Obstacle avoidance for robots of multiple sizes and forms in Horizon Zero Dawn

Carles Ros Martínez

1 Introduction

The world of Horizon Zero Dawn is inhabited by a family of animal-inspired robots with vastly disparate sizes and forms. While this gives the player a delightful experience, it challenges traditional obstacle avoidance systems used in humanoid-based games like the Killzone series. Traditionally, obstacles are approximated as circles to take advantage of the rotational invariant nature of the circle, but in Horizon, the elongated nature of many of the robots makes circles a poor-fitting choice. Having large robots only accentuates this discrepancy even more. We needed a different solution for Horizon Zero Dawn because it was unacceptable for the robots to stay so far apart. We opted for velocity obstacles [Fiorini 1998] using rectangular shapes.

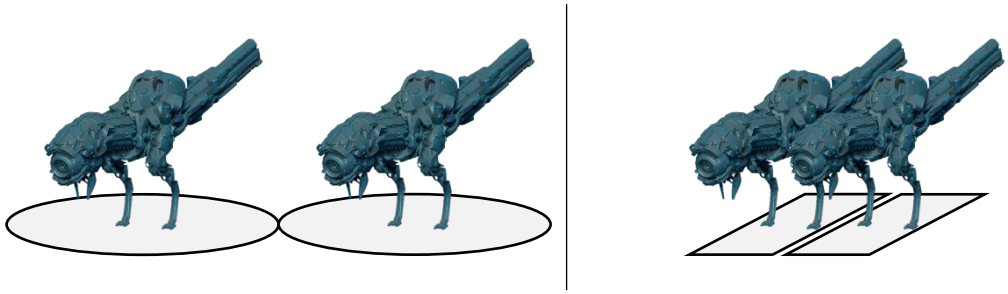

Figure 1 Two Watchers can't get close to each other if approximated by circles (left), but if using rectangles the gap disappears (right).

The velocity obstacle (VO) concept is well known in the game industry. Most papers and demos use circles to represent obstacles, but the VO formulation is actually shape agnostic, and theoretically can be used with any shape. Once more, the benefit of circles is their rotational invariant nature, which simplifies avoidance algorithms drastically, so the challenge of using any shape other than a circle is factoring in the rotation of the shape.

Our first implementation of VOs using rectangles considered only the initial orientation of the rectangles, and then assumed that the rectangle would not rotate for any chosen avoidance velocity. Obviously this is not true, as entities will typically rotate to face their movement direction. But we used this concept in our first prototype to have an idea of how well it could perform. The pleasant surprise was that this approach turned out to be good enough for Horizon, and it didn't seem worth the extra effort (and most likely performance cost) to factor in the rotation for different avoidance velocities. Although some clipping can occur when a robot changes directions abruptly nearby another robot, this was reduced by tweaking the scoring function to penalize rotations next to another obstacle, as

explained in detail later. The final game only very occasionally has clipping, and even then, it is mostly on dense scenarios with a lot of movement, where the player attention is almost always focused somewhere else.

In the next sections I go step by step through the final obstacle avoidance algorithm implemented in Horizon Zero Dawn, pointing out when a certain step has been highly tweaked for the specifics of Horizon, and if you should adjust it for your own game if planning to adopt this algorithm. At the end I present some conclusions and personal thoughts to move obstacle avoidance a step forward towards greater realism.

2 Gathering the most relevant obstacles

The first step of the avoidance algorithm is to gather those obstacles that are most relevant. The cost of generating VOs for many obstacles, and especially generating the combined VO that we'll see later, can be expensive. Intuitively, when there are many obstacles we're interested mainly in the nearby ones, but measured in collision time, not in distance [Karamouzas 2014]. Taking advantage of this, we limit the obstacle avoidance to a subset of the obstacles. Specifically, in Horizon we select up to 5 obstacles every update.

Obstacles are selected based on what we call the *conservative collision time*. This is a rough estimate of the time it would take for an entity *a* to collide with an obstacle *b* in the worst case (i.e. when *a* chooses a velocity that takes *a* the fastest possible towards *b*):

$$conservative\_collision\_time(a,b) = \frac{rough\_gap(a,b)}{max\_approach\_speed(a,b)} \quad (1)$$

where

$$rough\_gap(a,b) = d_{ab} - (r_a + r_b) \quad (2)$$

and

$$max\_approach\_speed(a,b) = s_{a_{max}} + \overrightarrow{v_b} \cdot (\overrightarrow{p_a} - \overrightarrow{p_b})_u \quad (3)$$

with d_{ab} being the distance between the center positions $\overrightarrow{p_a}$ and $\overrightarrow{p_b}$ of *a* and *b* respectively, r_a and r_b the radii of the circumcircles of the respective rectangles of *a* and *b*, $s_{a_{max}}$ the maximum speed at which *a* can move, and $\overrightarrow{v_b}$ the current velocity of *b*. See Figure 2 for an example where an obstacle farther away than a closer obstacle has a higher relevancy.

From all obstacles, we choose those which have a smaller conservative collision time. The *conservative_collision_time(a,b)* function is highly inaccurate, but it's fast and sufficient to give us a good subset of the most relevant obstacles to consider for avoidance.

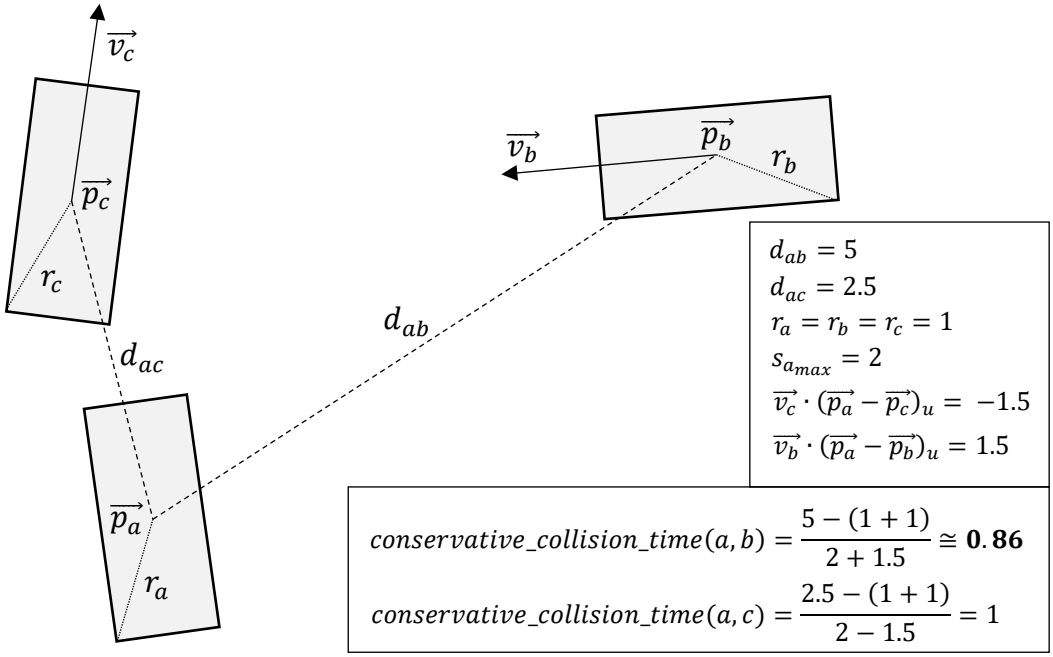

Figure 2 A farther obstacle is more relevant if it has a smaller estimated collision time.

In some situations we don't want to avoid an obstacle, but rather delegate to the obstacle the responsibility to avoid us. In Horizon there are three cases where this happens:

- **The obstacle is at the back of the entity**. During obstacle avoidance the entities have omni-perception. This is convenient, but we don't want an entity trying to avoid an obstacle approaching from behind. To prevent this we filter out all the obstacles within a 120 degrees arc at the back.
- **The obstacle has a lower avoidance priority**. In Horizon, the king of the robots doesn't step aside for puny robots. To achieve this, each robot is given an avoidance priority and robots with smaller avoidance priority are ignored. This also means that stationary robots will have to move out of the way of higher priority robots. The details of this behavior are outside the scope of this chapter, but briefly it's a combination of placing a "danger area" in front of the higher priority robot and having the stationary robots move aside when they detect that they are inside a danger area.
- **The obstacle is located beyond the target destination**. We don't want to try to avoid collisions that would occur after reaching the destination. Velocity obstacles can be built to ignore such collisions, but we'll see in the next section that we don't use this extension, and why. In practice we found that just ignoring obstacles beyond the destination is good enough. Of note here is that in Horizon the target destination is the next point on the path, and the path is "string pulled" continuously as the robot moves.

3 Building the velocity obstacles

Once we have the most relevant obstacles, we have to build the velocity obstacle for each one of them. A velocity obstacle is a well-known concept [Fiorini 1998], but in this section I will still go over an overview of how a velocity obstacle is built, and especially the particularities of how they are build in Horizon.

Consider the two robots shown in Figure 3, a Watcher and a Thunderjaw, each with a different rectangle used for its obstacle representation, and let the Watcher be the one avoiding the Thunderjaw. To generate the VO, we first want to reduce the two shapes problem to a point and a shape problem. We do this by adding the two rectangles together in an operation that's called the Minkowski sum, in which each vector of one set (the rectangle of the Watcher) is added to all the vectors of another set (the rectangle of the Thunderjaw). Visually you can imagine it as if the rectangle of the Thunderjaw is being expanded by the rectangle of the Watcher. Figure 3 shows this expansion by placing the Watcher's rectangle on each one of the vertices of the Thunderjaw's rectangle, and then connecting all the external vertices like if we were running an elastic band around them. The elastic band conforms the perimeter of the shape resulting from the Minkowski sum.

Next, we create an infinite cone by intersecting rays from the position of the Watcher with the outer bounds of the Thunderjaw (after computing the Minkowski sum). Finally, we offset this cone by the velocity of the Thunderjaw. The resulting cone is the velocity obstacle, and its main characteristic is that any velocity vector the Watcher takes which falls inside the velocity obstacle means that the Watcher will collide at some point in time with the Thunderjaw, as long as both robots keep the same velocities. On the other hand, if the velocity vector falls outside the velocity obstacle then it's guaranteed that there won't be a collision, again as long as both robots keep the same velocities.

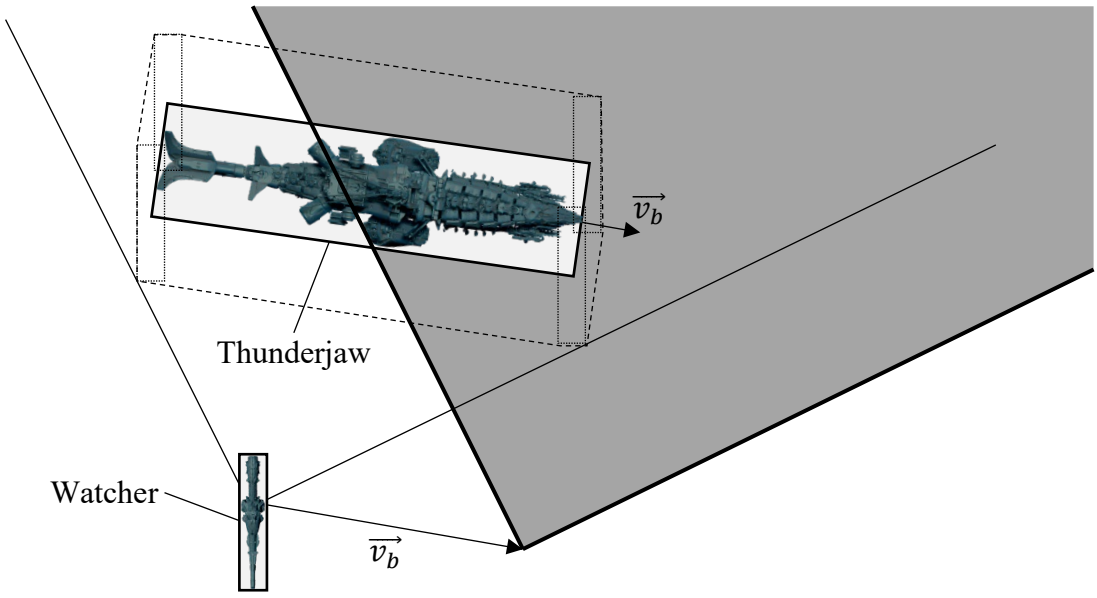

Figure 3 Velocity obstacle (dark grey cone) for a Watcher induced by a Thunderjaw.

Typically when constructing a VO you perform one additional step, which consists of

truncating the VO cone by an amount proportional to how far ahead in time you want to look for collisions. So, some velocity vectors that result in collisions beyond the time threshold will be considered collision free. Although we considered truncating the cones based on the size of the obstacle to avoid, at the end we found that we always wanted to start avoiding as soon as possible, so the VOs of Horizon are not truncated. Instead, we filter obstacles beyond a collision time threshold during the object selection phase.

This VO is specific to this Watcher-Thunderjaw pair. A specific VO has to be built for each other obstacle the Watcher has to avoid.

4 Selecting the avoidance velocity

We now have the velocity obstacles for the most relevant obstacles. This tells us which velocities are collision free and which aren't. At this point, we check if the desired velocity (i.e. the velocity the entity would move if there were no obstacles) falls outside all the VOs. If that's the case then we know the entity won't collide and we're done.

On the other hand, if the desired velocity falls inside one of the VOs, then the entity has to choose one of the velocities outside all of the VOs in order to avoid the obstacles. Selecting the best avoidance velocity among all the ones that fall outside the VOs depends on a combination of different and highly subjective criteria, but regardless of that criteria, it makes sense to assume that this velocity will fall on the edge of one of the VOs [Guy 2009][Snape 2011]. Such a velocity will avoid the obstacle by a minimum margin, which intuitively is what you'd expect.

4.1 Building the combined VO

Figure 4 shows the Watcher and Thunderjaw from Figure 3, and their VO (VO_{ab}). But now there's a second Watcher, which the first Watcher must also avoid, and so, a VO is created for it (VO_{ac}). The union of the two VOs is what I call the *combined VO*, shown in thick lines in Figure 4. We need to calculate the edges of the combined VO for the next step.

The following steps describe how the combined VO is generated in Horizon:

1. For each edge of a VO, find all the intersections among the edges of the other VOs.
2. Count how many VOs the start of the edge (the apex of the VO) is in.
3. After each intersection keep a count of how many VOs the edge is in.
4. Build a segment for each pair of consecutive intersection points outside all the VOs.
5. Repeat steps 1 to 4 for all the VOs.

All the segments from step 4 will form the perimeter of the combined VO, which is what we're looking for. Note that you'll have to deal with special cases like overlapping edges, coincident intersection points and VOs with a coincident apex.

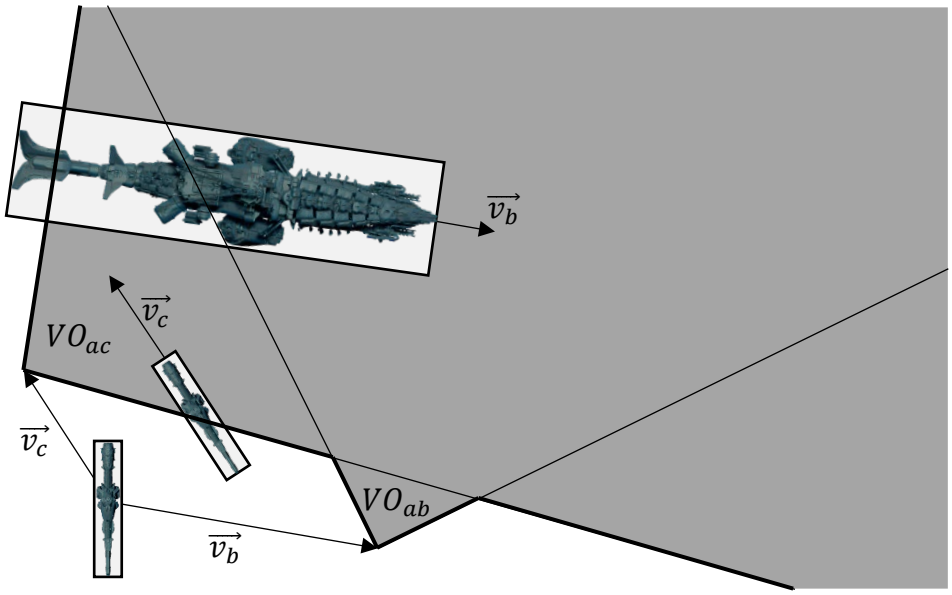

Figure 4 The combined VO (dark grey area) is the union of all the VOs.

4.2 Generating the candidate velocities

Once we have the perimeter of the combined VO, we have the subset of velocities that lie along the edge of a VO, but this still gives us an infinite choice of velocities. We need to reduce this to a small subset that can be scored in the next step. In Horizon we select as candidate velocities those velocities that meet one of the following criteria:

- Any vertex of the combined VO
- For each edge E of the combined VO:
  - The intersection of $\overrightarrow{v_{du}} \cdot s_{max}$ against E
  - The closest point on E to $\overrightarrow{v_d}$
  - The points on E whose velocity vector has a magnitude of s_d, s_{min} or s_{max}

with $\overrightarrow{v_d}$ being the desired velocity, $\overrightarrow{v_{du}}$ the unit vector of $\overrightarrow{v_d}$, s_d the length of $\overrightarrow{v_d}$, and s_{min} and s_{max} the entity's minimum and maximum movement speed respectively (see Figure 5).

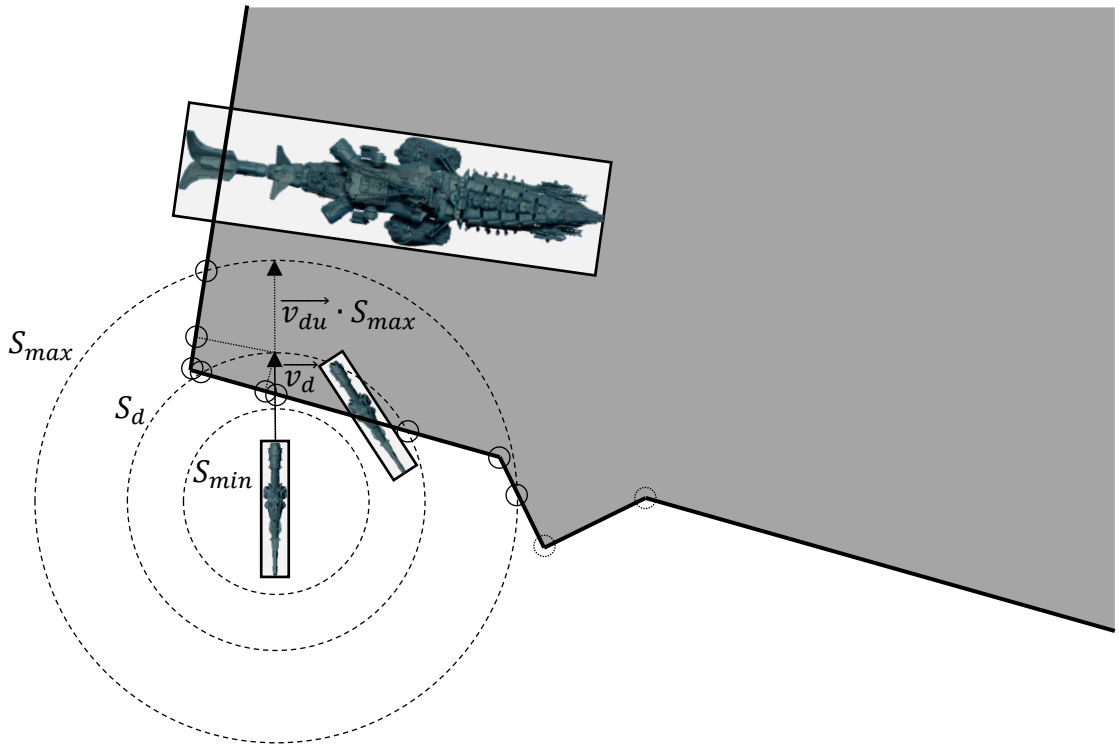

Figure 5 All the candidate avoidance velocities lay on the edge of the combined VO.

Even if there is a valid velocity that will avoid all the obstacles, entities have movement restrictions that may forbid taking that velocity. In this case we have to filter out these velocities. For example, in Figure 5 the two candidate velocities outside the allowed movement speed range of the Watcher are discarded.

4.3 Scoring the candidate velocities

The next step is to score the candidate velocities. Equation 4 shows the exact scoring function used in Horizon, but don't try to read too much into it. It's not based on any mathematical proof, but rather it's the evolution of a long story of tweaks specific to the different robot sizes in Horizon and their particular behaviors. I provide it more for inspiration than anything else, at the end you want to write your own.

$$effort = (1 - p_o)(400\alpha_d + 50\beta_d + 70\gamma) + 0.5(1 - p_t)(400\alpha_c + 50\beta_c) \quad (4)$$

All parameters in Equation 4 are normalized to [0,1]. α_d is the deviation from the desired direction, β_d is the deviation from the desired speed, α_c is the deviation from the current direction, β_c is the deviation from the current speed, and γ is a binary parameter set to 1 when the candidate velocity turns against the current angular velocity. The equation is governed by two main parameters: the proximity to a static obstacle (p_o) and the proximity to the target destination (p_t). p_t is a binary parameter set to 1 when close to the target destination. Its purpose is to prefer velocities that take directly to the destination when almost there. p_o is a linear measure of how close the entity is from colliding with the VO of a stationary obstacle, if following a linear trajectory towards the target destination. It

encourages a robot to keep the same avoidance direction when avoiding much larger obstacles or groups of obstacles forming a wall. It also helps prevent a robot from getting trapped in local minima and reduces the chance of turning when next to an obstacle, which is when clipping occurs.

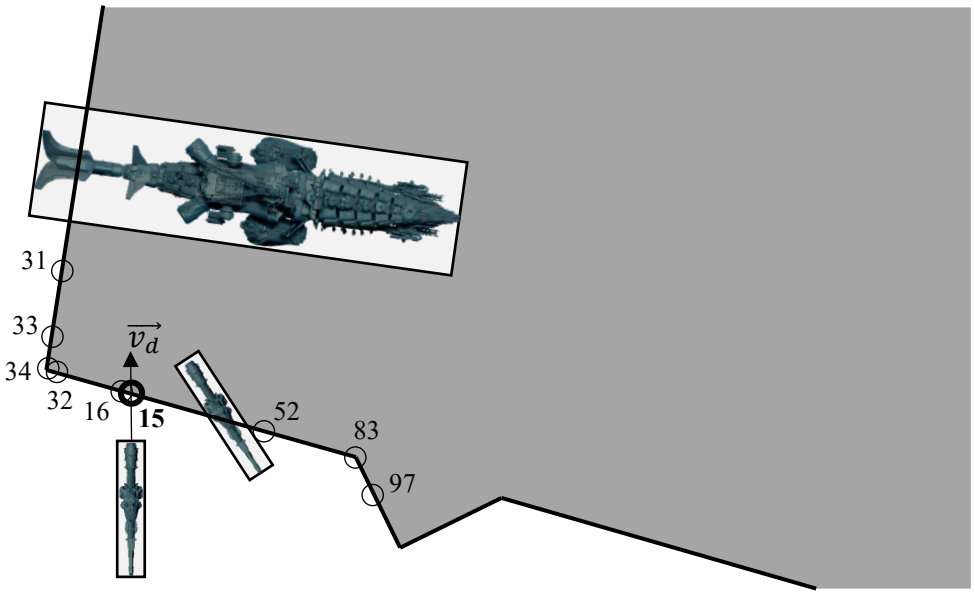

Figure 6 The Watcher prefers to slow down a bit to avoid the obstacles than to turn.

The candidate velocity with best score (or least effort if using Equation 4) is the avoidance velocity the entity has to follow for this game update. In Horizon entities move following a path, not velocity vectors. To have the entity follow the avoidance velocity we modify the path to match the avoidance velocity for one update. This is quite ugly, but does the trick.

Finally, in some situations there won't be any valid avoidance velocity. In this case a null velocity is returned, causing the robot to stop. If, after a short while, there still isn't any valid avoidance velocity, then the desired velocity is returned instead. This is rare, but it's a safety mechanism to ensure that the robot doesn't get trapped in an edge case.

5 Odds and ends

There are a couple of additional aspects related to obstacle avoidance worth mentioning. These are not part of the obstacle avoidance algorithm itself, but they have a big impact on the final avoidance quality, and movement overall.

One is the velocity of the obstacles. Instead of using the velocity of the last frame, velocities for obstacles are averaged over their last half second. This eliminates any noise in the velocity for a particular frame and also gives a more natural delayed reaction to those obstacles that suddenly change their trajectory.

Another "trick" we use in Horizon is to have the civilians in idle behavior prolong their stops when failing to find an avoidance velocity. This is inspired by observing the civilian behavior in The Elder Scrolls: Skyrim, and gives the illusion that the civilians are relaxed, which matches well with the relaxed movement they use to move around the city.

Also, regarding civilians, we sometimes observed them performing a 360 degree loop to avoid an obstacle. This looks very unnatural, so we detect when a civilian is potentially starting a 360 degree loop and force it to stop instead. This, when combined with the prolonged stop trick, results in more human-like behavior.

We must also point out that, despite this chapter being focused on rectangular-shaped robots, Horizon also has humans and robots that are better approximated by circles. We do perform the Minkowski sum between two circles, but when the VO involves a rectangle and a circle we just convert the circle to a rectangle first. The situations where both shapes mix is rare in Horizon, so it didn't seem worth the extra development time to address this more precisely.

So far, no mention has been made about static geometry. Horizon uses a navmesh for path planning and constraining entities inside the navigation space. Despite this, obstacle avoidance in Horizon entirely ignores the navmesh. Due to the open nature of the world, where most of the movement occurs in open space, the extra development time required to support this was determined to be better spent in other areas. In those cases where the avoidance hits a navmesh boundary, the entity just stops. When it starts moving again the collision usually resolves itself in one of three ways: either the obstacles the entity was trying to avoid have moved and the way is now clear, the entity chooses to avoid away from the navmesh boundary, or the entity insists on avoiding towards the navmesh boundary. If the last case keeps repeating the visual effect is an entity "squishing" through an obstacle and a wall. There were concerns that this could be too noticeable in the cities, so the decision was risky, but in retrospective I think it was the right decision. Still, I wouldn't recommend neglecting the static geometry if you can afford it - this problem will occur too frequently in games with many narrow areas. An option is the solution proposed by [Snape 2011] to handle static obstacles, where each navmesh edge would be a line static obstacle, and then a velocity obstacle would be generated for it as per usual.

One last aspect worth mentioning is the performance cost of the obstacle avoidance algorithm presented in this chapter. Its implementation in Horizon takes about 60µs per entity in dense areas (e.g. a herd of 20 robots fleeing), running on a PlayStation 4. This accounts for about ⅓ of the CPU time spent in movement, excluding animation logic.

Finally, even though Horizon uses the basic VO algorithm, it is worth mentioning variations of this algorithm which we did not use. A popular variation is ORCA [van den Berg 2011], which incorporates the knowledge that other entities are also avoiding, while also focusing on improving the performance. Then, outside the domain of velocity obstacles, some developers have favored animation driven avoidance algorithms due to the conflict of combining a steering based avoidance solution (like velocity obstacles) with an animation driven locomotion system [Anguelov 2013].

6 Conclusion

This chapter presented how Horizon Zero Dawn uses velocity obstacles for obstacle avoidance, and it shows how constructing velocity obstacles from rectangles can give good enough results for elongated obstacles, even without fully factoring in the rotation of the rectangles. All in all, I'm quite pleased when I see two long robots walking side by side without leaving any large gap, as it would have happened if we'd use circles instead of

rectangles. The biggest caveat I have is that we use a velocity based system for avoidance, while our path following solution prioritizes animation over velocity vectors, meaning path following and avoidance systems are fighting against each other. Also, velocity obstacles result in vehicle-like movement, which looks, ironically, robotic, when applied to life-like entities. These two points are something that we'd like to solve in a future game.

After Horizon was released we concluded we were pleased with how the robots move and avoid, but not entirely with the humans, especially the crowds in the cities. It was not so much the glitches in the avoidance, but their lack of realism. Even when civilians perform a perfect avoidance, in many occasions it still looks weird for a human observer, something that doesn't occur with the robots. Our conclusion was that this is caused by our familiarity with human avoidance in real life, while we don't have such association with the robots. Subsequently I spent a few days looking at videos of real crowds, hoping to find patterns we could implement to improve our crowds. Looking at these videos it convinced me that stepping away from an algorithmic approach is the only way to achieve a big leap forward in obstacle avoidance realism and crowd simulation in general when it comes to humans. Too many irregular perturbations in their movements, too many gestures whose context is hard to pinpoint, too many differences among characters of the same crowd, too many differences depending on the location, mood, age, personality, social norms, goals, weather, culture, and on and on.

Another important conclusion from observing real crowds is that humans are not that good at avoidance. Even though we have the capability of performing perfect avoidance, in a real crowd you can observe many different levels of avoidance quality. What humans are excellent at is at recovery behavior from a late avoidance, or even a failed avoidance. Again, the set of 'recovery' behaviors and movement is huge, with many nuances depending on the person, context, etc.

Given the difficult nature of defining what realistic obstacle avoidance is, I feel like a better approach is try to imitate real life human avoidance. This is the field of machine learning, and it's what we're currently researching at Guerrilla.

References

[Anguelov 2013] B. Anguelov. 2013. Collision Avoidance for Preplanned Locomotion, *Game AI Pro*: 297-305.

[Fiorini 1998] P. Fiorini, and Z. Shiller. 1998. Motion planning in dynamic environments using Velocity Obstacles. *International Journal of Robotics Research* 17, no. 7 (July): 760-772.[van den Berg 2011] J. van den Berg, S. J. Guy, M. Lin, D. Manocha, C. Pradalier, R. Siegwart, and G. Hirzinger (eds.). 2011. Reciprocal n-body Collision Avoidance. *Robotics Research: The 14th International Symposium ISRR, Springer Tracts in Advanced Robotics* 70: 3-19. Springer-Verlag.

[Karamouzas 2014] I. Karamouzas, B. Skinner, and S. J. Guy. 2014. Universal Power Law Governing Pedestrian Interactions. *Physical Review Letter* 113, no. 23 (December): 238701.

[Guy 2009] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, and P. Dubey. 2009. ClearPath: Highly Parallel Collision Avoidance for Multi-Agent Simulation. *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer*

Animation: 177-187.
[Snape 2011] J. Snape, J. van den Berg, S. J. Guy, and D. Manocha. 2011. The Hybrid Reciprocal Velocity Obstacle. *IEEE Transactions on Robotics* 27: 696-706.