18

Context Steering

Behavior-Driven Steering at the Macro Scale

Andrew Fray

18.1 Introduction
18.2 When Steering Behaviors Go Bad
18.3 Toward Why, Not How
18.4 Context Maps by Example
18.5 Racing with Context
18.6 Advanced Techniques
18.7 Conclusion
References

18.1 Introduction

Steering behaviors are extremely common in the games industry [Reynolds 87, Reynolds 99]. Their popularity is with good cause, promising a fast-to-implement core with emergent behavior from simple components.

However, steering behaviors are not suited for some types of game. When the player can pick out and watch individual entities, collision avoidance and consistent movement become very important. Achieving this can cause behavior components to balloon in size and become tightly coupled. Entity movement then becomes fragile and hard to tune.

In this chapter, we'll outline how you can identify the games for which steering behaviors aren't a good fit and describe a new approach for those problems called *context steering*. Context steering behaviors are small and stateless and guarantee any desired movement constraint. When used to replace steering behaviors on the game *F1 2011*, the codebase shrunk by 4000 lines, yet the AI were better at avoiding collisions, overtaking, and performing other interesting behaviors.

18.2 When Steering Behaviors Go Bad

A steering behavior system is used to move an entity through a world. The system consists of multiple child behaviors. During each update, the child behaviors are asked for a vector representing how they would like the entity to move. The vectors are combined to produce a final velocity. That's really it; the simplicity of the system is one of its strengths.

Note that the behavior vectors can be either a desired final velocity or a corrective force to the current velocity. This chapter will show behavior output as a final velocity. It doesn't change the arguments either way, but it makes the diagrams easier to arrange and understand.

Imagine an entity with free movement on a 2D plane. The entity cares about avoiding *obstacles* and chasing *targets*. At the instant of time shown in Figure 18.1, there are two possible targets in the scene and one obstacle.

What's the ideal result here? Assuming our only concern on choosing a target is distance, the entity should move toward target A. However, there's an obstacle in the way, so moving toward target B would be best. Can that decision emerge from small simple behaviors?

We start with two simple steering behaviors: *chase*, for approaching targets, and *avoid*, for not hitting obstacles. Our avoid behavior sees the nearby obstacle and returns a velocity to avoid it. The chase behavior knows nothing about obstacles and so returns a velocity toward the nearest target, target A.

The behavior system combines these behaviors. Let's assume they're just averaged for now, although you can use more complex combination techniques. The final vector is very close to 0, and the entity hardly moves. Players are not going to think this is an intelligent entity!

Steering behavior systems have evolved some Band-Aids to deal with this situation over the years. Here's a few ways this stalemate might be solved:

We could add *weighting* to our behaviors, so avoid heavily outweighs chase when there is an obstacle nearby. Now the entity has a strong northward velocity, but at some point, it will reach equilibrium again. We've only succeeded in moving the problem, at the cost of a new weighting parameter. That parameter will invariably need tweaking any time we change any of the affected behaviors.

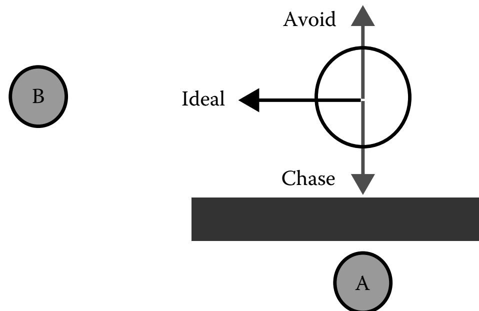

Figure 18.1

Entity layout, ideal path, and first path.

We could add *prioritization*, so avoid is the only behavior that runs when near obstacles, but then movement near obstacles is very single minded, and not very expressive.

Finally, we could add some *awareness* of obstacles to the chase behavior. It could raycast to reject targets that don't have a clear path or pathfind to all targets and select the one with the shortest path. Both of these introduce the concept of obstacles into chase, which increases coupling. In most game engines, raycasts and pathfinds are going to be either expensive or asynchronous, both of which introduce different types of complexity. This makes chase neither "small" nor "stateless."

There doesn't seem to be a good way to fix this.

This may sound like a forced example, but it's based on a real experience. In *F1 2010*, our equivalent of the avoid behavior had to be incredibly robust, which meant it often demanded to run frequently and in isolation, dominating how the opponent cars moved. To put some expressiveness back into the AI, we extended the avoid behavior over and over again to make it avoid in intelligent ways, coupling it to multiple other behaviors and making it monolithic. By the end, it had decomposed into an old-school sequence of if/else blocks with a thin steering behavior wrapper and was a maintenance nightmare.

18.2.1 Flocks versus Groups

If steering behaviors are so broken, why are they so popular? Because not all games expose the right conditions to make the problems apparent. Steering behaviors are a statistical steering method. Most of the time, they will give you mostly the right direction. How often they give you inconsistent or bad results, and how bad that is for the player, is a per-game decision.

It's no coincidence that the most famous application of steering behaviors is flocking [Reynolds 87]. In flocking, the user is typically an external observer of many entities moving as a semicohesive group. The group seems to have lifelike properties and unpredictable but believable behavior. Really, the "entity" here is the flock, not the individual. The size of the flock can hide the odd inconsistent movement or collision of individuals.

In the racing genre, the player is often inside the "flock," wheel to wheel with AI cars. Here, inconsistent movements can be glaring and immersion breaking. They can result in missed overtaking opportunities, poor overtake blocking, or at worst collisions with other cars. Steering behaviors were not a good fit for *F1*.

18.2.2 Lack of Context

We now understand what steering behavior failure looks like and what types of games that matters for. But we don't yet know *why* steering behavior systems have this flaw. Once we understand that, we can design a solution.

A single steering behavior component is asked to return a vector representing its decision, considering the current state of the world. The framework then tries to merge multiple decisions. However, there just isn't enough information to make merging these decisions possible. Adding prioritization or weighting attempts to make merging easier by adding more information to the behavior's result, but it translates to louder shouting rather than more nuanced debate. By making chase aware of obstacles, we can make it produce more sensible results, yet this is just special casing the merge step. That is not a scalable solution.

Sometimes, the reasons why a behavior didn't want to go any other way—the *context* in which the decision was made—is just as important as the decision itself. This is a particular problem for collision avoidance behaviors, because they can only communicate in the language of desired velocity, not undesired velocity.

18.3 Toward Why, Not How

Returning a decision, even with some metadata, just isn't going to work. Instead, what if we could ask a behavior for the context in which it would make the decision, but skip the actual decision step? If we could then somehow merge all those contexts, some external behavior-agnostic processor could produce a final decision, fully aware of everything.

The context of avoid could be, "I feel moderately strongly we shouldn't go south," and the chase context could be, "it's a little interesting to go west and quite interesting to go south." It's a holistic view rather than a resolute decision. The framework then waves a magic wand and combines these contexts, revealing that the ideal decision is to go west.

The end result is as if chase was aware of obstacles and disregarded its favorite target because it was blocked, yet the individual behaviors were focused only on their concerns. The system is emergent and has consistent collision avoidance and small stateless behaviors.

18.3.1 Context Maps

The context steering framework deals in the currency of *context maps*. Imagine everything the entity cares about in the world projected onto the circumference of a circle around the entity, as shown in Figure 18.2. It's like a 1D image, and in fact, many of the tricks we'll show you later in the chapter follow from this image metaphor.

Internally, the context map is an array of scalar values, with each slot of the array representing a possible heading, and the contents of the slot representing how strongly the behavior feels about this heading. How many slots the array has is the "resolution" of the context map. (If you're already wondering if you need huge resolutions to have decent movement, then relax. I'll show you later why you need many less than you think.) By using this array format, we can easily correlate and merge different context maps and go from slots to headings and vice versa. This is our data structure for arbitrating between different behaviors.

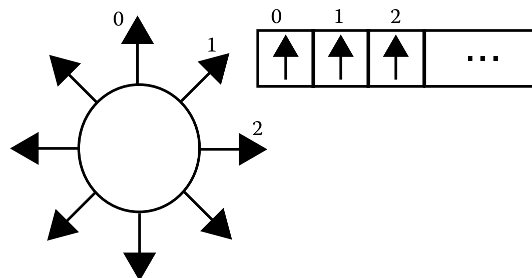

Figure 18.2

Mapping headings to context map slots.

In each frame, the framework will ask every behavior for two different context maps: the *danger* map and the *interest* map. The danger map is a view of everything the behavior would like to stay away from. As you'd suspect, the interest map is everything the behavior would like to move toward.

18.4 Context Maps by Example

What does our previous entity example look like, rewritten to use context maps? We can translate it by thinking about the information that informed the old behavior's decision and storing that information in the correct context map.

18.4.1 Chase Behavior

The chase behavior wants the entity to move toward targets, preferring near targets to far. However, choosing the best target requires making a decision, and we don't want to do that. So we're going to write all the targets into the interest map, with farther targets represented with lower intensity.

We could take a vector directly toward a target, translate that into a map slot, and write only into that slot. That captures that moving toward the target is desirable. However, we can also write over a range of slots, centered on the target with configurable falloff to zero. This captures that passing the target but just missing it is also an interesting thing to do, even if not the best. There's a lot of power and nuance in how this falloff works, giving you a lot of control over how the entity moves.

All this can be done with a quick for-each over all targets, some tuning constants, and no state. The resultant interest map is shown in Figure 18.3.

18.4.2 Avoid Behavior

The avoid behavior wants the entity to keep at least a minimum distance away from obstacles. We're going to render all obstacles into the danger map, in a very similar for-each loop to chase. The intensity of an obstacle in the danger map represents the distance to the obstacle. If the obstacle is beyond the minimum distance, it can be ignored. Again, falloff around the obstacle can be used in an interesting way. Here, it represents the heading

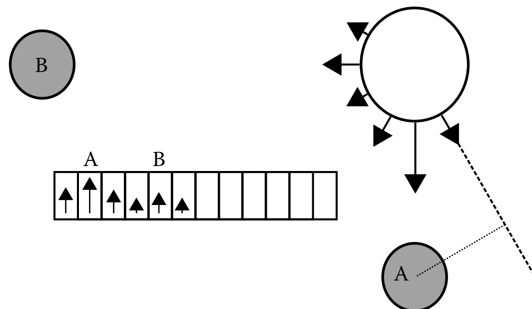

Figure 18.3

Chase behavior, writing into interest map.

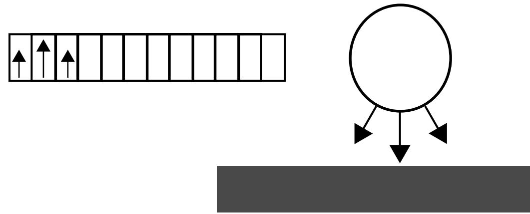

Figure 18.4

Avoid behavior, writing into danger map.

required to pass it without reentering the exclusion zone around it. This behavior is also stateless and small. The avoid behavior is shown in Figure 18.4.

18.4.3 Combining and Parsing

The output of each behavior can be combined with others by comparing each slot across multiple maps and taking a maximum. We could sum or average the slots, but we're not going to avoid a particular obstacle any more just because there's another obstacle behind it. We already must avoid the first obstacle, and that obscures any danger from the second. Through combining, we can reduce all output to a single interest and danger map pair.

The next step processes the maps, boiling down the entire shared context into a single final velocity. How this happens is game specific; the racing game example will have its own implementation.

First, we traverse the danger map to find the lowest danger and mask out all slots that have higher danger. In our example, there are some empty slots in the danger map, so our lowest danger is zero, and therefore, we mask out any slot with nonzero danger, shown in Figure 18.5(i). We take that mask and apply it to the interest map, zeroing out any masked slots (ii). Finally, we pick the interest map slot with the highest remaining interest (iii) and move in that direction (iv). The speed we move is proportional to the strength of interest in the slot; a lot of interest means we move quickly.

The final decision here is the correct decision. It is emergent—preserving collision avoidance while chasing a sensible target—yet we did it with small, stateless, and decoupled behaviors. It is the promise of steering behaviors at the macro scale.

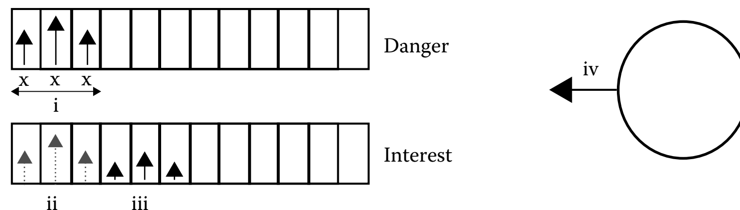

Figure 18.5

Processing the final maps.

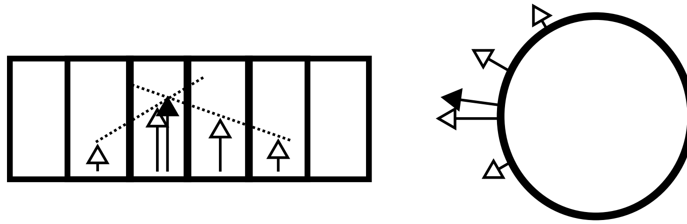

Figure 18.6

Subslot calculations.

18.4.4 Subslot Movement

You might initially think the context map is too limiting a system. The entity will always be locked to one of the slot directions, so either you need a bucketful, which sounds expensive, or you are stuck with robotic entities that can only move in very coarse directions.

It turns out we can keep the slot count low, for speed, and yet have movements in a continuous range. Once we have our target slot, we can evaluate the gradients of the interest around it and estimate where those gradients would have met. We then back-project this *virtual* slot index into world space, producing a direction to steer toward, as shown in Figure 18.6.

18.5 Racing with Context

Context steering doesn't just work for 2D entities on a plane. In fact, it is easily portable to any decision made in 1D or 2D space. Let's look at how the context steering for *F1* was implemented and how it differs from the entity example.

18.5.1 Coordinate System

We could pretend race cars moved with freedom in 2D space, but they don't. In *F1*, a low-level driver system followed a hand-placed *racing line* spline, braking for corners and accelerating down straights. The behavior system only needed to manage position on the track, rather than driving. This was done with a scalar left or right offset from the racing line. That's one of our dimensions. Although the driver will brake for corners for us, the behavior system must handle collision avoidance, so it needs to be able to slow down for emergencies. How much we want to slow down, if at all, is another scalar making our second dimension.

You can visualize the context map as a cross section of the track, with each slot representing a specific offset of the racing line, as shown in Figure 18.7. The map scales with the width of the track, with the left and right edges of the map lining up with the track edges. The racing line doesn't always map to the same slot; it will sweep from one edge of the map to the other as it moves across the track. In this and the following figures, the AI car is white.

18.5.2 Racing Line Behavior

The racing line behavior maps interest all across the track, with a peak around the racing line. It never quite reaches zero no matter how wide the track is. We only want to create a

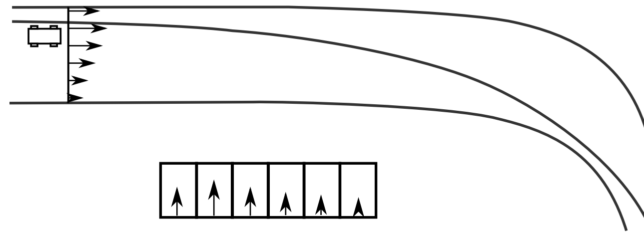

Figure 18.7

Racing line writing into interest map.

differential from slot to slot, so if the car is trapped at a far edge of the track by traffic, it always has an idea of which way is closer to the racing line and can tuck in tightly.

The behavior will write the most interest at the racing line, but never very much. Being able to reach the racing line should be good, but we want lots of room to be expressive about other interest map behaviors, while still having that important differential across the whole map.

18.5.3 Avoid Behavior

For an open-wheel racer, collision avoidance is paramount. Any type of connection (side to side or front to back) would be catastrophic. The avoid behavior evaluates all cars in the vicinity and writes danger into the map corresponding to the other car's racing line offset, with intensity proportional to the presented danger, as shown in Figure 18.8. Evaluating the danger of a car is complex. If a car is in front but at racing speed, then you should ignore them—writing danger for them will only make overtaking difficult. However, if a car is substantially below racing speed, you may need to take evasive action, so should write danger. Cars alongside are always considered dangerous. This is a good benefit of using the racing line as a coordinate system: the behavior system can be aware of a stationary car around a corner, where a raycasting approach might not see it until after the corner has been turned.

We've already seen how context steering can guarantee collision avoidance, but it can also be used more subtly. *F1* wrote high danger into the map over the width of the other car, but also a decreasing skirt of danger at the edges. This kept a minimum lateral separation between cars. The driver personality fed into this, writing wider skirts for drivers that were more cautious.

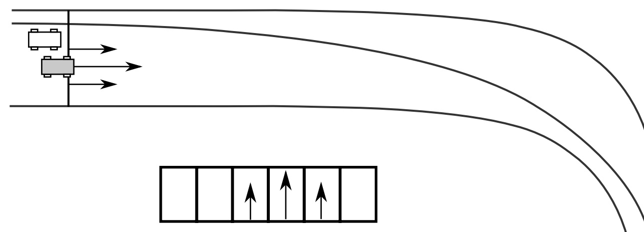

Figure 18.8

Avoid behavior written into the danger map.

18.5.4 Drafting Behavior

These two behaviors are enough for collision avoidance around a track, but it would make for quite a dull race. *F1* had four or five other behaviors that made the AI more expressive, but we'll just cover the drafting behavior.

Drafting happens when one car follows another closely and at high speeds. The trailing car doesn't need to do so much work to push air out of the way, so it can match the leading car's speed without using as much energy. At the right moment, the spare energy can be used to overtake.

F1's drafting behavior evaluated all the cars in front of the AI and scored each for "draftability." Cars going fast and near to us would score lots of points. Then the behavior would write pyramids of interest into the context maps at the corresponding racing line offset of each car, with more interest for more draftable cars, as shown in Figure 18.9.

18.5.5 Processing Context Maps

Now we have a pair of complex maps, with danger and interest in different places. How do we go from that to an actual movement? There are probably a few ways to do this that produce good consistent movement, but this is how *F1* did it.

First, we find the slot of the danger map corresponding to the car's current position on the track, shown in Figure 18.10(i). Then we walk left and right along the map, continuing as long as the danger in the next slot is less than the current. Once we cannot expand any

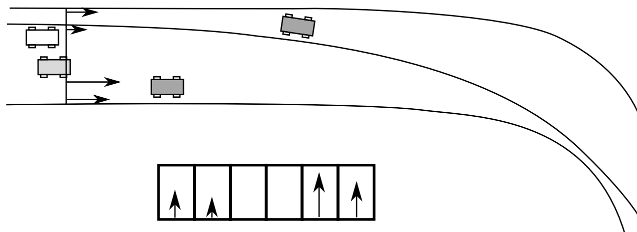

Figure 18.9

Draft behavior writing into interest map.

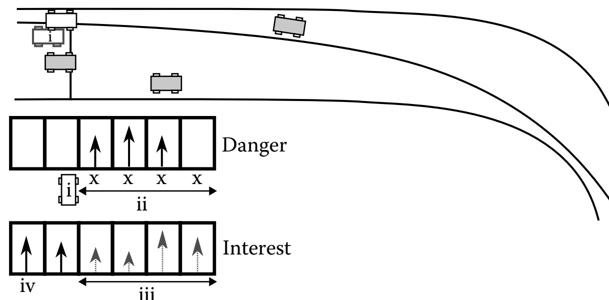

Figure 18.10

Processing the maps for a final racing line offset.

more, we mask all slots we can't reach (ii). We apply the mask to the interest map (iii) and pick the highest remaining slot (iv). The resulting movement picks the car in the far right of the figure to draft, avoiding the nearer car because it can't be reached without a collision.

This approach avoids moving into higher danger, which might represent a physical obstacle. It also stops us remaining in high danger because of high interest when there's an obvious escape route. Once we have all valid movements, it picks the most interesting of those movements.

To find out if we need to do emergency braking, we look at the highest danger across the planned journey from our current slot to the most interesting. If any slot is over some threshold of danger, we ask for braking with intensity proportional to the danger strength. We use a threshold because some danger can be informative without being a real issue, a developing situation to be aware of rather than a problem.

18.6 Advanced Techniques

There are several improvements we can make to the simple implementations outlined. These improvements are often easier to implement and maintain than their steering behavior counterparts, because they work at the level of the context map, not the individual behavior components.

18.6.1 Post-Processing

To avoid sharp spikes or troughs, we can apply a blurring function over the context maps after the behaviors have acted. As it's a global effect, it's easy to tweak and cheap to implement.

The chase behavior from our original steering behaviors example suffers from flip-flopping if the closest target oscillates back and forth between two choices. We can fix this with per-behavior hysteresis, but that adds state and complexity to behaviors. Context steering allows us to avoid flip-flopping much more easily. We can take the last update's context map and blend it with the current one, making high values emerge over time rather than instantly. This is a kind of global hysteresis that requires no support from the behaviors at all.

18.6.2 Optimizations

The overall complexity of the system is dependent on your implementation, but everything we've outlined here is linear in memory and CPU in proportion to the context map resolution. Doubling the size of the map will require twice as much memory and probably be twice as slow.

On the other hand, halving the map will double the speed. Because the system can still provide consistent collision avoidance and continuous steering even with a low-resolution map, you can construct a very granular level-of-detail controller to manage system load. Entities far from the player can be allocated small maps, producing coarser movements but requiring less system resources. Entities near the player can have more resolution, reacting to very fine details in the map. It's not very common to find an AI system that can be tweaked as subtly as this without compromising integrity.

Since the context maps are essentially 1D images, we can further optimize them using techniques from graphics programming. We can use vector intrinsics to write to

and process the map in chunks, providing a massive speed up. *F1* shipped like that, and although it made the guts of the processing harder to read, the payoff was worth it. We did that late in the project, when the implementation was nailed down.

Because the behaviors are stateless, and context maps merge easily, we can multithread them or put them on a PS3 SPU. You might also consider doing the behaviors and processing in a compute shader. Be sure to profile before and after, because some behaviors may be so simple that the setup and teardown costs of this kind of solution would be dominant. Batching behaviors into jobs or structuring the whole system in a data-orientated way is also possible. Doing this with more stateful and coupled steering behaviors would be difficult.

18.7 Conclusion

Steering behaviors remains extremely useful in many situations. If your game has individual entities that will be closely watched by the player and a world with strong physical constraints, steering behaviors can break down. For games that can be represented in two dimensions, context steering offers strong movement guarantees and simple, stateless, decoupled behaviors.

References

[Reynolds 87] Reynolds, C. 1987. Flocks, herds and schools: A distributed behavioral model. *International Conference and Exhibition on Computer Graphics and Interactive Techniques,* Anaheim, CA, pp. 25–34.

[Reynolds 99] Reynolds, C. 1999. Steering behaviors for autonomous characters. *Game Developers Conference,* San Francisco, CA.