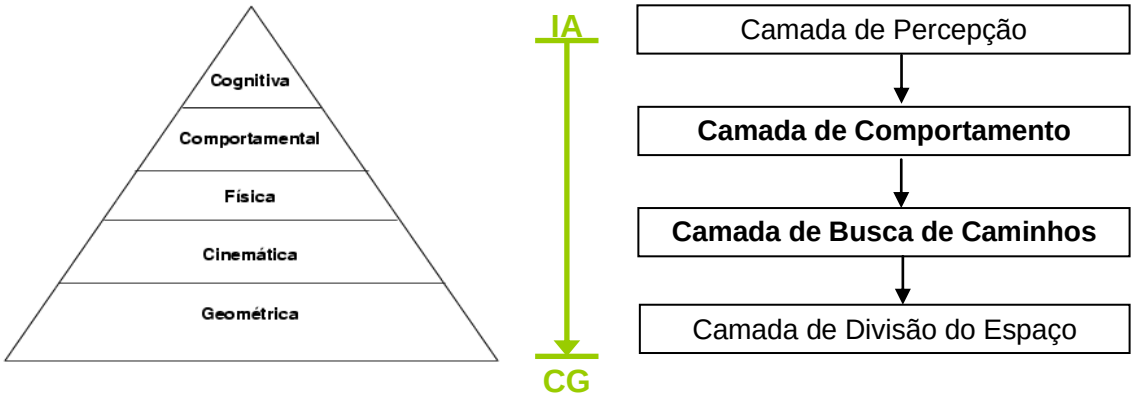
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
20/10/2007

Direcionamento de Personagens



1. Busca de Caminhos

Os algoritmos de busca de caminho, como o A* e Dijkstra encontram caminhos mais curto entre dois pontos de um grafo que pode representar um cenário qualquer. Como resultado, são geradas seqüências de nós (*waypoints*) que o agente deve percorrer para chegar ao destino.

Até este momento, não foi discutido como o agente pode seguir este caminho, ou seja, como ele deve se locomover entre cada par de nós. Nesta seção são apresentadas técnicas voltadas a este propósito, chamadas *steering behaviors* (Comportamentos de direcionamento).

Steering Behaviors podem ser usados para refinar o caminho gerado pelos algoritmos de *path-finding*, bem como para comportamentos de movimentação mais reativos, como fugir ou perseguir um inimigo nas proximidades, evitar colisão com paredes, bem como controlar a movimentação de grupo de indivíduos.

2. Estrutura das entidades móveis

Steering Behaviors fazem uso de forças para determinar a direção que os personagens. Essas forças são representadas por vetores 2D, que alteram a direção e velocidade do personagem em direção ao alvo selecionado. São usados conceitos físicos, já vistos em aula, para controlar este comportamento.

Utiliza-se um modelo de massa pontual, onde:

- A posição é ajustada pela velocidade
- A velocidade é ajustada por forças de direcionamento.

O método `update()` da classe Actor ilustra o processo utilizado na determinação da nova direção e velocidade do personagem, em função de uma força gerada pelos Steering Behaviors. O método `render()` é usado para desenhar o personagem, além de informações das forças aplicadas.

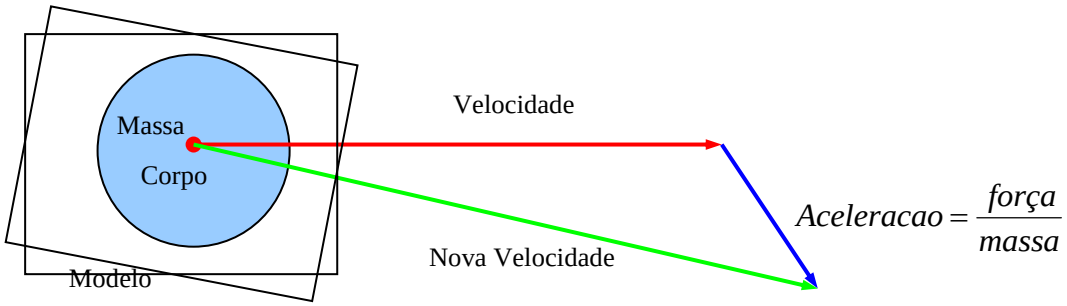


Figura 1: Modelo de veículo baseado em massa pontual [4].

```
class Actor
{
public:
   Vector2 pos,  //posicao
           dir,  //direcao
           lado, //vetor perperndicular a direcao
           vel;  //velocidade

   float   raio;
   double  massa;        //massa do actor
   float   timeElapsed;
   double  maxVel;       //velocidade maxima
   double  maxForca;     //forca maxima que o actor pode produzir

   Steering *steering;
   Vector2   steeringForce;

   Actor();

   void update(double time_elapsed, Vector2 targetPos, int steeringType);

   void render();
};
```

```
void Actor::update(double time_elapsed, Vector2 targetPos, int steeringType)
{
   timeElapsed = time_elapsed;

   steeringForce = steering->calculateSteering(this, targetPos, steeringType);

   Vector2 acceleration = steeringForce / massa;

   vel += (acceleration * time_elapsed);

   vel.truncate(maxVel);

   pos += vel * time_elapsed;

   if (vel.lenghtSqr() > 0.00000001)
   {
      Vector2 tmp(vel);
      tmp.normalize();
      dir  = tmp;
      lado = dir.perp();
   }
}
```

Classe Actor. Adaptado de [2].

Na chamada **`calculateSteering(this,targetPos,steeringType)`** calcula-se a força a ser aplicada sob o personagem para alterar sua nova posição. Passa-se como parâmetro o próprio personagem, a posição destino e o comportamento a ser aplicado. A escolha de comportamento é realizada pelo usuário, pelo método `keyboard()` da função `main()`.

```
void keyboard(unsigned char key, int x, int y)
{
   switch(key)
   {
      case 'a':
         steeringType = ARRIVE;
         break;

      case 's':
         steeringType = SEEK;
         break;

      case 'f':
         steeringType = FLEE;
         break;

      case 'w':
         steeringType = WANDER;
         break;
   }
}
```

O método render é usado para desenhar o personagem, bem como seus vetores direção, força e atributos do comportamento desempenhado.

```
void Actor::render()
{
   //renderiza o steering behavior
   steering->render(this);

   glPushMatrix();
   glLoadIdentity();

   //desenha o vetor direcao
   glColor3d(0, 1, 0);
   glTranslatef(pos.x, pos.y, 0);
   glBegin(GL_LINES);
      glVertex2d(0, 0);
      glVertex2d(dir.x*30, dir.y*30);
   glEnd();

   //desenha o vetor forca
   glColor3f(0,0,1);
   glBegin(GL_LINES);
      glVertex2d(0, 0);
      glVertex2d(steeringForce.x, steeringForce.y);
   glEnd();

   //calcula o angulo do personagem
   Vector2 vet(0,1);
   float ang = dir.angle(vet);
   ang *= (180/3.14);

   //desenha o actor
   glColor3d(0, 1, 0);
   glRotatef(-ang, 0, 0, 1);
   glBegin(GL_TRIANGLES);
      glVertex2d(-5, -5);
      glVertex2d(5, -5);
      glVertex2d(0, 15);
   glEnd();

   glPopMatrix();
}
```

3. Comportamentos de Direcionamento

O comportamento de um agente engloba as possíveis ações que este agente pode desempenhar. Essas ações são resultado da combinação de um elenco de regras que foram associadas ao agente. A especificação destas regras é resultado da avaliação das principais características do sistema onde o agente está inserido e, conseqüentemente, do que é esperado que o agente seja capaz de realizar. Para um melhor entendimento do comportamento de um agente, Reynolds [3] sugere quebrá-lo em várias camadas, como apresentado na Figura 2, que retrata o comportamento de movimentação em uma hierarquia de 3 camadas.

Pela análise da Figura 2, no topo da hierarquia está a seleção da ação, ou escolha dos objetivos e definição de estratégia para atingi-los. A camada de direcionamento (*steering*) é responsável pelo cálculo da trajetória requerida para satisfazer os objetivos e planos definidos pela camada de seleção de ação. Nesta camada são produzidas forças que descrevem para onde o agente deve ir (movimentação), desviando de possíveis obstáculos, e qual deve ser a velocidade de sua viagem. Por último, tem-se a locomoção, que descreve como o agente deve se locomover entre o ponto de origem e o ponto de destino. Pode envolver articulação de pernas ou movimentação de rodas. Para o caso de jogos, define como deve ser a animação gráfica da ação que está sendo realizada, e está intimamente relacionada com o tipo de personagem.

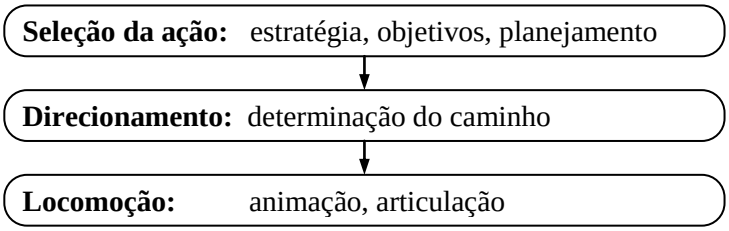


Figura 2: Comportamento de manobras dividido em 3 camadas.

3.1. Seek

O comportamento *seek* faz o personagem se locomover em direção a uma posição fixa no espaço global (Ver comportamento *pursuit* para o caso do alvo ser móvel). A velocidade desejada é um vetor na direção da posição do vetor para o alvo. Se após chegar ao alvo o personagem continua andando, ele vai fazer a volta para ser aproximar novamente. Para evitar esse efeito, deve-se utilizar o comportamento *arrive*.

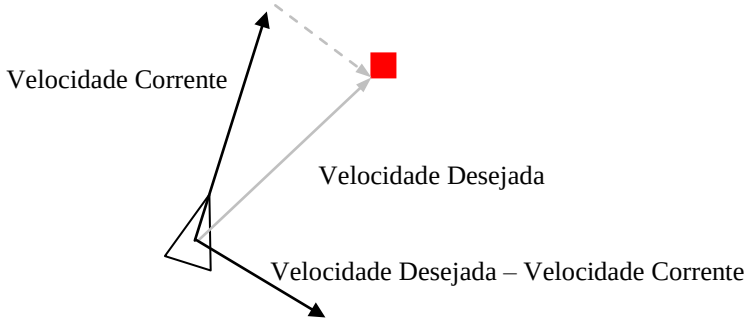


Figura 3: Seek Behavior.

```
Vector2 Steering::seek(Actor *actor, Vector2 targetPos)
{
   Vector2 desiredVelocity;
   desiredVelocity = targetPos - actor->pos;
   desiredVelocity.normalize();
   desiredVelocity *= actor->maxVel;

   return (desiredVelocity - actor->vel);
}
```

Algoritmo 4: Comportamento Seek [2].

3.2. Flee

Comportamento semelhante ao *seek*, exceto que a força resultante ocorre na direção oposta, como mostrado na Figura 4.

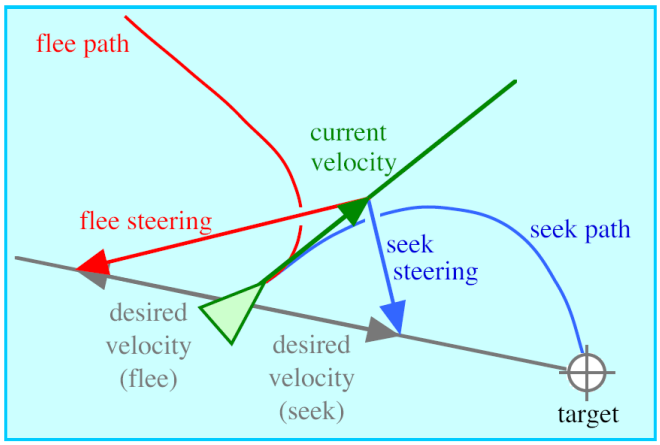


Figura 4: Comparação entre os comportamentos Seek e Flee [3].

```
Vector2 Steering::flee(Actor *actor, Vector2 targetPos)
{
   double panicDistance = 100;
   if( actor->pos.distance(targetPos) > panicDistance )
   {
     return Vector2(0,0);
   }
   Vector2 desiredVelocity;
   desiredVelocity = actor->pos - targetPos;
   desiredVelocity.normalize();
   desiredVelocity *= actor->maxVel;

   return (desiredVelocity - actor->vel);
}
```

Algoritmo 5: Comportamento Flee [2].

3.3. Arrive

O comportamento *arrive* é semelhante ao *seek*, com a diferença que o agente reduz a velocidade a medida que se aproxima do destino, e possivelmente parando exatamente sob o ponto de destino. O comportamento *arrive* pode ser aplicado somente quando o personagem se encontra a uma determinada distância do alvo.

```
Vector2 Steering::arrive(Actor *actor, Vector2 targetPos)
{
   Vector2 toTarget = targetPos - actor->pos;

   double dist = toTarget.lenghtSqr();
   float deceleration = 1;

   if(dist > 0)
   {
      double speed =  dist / deceleration;

      speed = speed < actor->maxVel ? speed : actor->maxVel;

      Vector2 desiredVelocity = toTarget * speed / dist;

      return (desiredVelocity - actor->vel);
   }
   return Vector2(0,0);
}
```

Algoritmo 6: Comportamento Arrive [2].

3.4. Pursuit

Pursuit é semelhante ao comportamento *seek* exceto pelo fato que o alvo é outra entidade móvel. Este comportamento requer a predição da posição futura do alvo. Essa posição é reavaliada a cada passo do algoritmo, desde que o personagem sendo perseguido não esteja se locomovendo na direção do perseguidor (ângulo inferior a 20 graus). A predição assume que o alvo não irá alterar a direção de deslocamento.

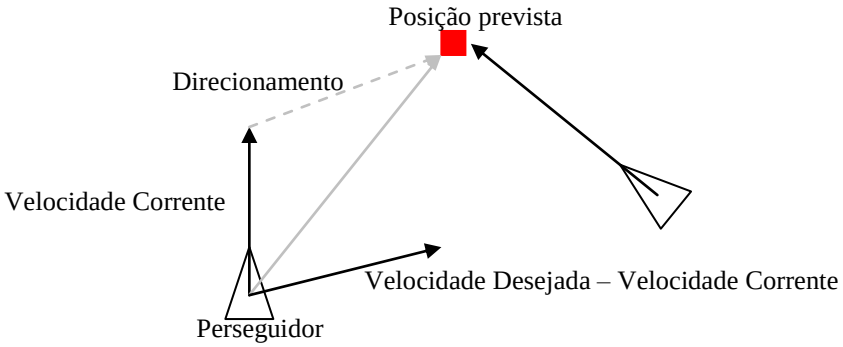


Figura 5: Pursuit Behavior

```
Vector2D Steering::Pursuit(Actor *actor evader, Actor *pursuer)
{
   //if the evader is ahead and facing the agent then we can just seek
   //for the evader's current position.
   Vector2D ToEvader = evader->pos - pursuer->pos;

   double relativeDir = pursuer->dir.Dot(evader->dir);

   if ( (ToEvader.Dot(pursuer->dir) > 0) &&
        (relativeDir < -0.95))  //acos(0.95)=18 degs
   {
      return seek(evader->pos);
   }

   double LookAheadTime = ToEvader.Length() /
                          (pursuer ->maxSpeed + evader->speed);

   //now seek to the predicted future position of the evader
   return Seek(evader->pos + evader->vel * LookAheadTime);
}
```

Algoritmo 5: Comportamento Pursuit [2].

3.5. Wander

Produz um comportamento que faz o personagem andar em um caminho aleatório. Para garantir que o caminho se pareça real, ao invés de simplesmente gerar uma força aleatória a cada passo, projeta-se a força em um cilindro, posicionado a frente do personagem. Deste modo, tem-se uma certa continuidade nas forças aplicadas a cada passo do algoritmo, fazendo com que o personagem consiga percorrer certas distâncias por caminhos persistentes. A cada passo do algoritmo, pequenos deslocamentos são adicionados ao alvo, o que faz com que o alvo se mova para frente e para traz do círculo. Dependendo do tamanho do círculo, distância do personagem ao círculo e da variação a cada passo, diferentes comportamentos podem ser obtidos.

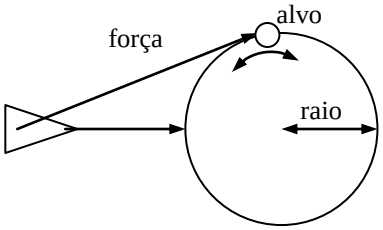


Figura 7: Estratégia de movimentação do comportamento Wander [1].

```
Vector2 Steering::wander(Actor *actor)
{
   double JitterThisTimeSlice = wanderJitter * actor->timeElapsed * 9;

   Vector2 tmp(randomClamped() * JitterThisTimeSlice,
               randomClamped() * JitterThisTimeSlice);

   wanderTarget += tmp;

   wanderTarget.normalize();

   wanderTarget *= wanderRadius;

   //Pozzer
   //projecao do circulo na frente do actor
   Vector2 target(actor->pos);
   target += (actor->dir*wanderDistance);
   target += wanderTarget;

   return target - actor->pos;
}
```

Algoritmo 6: Comportamento Wander [2].

3.6. Path following

Este comportamento gera forças para mover o Actor sob uma seqüência de *waypoints* que determinam um caminho a ser seguido. Este caminho pode ser resultado do algoritmo A* ou Dijkstra. Se o caminho for circular, o Actor fica continuamente percorrendo o caminho estipulado. Caso o personagem não esteja inicialmente sob um *waypoint* do caminho, ele deve se dirigir a um deles para iniciar o percurso. Caso o caminho não for circular, o personagem deve ir inicialmente para o primeiro *waypoint* do caminho.

```
Vector2 Steering::FollowPath()
{
   //move to next target if close enough to current target (working in
   //distance squared space)
   if(Vec2DDistanceSq(m_pPath->CurrentWaypoint(), m_pVehicle->Pos()) <
     m_dWaypointSeekDistSq)
   {
      m_pPath->SetNextWaypoint();
   }

   if (!m_pPath->Finished())
   {
      return Seek(m_pPath->CurrentWaypoint());
   }
   else
   {
      return Arrive(m_pPath->CurrentWaypoint(), normal);
   }
}
```

Algoritmo 7: Comportamento Path Folow [2].

3.7. Wall avoidance

O tratamento de colisão com muros gera forças que são perpendiculares ao muro, que visam afastar o personagem sempre que este estiver muito próximo. A determinação de proximidade é feita por meio de antenas, como mostrado na Figura 8. Cada personagem possui 3 antenas. Quando qualquer uma delas tocar ou atravessar o muro, calcula-se uma força de repulsão proporcional ao nível de penetração da antena dentro do muro.

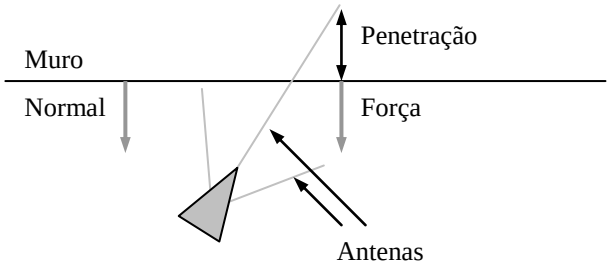


Figura 8: Colisão com muros

No Algoritmo 8 é apresentada a solução proposta por [2], e no algoritmo 9, uma solução mais simplificado que trata um único muro, porém com duas normais. No algoritmo 9, utiliza-se um teste para saber a posição relativa do personagem em relação ao muro, para então poder determinar em que direção a força de repulsão devem ser aplicada.

```
Vector2D SteeringBehavior::WallAvoidance(const std::vector<Wall2D>& walls)
{
  //the feelers are contained in a std::vector, m_Feelers
  CreateFeelers();

  double DistToThisIP    = 0.0;
  double DistToClosestIP = MaxDouble;

  //this will hold an index into the vector of walls
  int ClosestWall = -1;

  Vector2D SteeringForce,
           point,         //used for storing temporary info
           ClosestPoint;  //holds the closest intersection point

  //examine each feeler in turn
  for (unsigned int flr=0; flr<m_Feelers.size(); ++flr)
  {
    //run through each wall checking for any intersection points
    for (unsigned int w=0; w<walls.size(); ++w)
    {
      if (LineIntersection2D(m_pVehicle->Pos(),
                             m_Feelers[flr],
                             walls[w].From(),
                             walls[w].To(),
                             DistToThisIP,
                             point))
```

```
      {
        //is this the closest found so far? If so keep a record
        if (DistToThisIP < DistToClosestIP)
        {
          DistToClosestIP = DistToThisIP;

          ClosestWall = w;

          ClosestPoint = point;
        }
      }
    }//next wall


    //if an intersection point has been detected, calculate a force
    //that will direct the agent away
    if (ClosestWall >=0)
    {
      //calculate by what distance the projected position of the agent
      //will overshoot the wall
      Vector2D OverShoot = m_Feelers[flr] - ClosestPoint;

      //create a force in the direction of the wall normal, with a
```

```
      //magnitude of the overshoot
      SteeringForce = walls[ClosestWall].Normal() * OverShoot.Length();
    }
  }//next feeler

  return SteeringForce;
}
```

Algoritmo 8: Colisão com muros [2]

```
Vector2 Steering::wallAvoidance(Actor *actor)
{
   createFeelers(actor);

   double distance = 0.0;

   Vector2 SteeringForce,
           point;         //used for storing temporary info

   //examine each feeler in turn
   for (int i=0; i<3; i++)
   {
      if (LineIntersection2D(actor->pos, antenas[i], wall[0], wall[1],
          distance, point))
      {
         Vector2 OverShoot = antenas[i] - point;

         SteeringForce = wall[2] * OverShoot.lenghtSqr();

         //Pozzer
         Vector2 v1(wall[0] - wall[1]);
         Vector2 v2(wall[0] - actor->pos);
         int side = v1.side(v2);

         //printf("\nColidiu lado %d antena %d ", side, i);
         if( side == -1 )
            return SteeringForce;
         else
            return SteeringForce.reverse();
      }
  }//next feeler

  return SteeringForce;
}
```

Algoritmo 9: Colisão com um muro. Adaptado de [2]

Deve-se observar que nos dois algoritmos, não são tratadas colisões com o personagem andando de ré. É dado como exercício a solução para este problema. Outras modificações também podem ser feitas neste algoritmo, como uma função não linear de repulsão, onde a reação cresce não linearmente com a taxa de penetração do muro. Para alguns tipos de jogos, pode ser interessante que a colisão não seja tratada com um amortecimento. Quando uma antena tocar no muro, o personagem pode ser parado imediatamente. Cabe também como exercício a adaptação desta abordagem.

3.8. Offset Pursuit

Este comportamento é utilizado em formações, onde atores assumem posições relativas em relação a um líder. O algoritmo 10 deve ser chamado para cada actor que deve assumir uma posição relativa a um líder.

```
Vector2D SteeringBehavior::OffsetPursuit(const Vehicle* leader,
                                         const Vector2D offset)
{
   //calculate the offset's position in world space
   Vector2D WorldOffsetPos = PointToWorldSpace(offset,
                                               leader->dir,
```

```
                                                leader->lado,
                                                leader->pos);

    Vector2D ToOffset = WorldOffsetPos - actor->pos;

    //the lookahead time is propotional to the distance between the leader
    //and the pursuer; and is inversely proportional to the sum of both
    //agent's velocities
    double LookAheadTime = ToOffset.Length() /
                           (actor->maxVel + leader->vel);

    //now Arrive at the predicted future position of the offset
    return Arrive(WorldOffsetPos + leader->vel * LookAheadTime, fast);
}
```

Algoritmo 10: Comportamento Offset Pursuit [2].

4. Comportamentos de Grupos

Nas Figuras 9 e 10 são apresentados exemplos práticos do uso de comportamentos de manobra em grupos. Deve-se observar que a modelagem do comportamento individual de cada indivíduo neste tipo de aplicação é inviável, visto que exigiria esforços de tempo e dinheiro enormes.

Figura 9: Walt Disney Pictures

Figura 10: Eletronic Arts

Comportamentos de grupos levam em consideração alguns ou todos os personagens do ambiente. Geralmente utilizam-se áreas de influência para determinar quais atores devem pertencer ao grupo de um ator, que possui como centro a posição do ator. Pode-se utilizar um círculo para selecionar os elementos que estão dentro da vizinhança, centrado no líder, como mostrado na Figura 11.

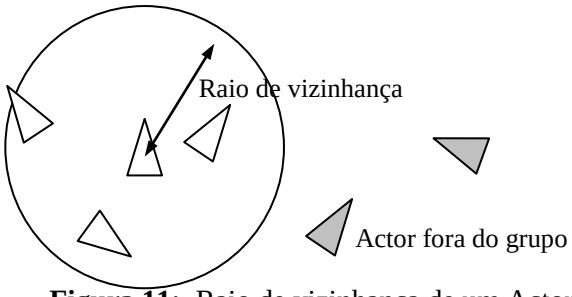


Figura 11: Raio de vizinhança de um Actor

Para calcular a vizinhança de um personagem, utiliza-se o método `TagNeighbors()`, como mostrado no Algoritmo 11. Deve-se observar que este método é chamado para cada personagem, assim que seu método `update()` é executado. Em termos cronológicos, inicialmente o cenário é atualizado. Isso faz com que cada personagem seja atualizado, o que causa a definição da vizinhança para então definir os comportamentos em grupo a serem aplicados a cada personagem. Para tornar a representação mais dinâmica, caso um personagem não perteça a nenhum grupo, ele assume o comportamento *wander* até encontrar um personagem em sua vizinhança.

```
template <class T, class conT>
void TagNeighbors(const T& entity, conT& ContainerOfEntities, double radius)
{
  //iterate through all entities checking for range
  for (typename conT::iterator curEntity = ContainerOfEntities.begin();
       curEntity != ContainerOfEntities.end();
       ++curEntity)
  {
    //first clear any current tag
    (*curEntity)->UnTag();

    Vector2D to = (*curEntity)->Pos() - entity->Pos();

    double range = radius + (*curEntity)->BRadius();

    //if entity within range, tag for further consideration. (working in
    //distance-squared space to avoid sqrts)
    if ( ((*curEntity) != entity) && (to.LengthSq() < range*range))
    {
      (*curEntity)->Tag();
    }

  }//next entity
}
```

Algoritmo 11: Função para determinar os vizinhos de um personagem [2].

Uma aplicação muito comum deste tipo de comportamento ocorre em formações (*flocking*), onde grupos de indivíduos comportam-se segundo regras, como ocorre na natureza com bando de pássaros ou cardumes de peixes.

Formações fazem uso de 3 tipos de comportamentos em grupo: coesão (*cohesion*), separação (*separation*) e alinhamento (*alignment*), todas operando juntas. Na Figura 12 são apresentados exemplos detes 3 comportamentos.

4.1. Separation

Cria forças que direcionam o personagem para fora da região da vizinhança, fazendo com que os personagens se afastem uns dos outros.

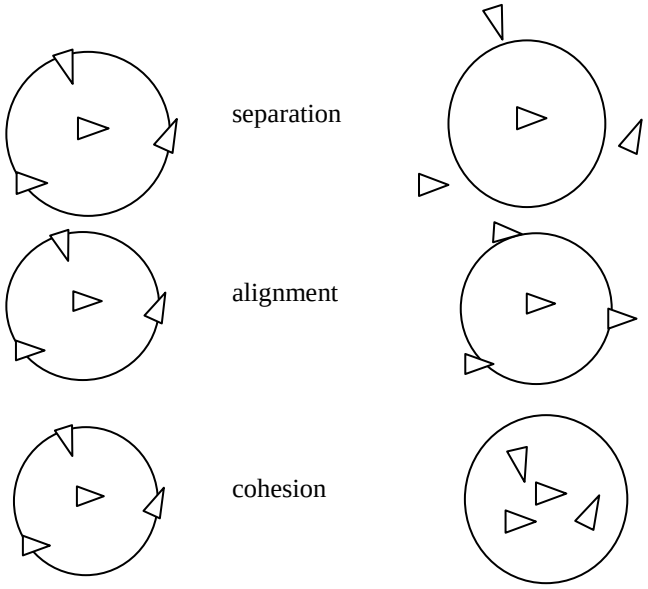


Figura 12: Comportamentos em grupo

```
Vector2D SteeringBehavior::Separation(const vector<Vehicle*> &neighbors)
{
  Vector2D SteeringForce;

  for (unsigned int a=0; a<neighbors.size(); ++a)
  {
    if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged() &&
      (neighbors[a] != m_pTargetAgent1))
    {
      Vector2D ToAgent = m_pVehicle->Pos() - neighbors[a]->Pos();
      SteeringForce += Vec2DNormalize(ToAgent)/ToAgent.Length();
    }
  }

  return SteeringForce;
}
```

Algoritmo 12: Comportamento Separation [2].

4.2. Alignment

Faz com que o vetor direção do personagem aponte na mesma direção dos personagens do grupo.

```
Vector2D SteeringBehavior::Alignment(const vector<Vehicle*>& neighbors)
{
  //used to record the average heading of the neighbors
  Vector2D AverageHeading;

  //used to count the number of vehicles in the neighborhood
  int    NeighborCount = 0;

  //iterate through all the tagged vehicles and sum their heading vectors
  for (unsigned int a=0; a<neighbors.size(); ++a)
  {
    if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged() &&
      (neighbors[a] != m_pTargetAgent1))
    {
      AverageHeading += neighbors[a]->Heading();

      ++NeighborCount;
    }
  }

  //if the neighborhood contained one or more vehicles, average their
```

```
  //heading vectors.
  if (NeighborCount > 0)
  {
    AverageHeading /= (double)NeighborCount;

    AverageHeading -= m_pVehicle->Heading();
  }

  return AverageHeading;
}
```

Algoritmo 13: Comportamento Alignment [2].

4.3. Cohesion

Cria forças para mover o personagem para o centro de massa do grupo.

```
Vector2D SteeringBehavior::Cohesion(const vector<Vehicle*> &neighbors)
{
  //first find the center of mass of all the agents
  Vector2D CenterOfMass, SteeringForce;

  int NeighborCount = 0;

  //iterate through the neighbors and sum up all the position vectors
  for (unsigned int a=0; a<neighbors.size(); ++a)
  {
    if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged() &&
      (neighbors[a] != m_pTargetAgent1))
    {
      CenterOfMass += neighbors[a]->Pos();

      ++NeighborCount;
    }
  }

  if (NeighborCount > 0)
  {
    CenterOfMass /= (double)NeighborCount;
    SteeringForce = Seek(CenterOfMass);
  }

  return Vec2DNormalize(SteeringForce);
}
```

Algoritmo 14: Comportamento Cohesion [2].

4.4. Flocking

Flockings sao agrupamentos de indivíduos de mesma espécie (peixes ou aves, por exemplo) que se locomovem juntos em uma mesma direção. A técnica de flocking produz resultados conhecidos como comportamento emergente, que é caracaterizado por uma representação que se parece complexa, mas que ao mesmo tempo é derivada da execução de regras simples combinadas. Cada entidade individual do grupo não faz idéia de que tipo de comportamento global está sendo gerado.

Flocking é resultado da combinação simultânea dos três comportamentos em grupo: separação, coesão e alinhamento. Dependendo da contribuição de cada comportamento, diferentes efeitos são obtidos.

4.5. Combinação de Comportamentos em Grupo

Geralmente utilizam-se a combinação de vários comportamentos para se conseguir o comportamento desejado. Dificilmente é utilizado um único comportamento. Por exemplo, em um jogo pode-se designar o

comportamento **wander** para que um personagem procure por inimigos no cenário. Porém, este mesmo personagem não deve atravessar paredes. Por isso, deve-se adicionar o comportamento **wall avoidance**, bem como o comportamento **separation** para evitar que personagens do mesmo time andem sempre juntos.

Para os demos de [2], define-se um personagem com comportamento Wander e os demais com comportamento Flocking, como mostrado no seguinte algoritmo:

```
for (int a=0; a<Prm.NumAgents; ++a)
{
    Vehicle* pVehicle = new Vehicle(...);
    pVehicle->Steering()->FlockingOn();
}

//o personagem com id = NumAgents tem comportamento wander.
m_Vehicles[Prm.NumAgents-1]->Steering()->FlockingOff();
m_Vehicles[Prm.NumAgents-1]->SetScale(Vector2D(10, 10));
m_Vehicles[Prm.NumAgents-1]->Steering()->WanderOn();
m_Vehicles[Prm.NumAgents-1]->SetMaxSpeed(70);


//todos devem fugir do personagem com id = NumAgents
for (int i=0; i<Prm.NumAgents-1; ++i)
{
    m_Vehicles[i]->Steering()->EvadeOn(m_Vehicles[Prm.NumAgents-1]);
}
```

Algoritmo 15: Associação de comportamentos aos personagens [2].

Quando cada personagem é processado, utiliza-se o método `calculate()` para determinar a força resultante que aplicam sobre o personagem. Deve-se lembrar que a força resultante não pode ser superior a uma força máxima designada a cada personagem. Se a força resultante for maior, esta deve ser truncada de forma que a magnitude da força não ultrapasse o limite.

Existem várias formas de fazer este cálculo. O mais direto consiste em somar todas as forças aplicadas e após truncar pela força máxima aplicável.

```
Vector2 Calculate()
{
   Vector2 steeringForce;
   steeringForce += wander() * wanderAmount;
   steeringForce += wallAvoidance() * wallAmount;
   steeringForce += Separation() * separationAmount;

   return steeringForce.truncate(MAX_FORCE);
}
```

Algoritmo 16: Abordagem simples determinação da força resultante [2].

Este algoritmo apresenta um problema grave no que se refere a influência de cada força na força resultante, pois não prioriza forças de maior importância. Neste exemplo, a força de repulsão contra muros deve ser de maior importância, pois senão um personagem, cercado por vários outros, pode acabar sendo empurrado para dentro do muro devido a força de separação.

Uma solução para este problema é o uso de uma estratégia de priorização, que visa resolver o problema que ocorre quanto existem forças contrárias resultantes de diferentes comportamentos, como citado no exemplo anterior.

Nesta abordagem, os comportamentos são avaliados na ordem de importância. Assim, cada força é calculada separadamente e então somada a força resultante, pelo método `accumulateForce()`. Antes de somar a força à força resultante, testes são realizados e 3 situações podem ocorrer:

- A nova força pode ser adicionada a resultante se a soma não ultrapassar o limite máximo de força
- Se não existe mais sobra de força resultante, a nova força e todas as forças por ainda serem calculadas são descartadas, visto que são de menor importância;

- Se ainda existir um pouco de força resultante, a força em questão é truncada para o valor que ainda pode ser adicionado a força resultante.

Este algoritmo apresenta várias vantagens, como a priorização de forcas mais importante, descarte de forças que não devem ser consideradas e otimização do processamento, pois vários comportamentos não precisam ser avaliados caso a força máxima já foi atingida.

```
Vector2D SteeringBehaviors::Calculate()
{
   //reset the force.
   SteeringForce.Zero();

   SVector2D force;
   if (On(wall_avoidance))
   {
     force = WallAvoidance(Vehicle->World()->Walls()) *  MultWallAvoidance;
     if (!AccumulateForce(SteeringForce, force)) return SteeringForce;
   }

   if (On(separation))
   {
     force = Separation(Vehicle->World()->Agents()) * MultSeparation;
     if (!AccumulateForce(SteeringForce, force)) return SteeringForce;
   }
   return SteeringForce;
}
```

Algoritmo 17: Método para calcular a força resultante de vários comportamentos [2].

```
bool SteeringBehavior::AccumulateForce(Vector2D &RunningTot,
                                       Vector2D ForceToAdd)
{
   double MagnitudeSoFar = RunningTot.Length();
   double MagnitudeRemaining = m_pVehicle->MaxForce() - MagnitudeSoFar;

   if (MagnitudeRemaining <= 0.0) return false;

   double MagnitudeToAdd = ForceToAdd.Length();

   if (MagnitudeToAdd < MagnitudeRemaining)
   {
      RunningTot += ForceToAdd;
   }
   else
   {
      RunningTot += (Vec2DNormalize(ForceToAdd) * MagnitudeRemaining);
   }

   return true;
}
```

Algoritmo 18: Método avaliar se nova força pode ser acumulada [2].

Referências

[1] Matt Buckland. **Programming Game AI by Example**. Wordware publishing Inc, 2005 (*Referência usada na maior parte deste material*)

[2] Matt Buckland. **Programming Game AI by Example**. Resource Page. Disponível em: http://www.wordware.com/files/ai/

[3] Craig W. Reynolds. **Steering behaviors for autonomous characters,** Game Developers Conference, 1999. http://www.red3d.com/cwr/

[4] Craig W. Reynolds. **Steering behaviors for autonomous characters,** Game Developers Conference, 2001