AI for games

Lecture 2

Behaviours

Organiser

- Steering and physics
- Behaviours
  - Seek
  - Arrive
  - Pursue
  - Evade
  - Wander
  - Obstacle avoid
  - Wall avoid
  - Interpose
  - Hide

Organiser

- Combining behaviours
- Group behaviours
  - Flocking
  - Simulated teamwork

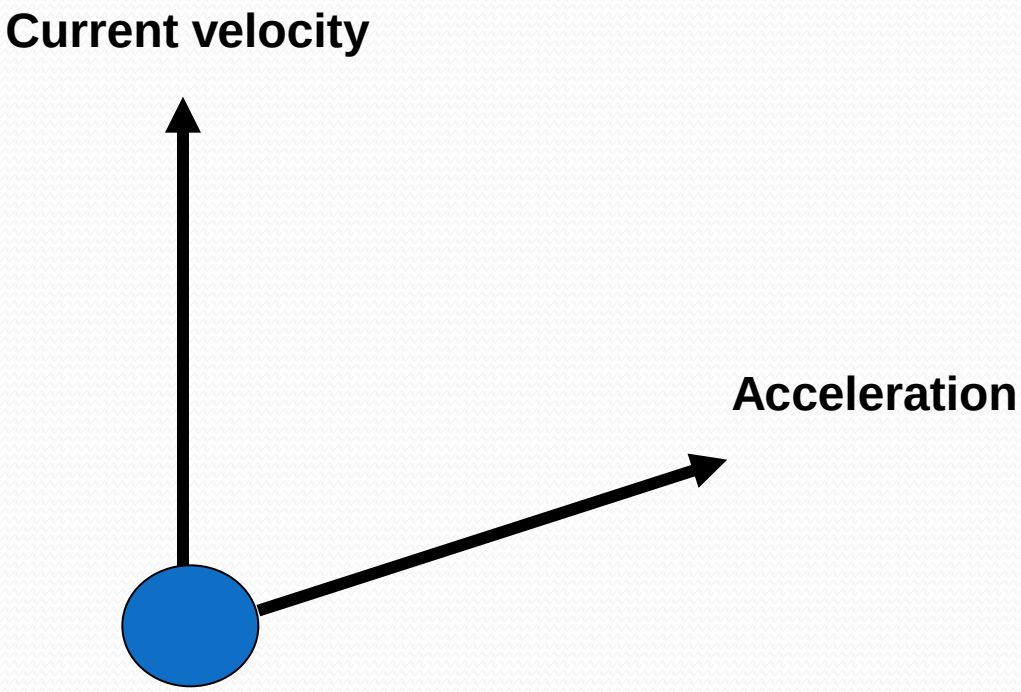
Craig Reynolds

- Much of this work is based on Craig Reynold's work.
  - Famous for flocking, but did a lot more.

Steering

- A behaviour produces a force (or an acceleration) in the direction the behaviour dictates.
- This will steer the entity in the direction requested eventually.

Steering



Steering

- Surely this will never actually reach the requested direction.
- It will, as long as:
  - Friction is applied.
- OR
  - Velocity is truncated to max speed.

Which behaviour?

- A higher level AI (state machine/expert system) will decide which behaviour to use at the moment. (Flee? Intercept? Hide?)
- In fact, will use several at once. (Intercept, but try to keep out of sight. Oh, and don’t bump into the walls.)

Physics

```
Vector2D acceleration;
if(bSeekOn)
  acceleration += Seek(target);
if(bHideOn)
  acceleration += Hide(target);
if(bAvoidWallsOn)
  acceleration += AvoidWalls();
```

- (Actually, this will be more complicated later.)

Physics

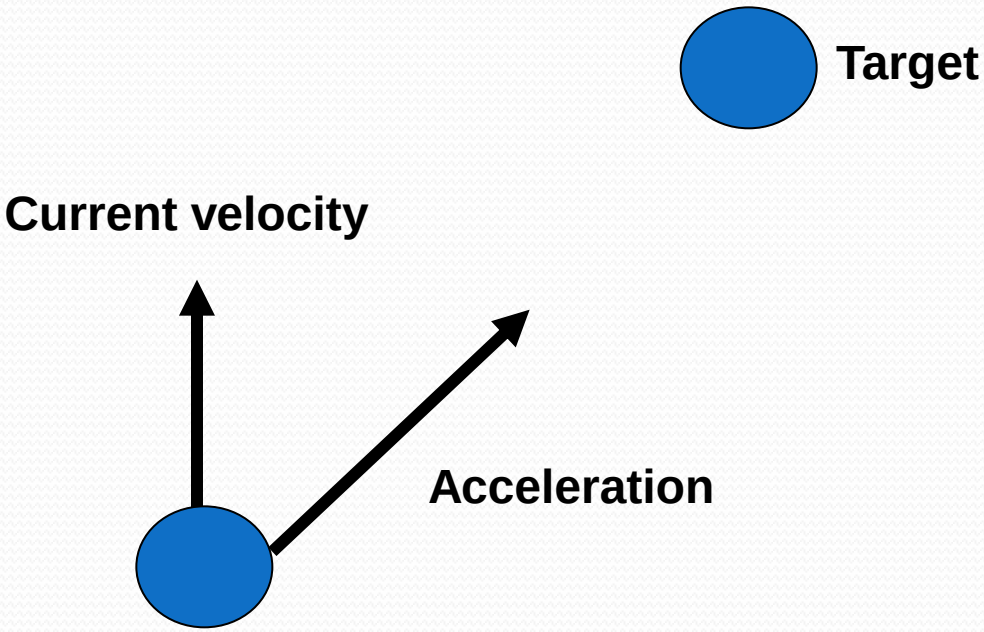
```
mVelocity += acceleration;

// Limit acceleration at this point?

if(velocity.magnitude()>kMaxSpeed)
{
  velocity = velocity.unitVector();
  velocity *= kMaxSpeed;
}
```

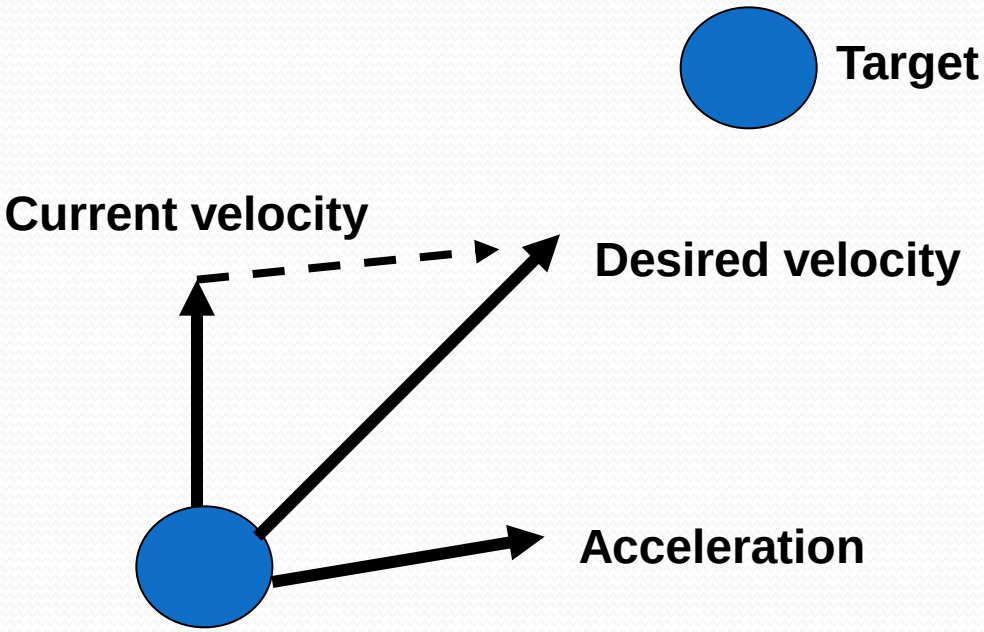
Seek

- Simple – accelerate towards target:



Seek

- Better – bring velocity in line with target asap.



Seek

- Acceleration = desired velocity – current velocity

```
vector2D Seek(Vector2D target)
{
  Vector2D desiredVelocity = (target –
    position).unitVector() * kMaxSpeed;
  Vector2D behaviourAccn =
  desiredVelocity-velocity;
  return behaviourAccn;
}
```

Flee

- Just the opposite direction.
- Otherwise the same as seek.
- Note that this runs directly away.
- Too easy a target?

Arrive

- More complex, as it slows down to a stop as it reaches the target.
- Rather than use kMaxSpeed, calculate speed as (distance / someConstant).
- If this is over kMaxSpeed, set to kMaxSpeed.

Pursue

- If you just head at the target, this is “pure pursuit”, and you will end up behind them.
- Need to lead the target a little.
- Calculating a perfect intercept for a non-cooperating target is not worth processor cycles.

Pursue

- Pick a point ahead of the target and aim for that.
- But how far ahead?
- Calculate a rough intercept time, and predict position of target at that time.
- Rough time is distance/maxspeed.
  - Can make this more fancy.

Pursue

```
Vector2D Pursue(targetPos, targetVelocity)
{
   double distance = (targetPos –
     position).magnitude();
   double time = distance/kMaxSpeed;
   Vector2D target = targetPos +
     targetVelocity*time;
   return Seek(target);
}
```

Pursue

- Can add an extra parameter to the function (set by the higher AI).
- This acts as a multiplier to the time.
- Greater than 1 gives you overlead pursuit.
- Less than 1 gives you underlead pursuit.
- Negative can give you a lag pursuit.
- (Good for getting on tail or blocking.)

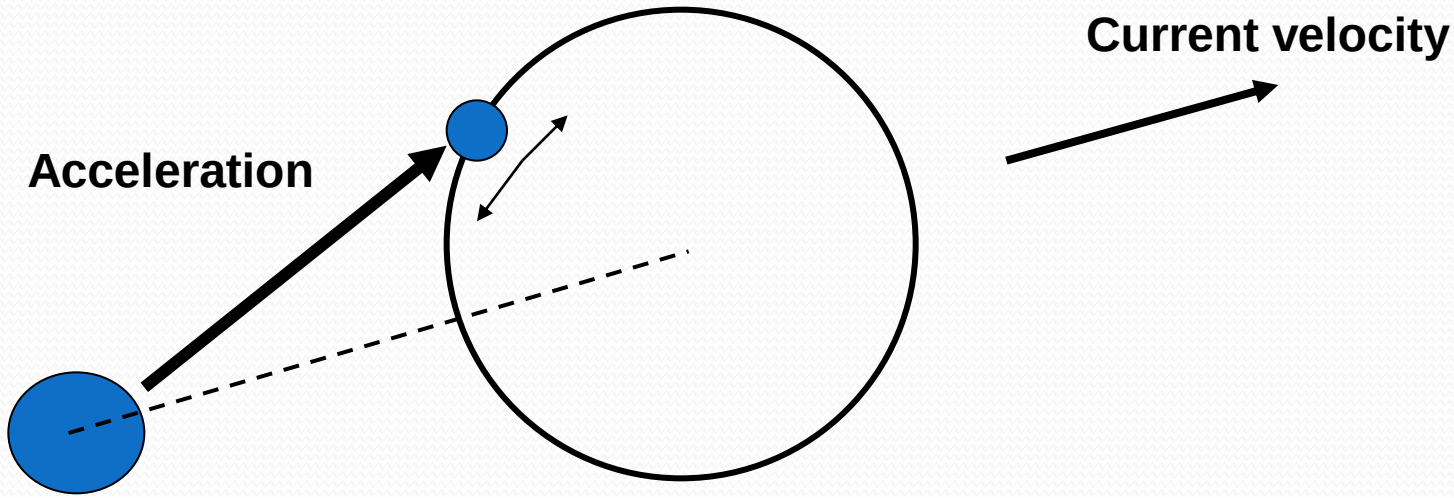
Evade

- Same as Pursue, but run away from anticipated point.
- All the same, but the final line uses Flee( ), not Seek( ).

Wander

- For situations where you want the character to wander at random.
- Not useful for BMB, but cute and handy for many games with ambient characters or non-combat situations.
- Just pick a random direction?
  - Why not?
- Question:
  - If you move in a random direction every time, do you move away from the starting point with time? *(Drunkard's Walk)*

Wander



Wander

1. Working in agent's local space
2. Add a random vector to the target position.
3. Clamp to the wander radius.
4. Project forward of agent by wander distance.
5. Project into world space.

(Alternative – use angles instead of vectors to avoid local space, but slower calculation.)

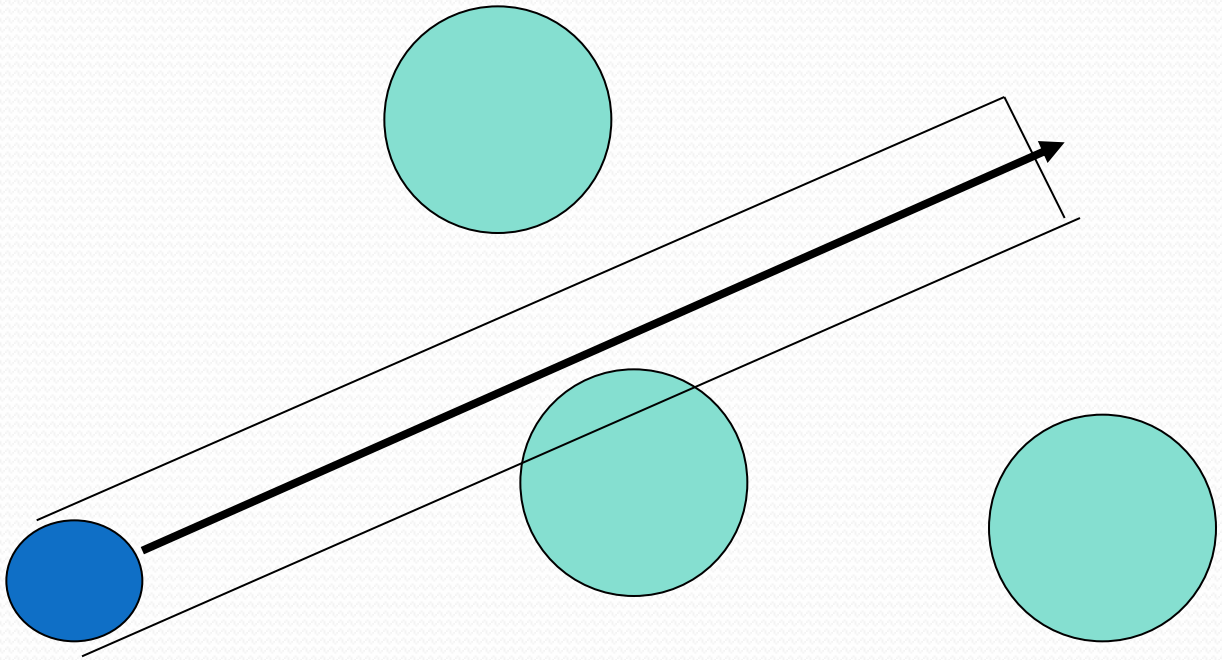
Wander

- Tinker with size of random addition, wander radius and wander distance to get effect desired.

Obstacle avoid

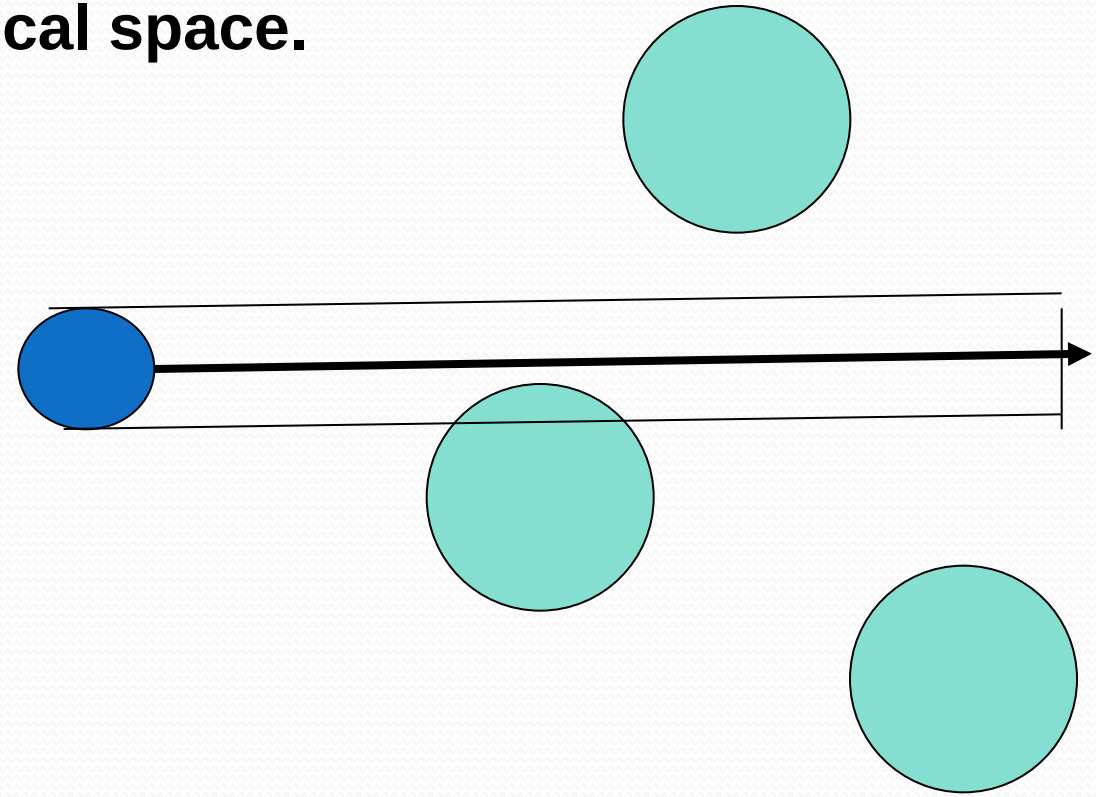
- This is for avoiding circular obstacles, not walls and so on.
- Obviously only has meaning when combined with another behaviour.

Obstacle avoid



Obstacle avoid

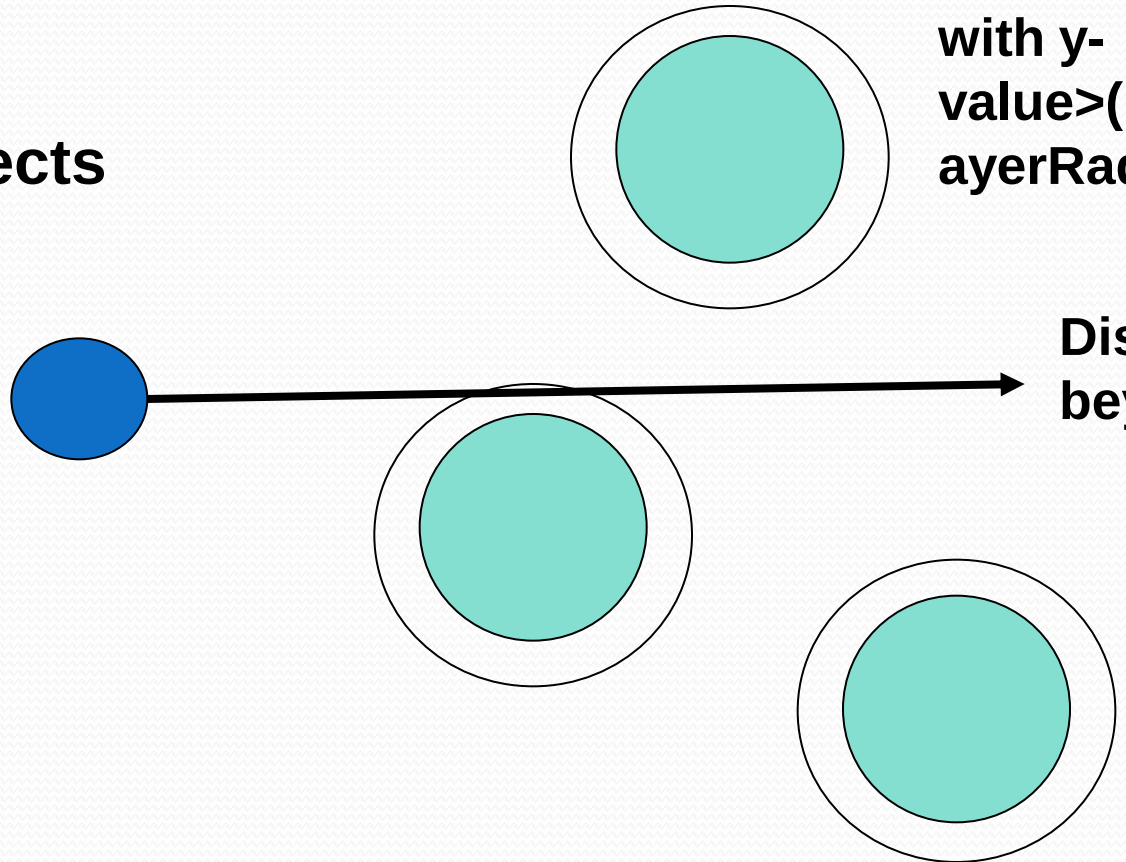
Transform all objects into local space.



Obstacle avoid

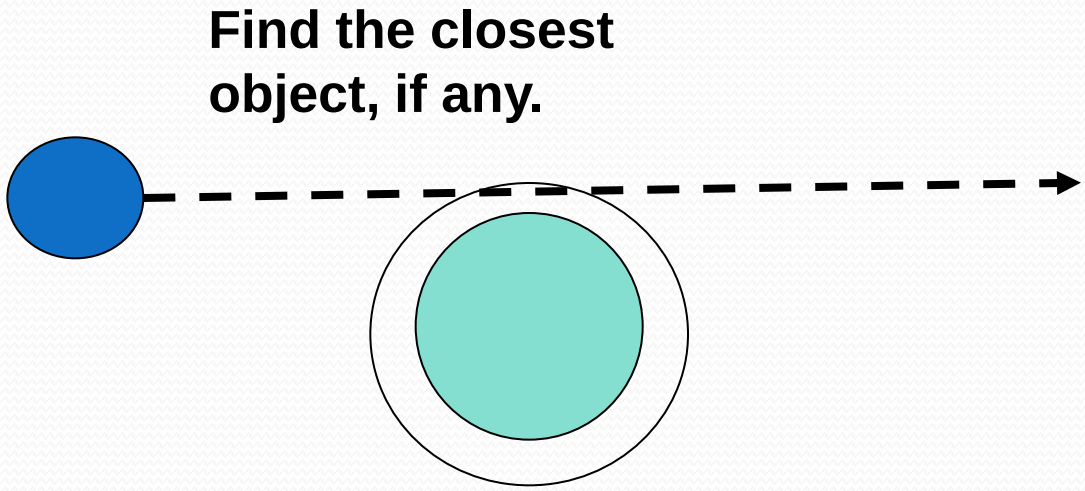
Discard objects with y-value>(radius+playerRadius).

Discard objects behind.



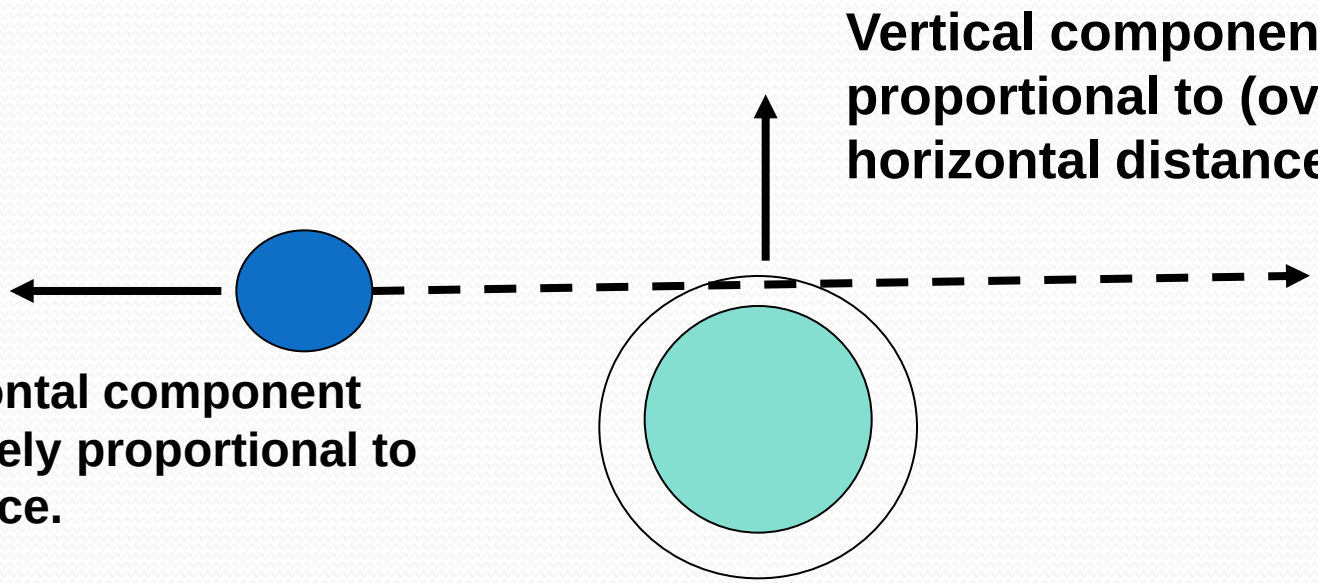
Discard objects beyond range.

Obstacle avoid

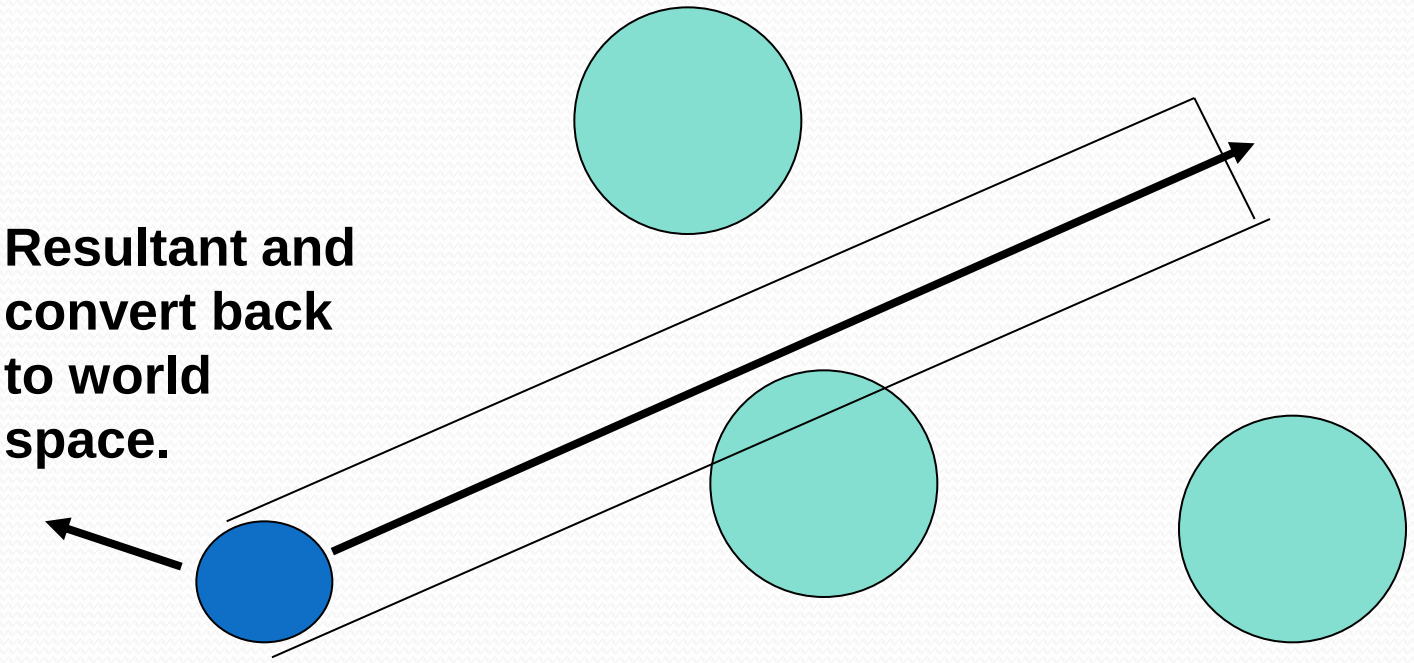


Obstacle avoid

Get two components of avoiding acceleration.

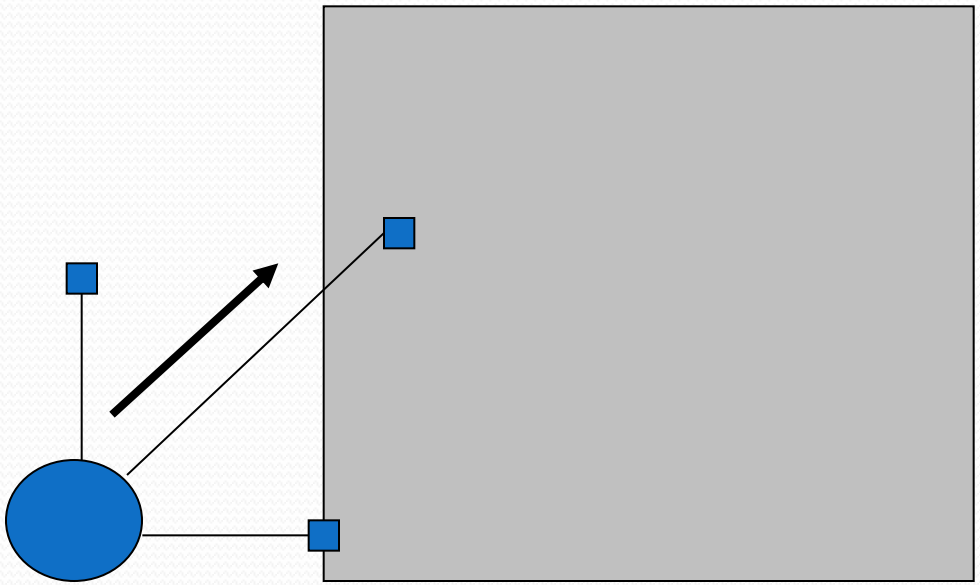


Obstacle avoid



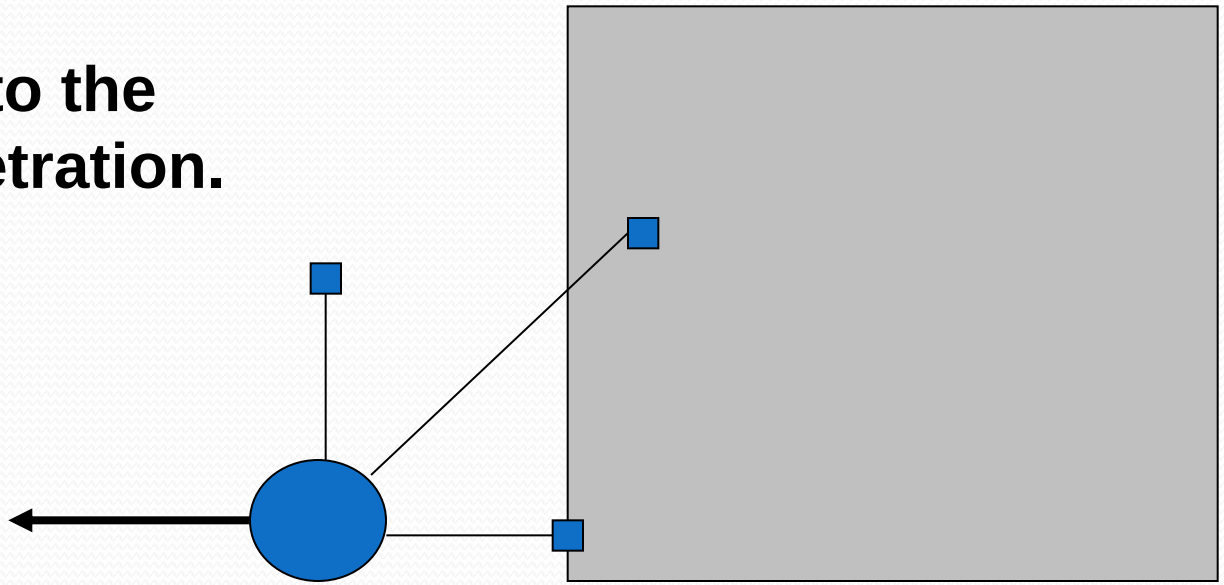
Wall avoid

Use three “feelers”. If any of them are inside a wall, take action.

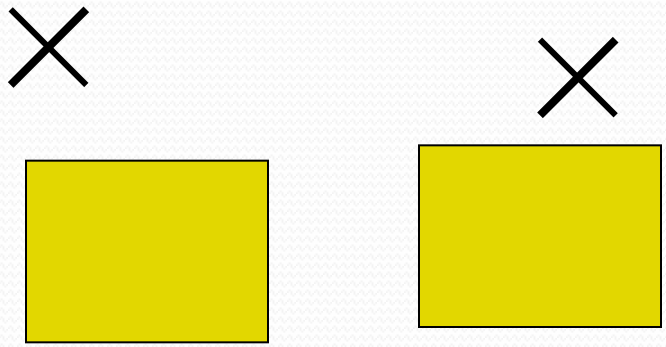


Wall avoid

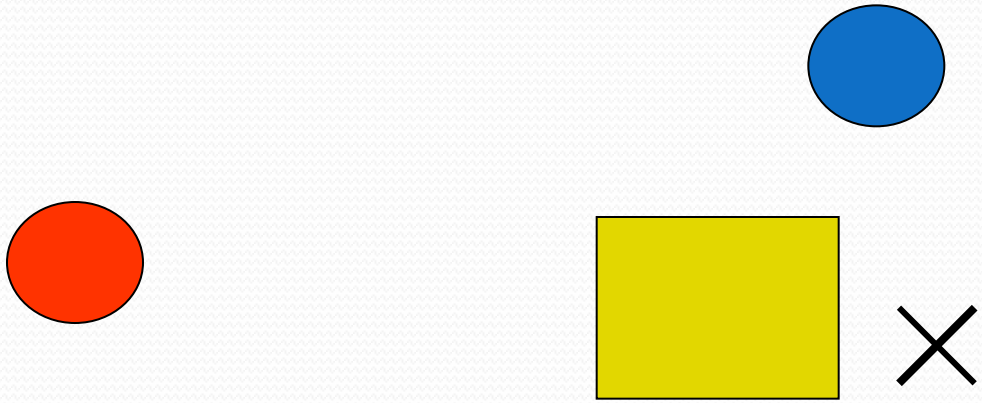
Acceleration is normal to wall, and proportional to the depth of penetration.



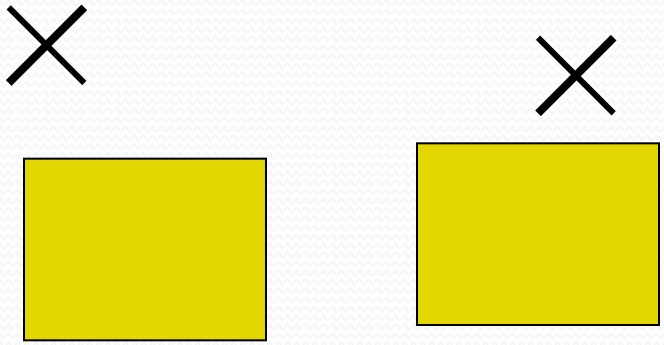
Hide



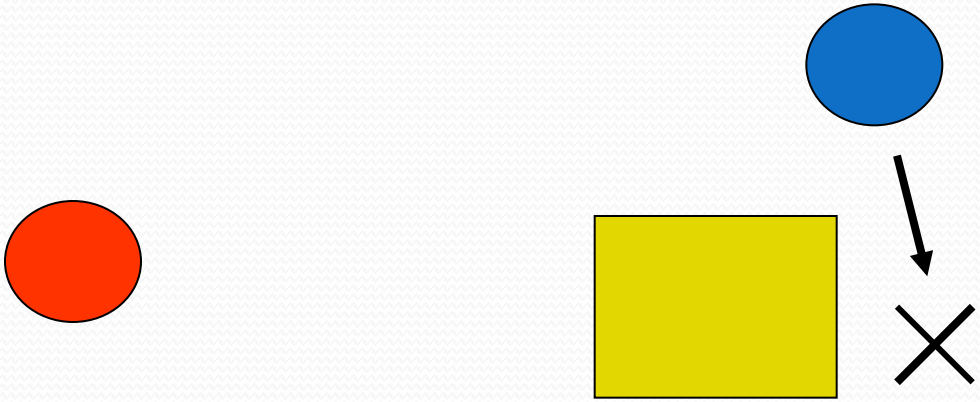
Generate list of points on opposite side of obstacle to target.



Hide



Seek( ) the closest.



Hide

- For large obstacles, you may need to generate several points.
- Also, you may need to weight the distance so that you may go to other nearby hiding points that are better in some way.
- (E.g. ones with higher tactical rating on your terrain analysis results.)

Combined behaviours

- If we have multiple behaviours, how do we combine them?
- First, the higher AI will turn some off.

```
bSeekOn = false;
bHideOn = true;
bAvoidWallsOn = true;
```

Etc.

- And may set other factors.

```
Vector2D target = opponents[xzy].position;
```

Combined behaviours

- But how to sum the rest?

Weighted sum

- Add them all up, with tinkerable weights?

```
Vector2D acceleration;
if(bSeekOn)
  acceleration += 0.745 * Seek(target);
if(bHideOn)
  acceleration += 1.2 * Hide(target);
if(bAvoidWallsOn)
  acceleration += 0.56 * AvoidWalls();
```

Weighted sum

- Can work, but costly, since all get evaluated each frame.
- Plus, opposing drives can cause agent to stick.
  - Often in an oscillating situation.
  - Asimov's "Catch the robot"

Prioritised weighted truncated running sum

- Calculate the (weighted) behaviours in order of importance, and keep the running total.
- But once the running total reached maximum acceleration, stop.
- This means that when something urgent is directing behaviour (avoid wall!!!!!), will act on it and will not bother calculate the rest.

Prioritised weighted truncated running sum

- Order of importance?
  - Avoid wall
  - Avoid obstacle
  - Hide
  - Evade
  - Pursue
  - Seek
- Or whatever.
- Can be altered by higher AI?

Prioritised dithering

- Very cheap for CPU.
- Pick one at random, but with a strong weight towards higher priorities.
- If it returns zero (e.g. no wall to avoid), run another one.
- Cheapest on random number generation to check each in turn, starting at highest priority, and not return to top if zero.

Summary

- Behaviour and physics
- Behaviours
- Combining behaviours
  - Weighted sum
  - Weighted truncated prioritised running sum
  - Prioritised dithering