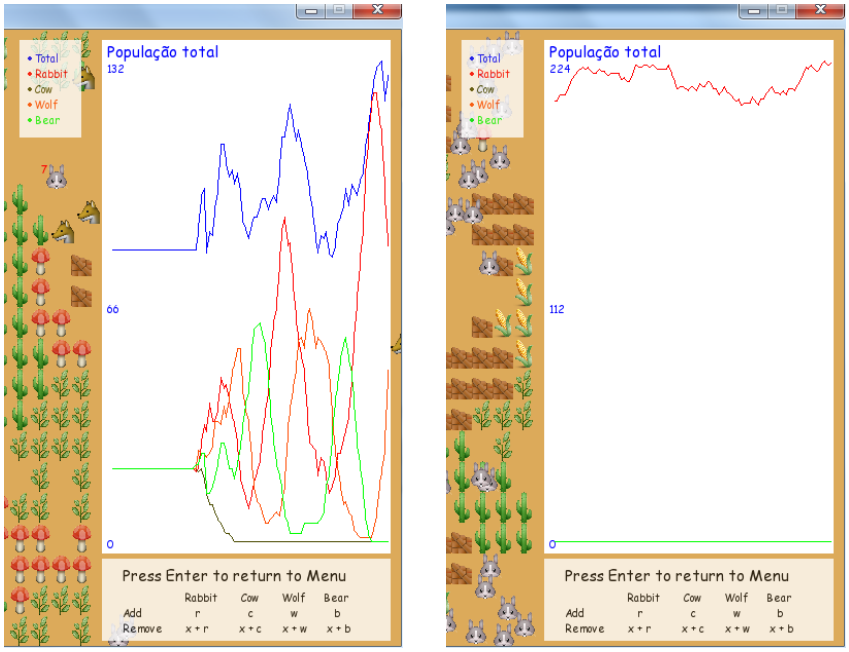


Ilustração 11 - Gráficos de população em função do tempo

Considerando os diferentes metabolismos

Fez-se a alteração do sistema de modo a que este possa ser o mais fiel à realidade possível, tendo em conta a complexidade do mesmo.

Considerando-se:

```
INITIAL_PREY_ENERGY = 20
ENERGY_TO_REPRODUCE_PREY = 40
ENERGY_UNGRY_PREY = 10

INITIAL_PREDATOR_ENERGY = 20
ENERGY_TO_REPRODUCE_PREDATOR = 40
ENERGY_UNGRY_PREDATOR = 10

INITIAL_WOLF_ENERGY = 20
ENERGY_TO_REPRODUCE_WOLF = 30
ENERGY_UNGRY_WOLF = 10

INITIAL_BEAR_ENERGY = 30
ENERGY_TO_REPRODUCE_BEAR = 40
ENERGY_UNGRY_BEAR = 15

INITIAL_RABBIT_ENERGY = 20
ENERGY_TO_REPRODUCE_RABBIT = 40
ENERGY_UNGRY_RABBIT = 10

INITIAL_COW_ENERGY = 30
ENERGY_TO_REPRODUCE_COW = 40
ENERGY_UNGRY_COW = 15

ENERGY_FROM_FOOD = {0:0,1:0,2:4,3:6,4:10,5:0,6:0,7:0}
ENERGY_FROM_ANIMAL = {'cow':50,'wolf':40,'bear':60,'rabbit':30}
```

Observações registadas:

- As observações são idênticas às registadas no ponto anterior, verifica-se apenas um melhor equilíbrio dinâmico nas dimensões das populações das diferentes espécies, conseguindo-se os objetivos desta alteração.

Conclusões

Na conclusão deste trabalho, em suma podem dar-se por cumpridos todos os objetivos do mesmo. Verificou-se a boa funcionalidade de todos os algoritmos e funções, quer quando testados por partes nas diversas versões desenvolvidas, quer quando compilados na versão final, estando a fase de testes já expressa no bloco anterior presente neste relatório.

Pode assim verificar-se de forma visual e mais intuitiva a ocorrência de diversos comportamentos típicos de sistemas naturais quer abordados em aula quer estudados anteriormente na secção de estudo recorrendo à ferramenta de simulação Vensim, assim como a utilização dos diversos comportamentos de agentes autónomos estudados numa aplicação útil.