Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Reciprocal Collision Avoidance with Maximum Velocity and Acceleration Constraints

Students' Scientific Conference 2015

Author
Varga Bálint Viktor

Supervisor
dr. Blázovics László

October 26, 2015

Contents

Abstract

Multiple methods were developed for local collision avoidance among navigating agents. Several of these methods are based on the basic concept of velocity obstacles (VO). One such method, called optimal reciprocal collision avoidance (ORCA) uses linear constraints, and an efficent low dimensional linear programming algorithm to compute the agents' new velocities. In this paper it is shown that the algorithm can be extended with any convex shaped constraints, such as circles, that can be used to represent simple dynamic constraints, like the agent's maximum velocity, and its maximum acceleration. The extended algorithm has been implemented in a game engine where its performance and usability were tested in several complex scenarios.

Keywords: collision avoidance; constraints; linear programming; ORCA; game engine

Összefoglaló

Kölcsönös ütközés elkerülés maximális sebesség, illetve gyorsulási kényszerekkel

Több megoldás is született a navigáló ágensek helyi ütközés elkerülésére. Számos megoldás ezek közül a velocity obstacle (VO) fogalmát használja fel alapul. Az egyik ilyen megoldás, az Optimális Kölcsönös Ütközés Elkerülés (optimal reciprocal collision avoidance, vagy ORCA) lineáris kényszerekkel, és egy hatékony alacsony dimenziójú lineáris programozási algoritmussal számolja ki az ágensek új sebességeit. Ebben a dolgozatban megmutatom, hogy az algoritmus kiegészíthető bármilyen konvex alakú kényszerrel, mint például körökkel, amelyekkel egyszerű dinamikus kényszereket jellemezhetünk, mint a maximális sebesség, és a maximális gyorsulás. A kiegészített algoritmust implementáltam egy játékmotorban, ahol annak teljesítményét, és használhatóságát vizsgálom néhány bonyolult esetben.

Kulcsszavak: ütközés elkerülés; kényszerek; lineáris programozás; ORCA; játékmotor

Introduction

An autonomous agent is an agent capable of independent decisions, I will usually refer to them as units. Units can exist in games, virtual realities, or in actual real life environments as robots. Sometimes we can regard ourselves (humans) units too. Units have sensing capabilities, they can make decisions based on what they sense and what they know, and they can move according to these decisions, this is called the look-compute-move model.

Units can have diverse destinations planned for them, and it is a rather difficult task to move to these destinations without colliding with the environment, or other units. Usually the task is split in half to two smaller tasks:

First is the high level task of finding a valid path from start to destination by only considering the terrain and its solid (non-moving) obstacles: pathfinding

Second is the low level task of avoiding collisions with other units (and obstacles) while following the path found : local collision avoidance

The focus of this paper is the low level artificial intelligence task of local collision avoidance (the second task).

This paper is organized as follows. In Chapter 1 related works and background information are introduced. Chapter 2 describes the deeper details of ORCA, and my extension for the ORCA algorithm. In Chapter 3 details about the implementation of the algorithm in a game engine are provided. Chapter 4 shows the results for some test scenarios. Chapter 5 shows the solution's scalability results. Finally Chapter 6 concludes the work.

Chapter 1

Related Works and Background Information

Collision avoidance has many approaches. Initially solutions tended to use repulsive forces to make units avoid colliding with their environment and each other. These forces can be the result of the near obstacles and units, or potential fields. Early on, Reynolds had succeeded in simulating flocks of birds using steering behaviors which is a force-based approach [1] [2].

The velocity based approaches can provide more robust solutions, where a unit can have more direct control over it, and still avoid collisions. Such a velocity based approach is the velocity obstacle approach.

For the sake of simplicity, a unit's shape is approximated with a circle, and the terrain is considered a 2D plane. With these simplifications it is relatively easy to deduce the Velocity Obstacle (VO) concept. It was first introduced by Fiorini and Shiller [3].

1.1 Velocity Obstacle approach

Two circular units (A, and B) are examined in a plane. Let A and B denote their positions, r_A, and r_B their radiuses, v_A, and v_B their velocities. The units can be seen in Figure 1.1.

The velocity obstacle for A induced by B is the set of velocities of A that will cause a collision with B if they both have constant velocities. That happens exactly when the distance between A and B is smaller than the sum of their radiuses, so after exactly t time:

$|A + v_A \cdot t - (B + v_B \cdot t)| < r_A + r_B$

From this equation it might not be trivial how a VO looks. Notice that the only variables are v_A, and t. For any given t, for example $t = 1$, the equation is just a circle, its center at $v_B + (B - A)$, its radius $r_A + r_B$ To make it simpler, let's rearrange the equation:

$|v_A - v_B - \frac{B-A}{t}| < \frac{r_A+r_B}{t}$

Now let's check some other t parameters:

$t = 2$ –> circle, center: $v_B + \frac{B-A}{2}$, radius: $\frac{r_A+r_B}{2}$

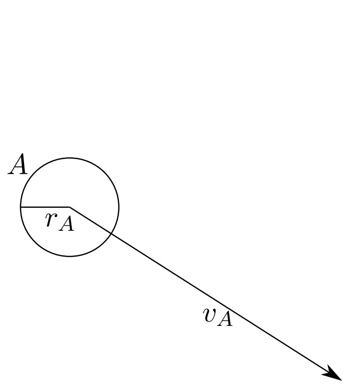

Figure 1.1: units

$\lim t = \infty$ −> the circle has a radius of 0, its center at v_B, so it's just one point.

If we draw all these circles, the area they covered would be an infinite cone, whose peak is at v_B, and its direction is $(B - A)$. Its edges would be tangents to every circle we might get by substituting t for any parameter. The cone shape can be explained by the following analogy:

If A takes B's coordinate system as its own (moving with v_B), but with the origin at A's position, A sees B essentially motionless. In this coordinate system, any velocity inside the cone will eventually result in a collision. The cone's edges (tangent lines to the bigger circle from A's position) represent velocities for A to just touch B on the left or the right side. To translate A's colliding velocities back from B's system, the cone has to be translated by v_B. These circles and the cone itself can be seen in Figure 1.2.

There are many VO based solutions for local collision avoidance with slight differences how they handle reciprocity, the algorithm's complexity, and the solution's ability to avoid collisions, generate smooth paths, and the time to resolve a test scenario. The currently available solutions are reviewed in Snape et al[4], but short descriptions can be read here too.

Usually there are convex or non-convex areas defined that represent the available or the restricted areas, and the closest point to a destination point (usually the preferred velocity) among the points in the available area needs to be selected. In general, this leads to a quadratic optimization problem with non-convex constraints.

VO

The basic Velocity Obstacle approach was successfully used for collision avoidance where a unit had to avoid collisions with moving obstacles [3]. The basis of the algorithm is that in each cycle, for the collision avoidance maneuver the unit chose a velocity outside any of the velocity obstacles (Figure 1.2). However, when there are multiple units using this same approach to avoid collision with each other, the units' velocities are oscillating because each unit assumes that other units will not change their velocities.

RVO

Reciprocal Collision Avoidance intends to solve the basic approach's oscillation by having reciprocity among units. Reciprocity can take many forms. Reciprocal Velocity Obstacles [5] are simple VO-s that are translated, so that when units select

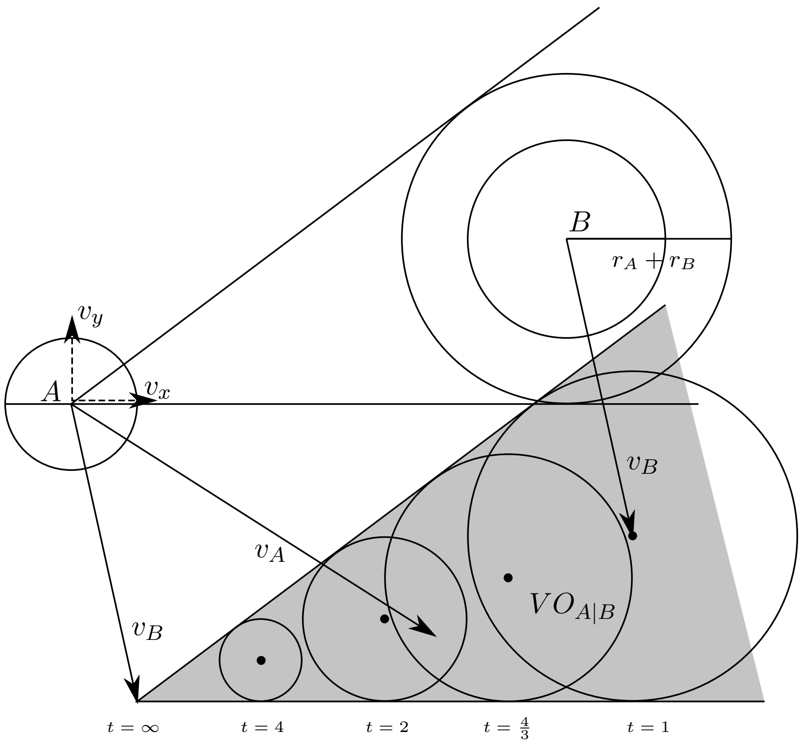

Figure 1.2: Velocity Obstacle

velocities outside these VO-s, the responsibility of avoiding collisions is shared. A 50:50 sharing can be seen in Figure 1.3.

If units choose velocities outside their correspondent RVO, and choose to pass each other on the same side, then the RVO method guarantees that they won't collide, and they won't oscillate. Choosing the closest velocity outside of the RVO, the units will pass each other on the same side. Units however usually need to move to a target position, so in order to get there, they choose velocities closest to their preferred speed. Without the guarantee, a new type of oscillation called reciprocal dances can appear, which is due to the units not choosing the same side.

HRVO

Hybrid Reciprocal Velocity Obstacle addresses reciprocal dances by discouraging units to pass on different sides. This is done by translating the RVO so that when a unit's velocity is to the right of its centerline (CL), then the RVO is translated left, so the unit should choose a velocity to the right of the centerline (Figure 1.4). Thus the HRVO consists of a side of the original VO, and a side of the RVO, which is why it's called hybrid. Further details can be read in Snape et al [6] [7].

ORCA

Optimal Reciprocal Collision Avoidance is another approach for addressing the reciprocal dances seen in RVO. It uses a truncated velocity obstacle ($t < \tau$), and computes smallest change vectors (u) to the relative optimization velocity to avoid collision. These smallest change vectors are then used to compute half-planes (position: $v_A^{opt} + u/2$, direction: n which points outwards from $v_A^{opt} - v_B^{opt} + u$ on the boundary of the truncated VO) that represent collision free velocities (Figure 1.5).

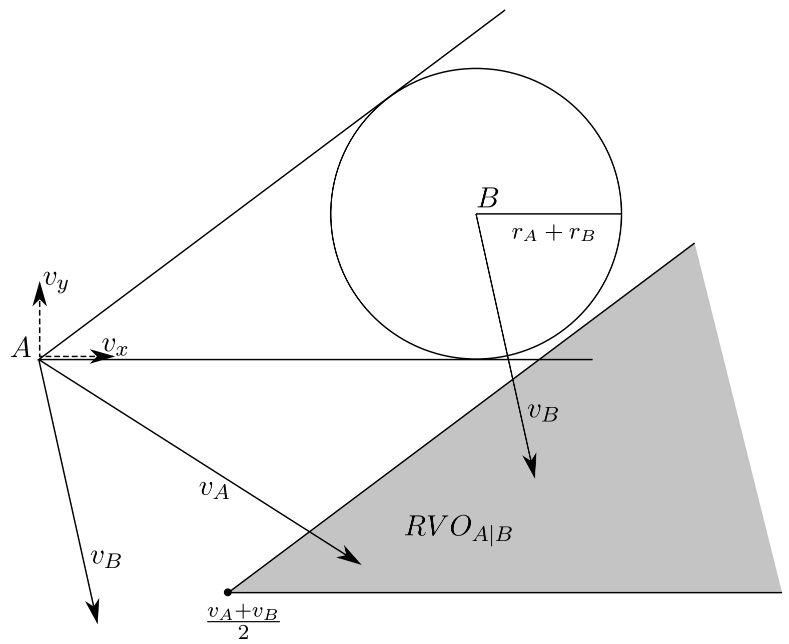

Figure 1.3: Reciprocal Velocity Obstacle

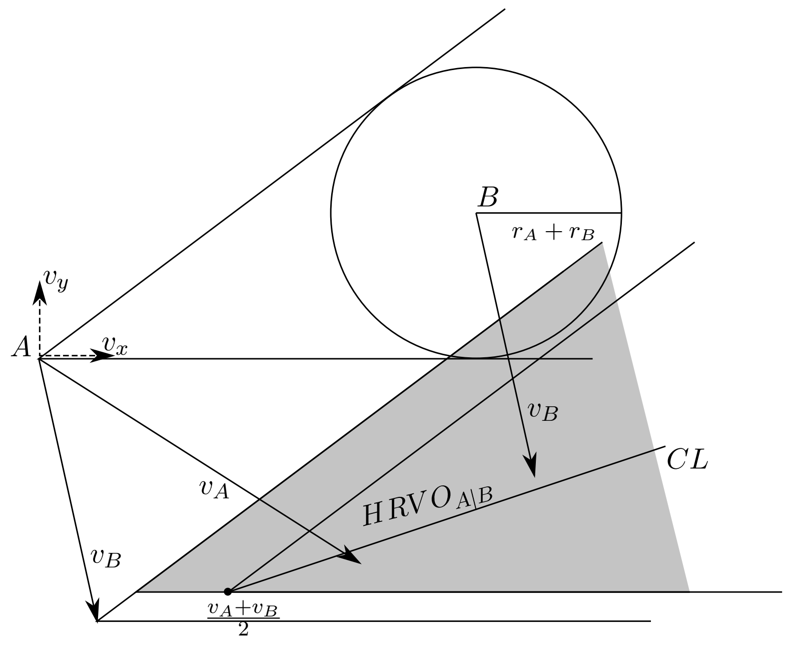

Figure 1.4: Hybrid Reciprocal Velocity Obstacle

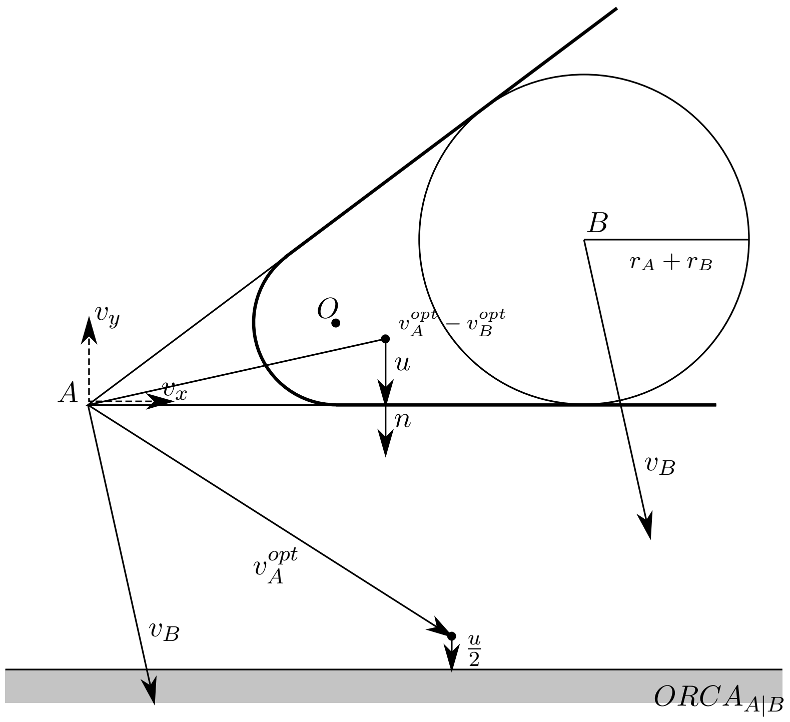

Figure 1.5: Optimal Reciprocal Collision Avoidance with $\tau = 2$

Due to the simplicity of the half-plane constraints, this type of collision avoidance can be computed very efficiently, solving its optimization problem involves randomized linear programming.

On a 4 year old cpu (released in 2011), it took an average time of 241 μs per time step to simulate the Circle-n scenario with $n = 1000$. Circle-n scenario can be viewed in Chapter 4.

I will go through details about ORCA in Chapter 2.

ClearPath

While ORCA uses simple linear constraints, that eliminates a lot of velocities that would not cause collisions, so it is possible that ORCA does not find a solution (or finds a relaxed one which does not guarantee collision), when in fact there is a velocity that would be collision free.

In this regard, RVO, and HRVO can be much less conservative (when their VO is truncated like in ORCA), and as such, they are much more likely to find a collision-free velocity than ORCA. However, finding an optimal solution to RVO's and HRVO's optimization problems is not self-evident. See Figure 1.6 for an illustration.

For simplicity random sampling is used sometimes to approximate the optimum, but there is a better algorithm.

ClearPath is an efficient highly parallel algorithm which can solve RVO's and HRVO's quadratic optimization problems as well with truncated velocity obstacles (truncated VO like in ORCA, but the round edge is approximated with a line, the overall VO they use is called FVO). It is an earlier concept than ORCA, and computationally more difficult, but fast nevertheless.

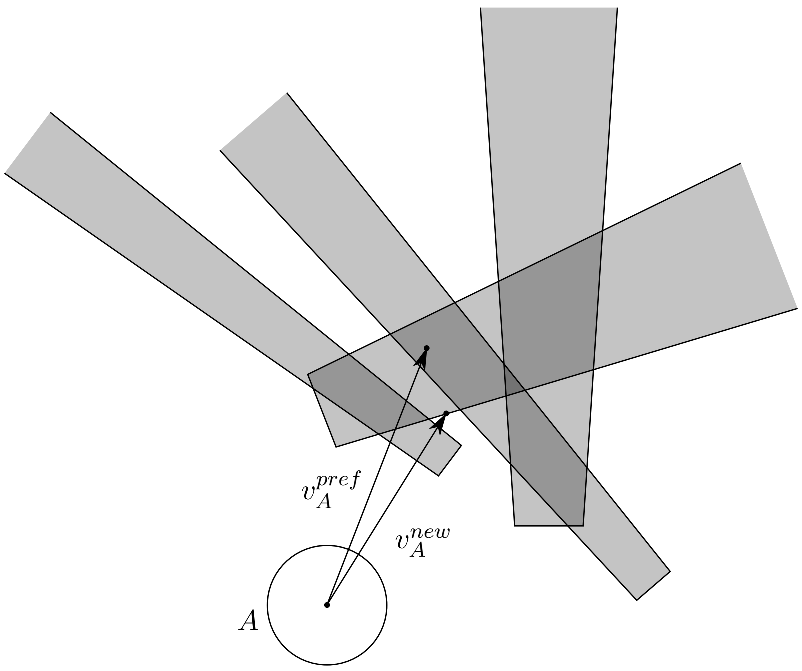

Figure 1.6: ClearPath with FVOs

ClearPath made impressive results, simulating a few hundred agents took 2.5 ms on a single Larrabee core, but the algorithm could handle hundreds of thousands of agents on a simulated Larrabee many-core processor in only 35 ms. Although the Larrabee GPGPU chip was cancelled, the algorithm is still an order of magnitude faster than prior VO-based methods. ClearPath made it clear that it's worth it to solve these problems in parallel on a multi-core processor.

Read more about ClearPath in Guy et. al[8].

Chapter 2

Optimal Reciprocal Collision Avoidance extended

In this chapter I present my extended ORCA solution that can handle maximum velocity, and maximum acceleration constraints. To understand it, we will go through the important ORCA details first. (A short description of the ORCA plane, and a Figure can be seen at the end of Chapter 1, further details can be read in Berg et al[9]).

2.1 ORCA details

ORCA uses a limited velocity obstacle, which is a VO, where we don't care about collisions that happen after τ time. This is sufficient because if we always avoid collisions for τ time in each cycle, there still won't be any, and since this limited VO is reduced in size compared to the original VO, it gives units more options to choose from. (For example when a group of objects are all going with the same speed and direction, they won't collide: $v_A^{opt} - v_B^{opt} = 0$, which is outside the limited VO), or we can say that they will collide after infinite time - which is the not limited VO's peak.)

The optimization velocity v_A^{opt} of unit A is the velocity used for calculating the smallest change vectors. A good choice of the optimization velocity is the current velocity : $v_A^{opt} = v_A$, but there can be other choices [9]. From here on I will refer to the relative optimization velocity as v_{rel}^{opt}, or simply v_{rel}, since I chose the current velocity for the optimization velocity.

The smallest change vector u is then calculated which is the smallest change to get the relative optimization velocity outside (to the boundary) of the limited VO. This is the vector from the endpoint of v_{rel} to one of the following:

- orthogonal projection of v_{rel} to the left side of limited VO
- orthogonal projection of v_{rel} to the right side of limited VO
- orthogonal projection of v_{rel} to the small circle located at $O = A + \frac{B-A}{\tau}$ with a radius of $\frac{r_A + r_B}{\tau}$.

When the smallest change vector is calculated then an ORCA constraint is created which is a half-plane (linear constraint). This method is reciprocal, so units share the responsibility for avoiding collisions (this is not the case for static obstacles). Reciprocity with a 50:50 sharing of responsibility means here that unit A has to change its velocity by at least $u/2$ (if the relative velocity is inside the limited VO, otherwise u is not halved), and unit B has to change its velocity by $-u/2$ (same here). This can be described with a half-plane. A point on the half-plane's edge is $v_A^{opt} + u/2$. Its normal's (n) direction is the outward direction from v_{rel} to the closest point on the boundary of the limited VO, and the available points lie towards n.

For a unit (A), all the other nearby influental units induce an ORCA constraint. The unit then needs to select a velocity inside all the ORCA half-planes, preferably something close to the unit's desired velocity (preferred speed). This problem is formulated as follows:

$$\begin{aligned} \underset{v}{\text{minimize}} \quad & |v - v_{pref}| \\ \text{subject to} \quad & A \cdot v < b \end{aligned} \tag{2.1}$$

where A is a $n \times 2$ matrix, v and v_{pref} are two-dimensional vectors, b is an n dimensional vector, and there are n half-plane ORCA constraints (one orca constraint is 3 numbers: a row in A matrix, and the corresponding number from b vector).

I'm going to refer to this problem as CPLP for Closest Point Linear Programming.

While this problem may seem familiar, it's not as easy as a simple 2D linear programming problem. The constraints are linear, but the objective function is quadratic. But before we solve the problem with a quadratic optimizer, it's worth noting that the problem is quite special with a low dimension. Special cases require special measures, and with the geometric interpretation of the problem, we can come up with our own solution. The geometric interpretation is the same as mentioned before: we need the closest point to another point (let's call this point *destination*, or *dest* for short) that is inside all the linear constraints. An example is shown in Figure 2.1.

Since the constraints are linear, the available area is convex (convex sets' intersection is convex).

Fiddling with this problem we can quickly discover, that the closest point to *dest* can be one of the following:

- orthogonal projection of *dest* to one of the linear constraints
- an intersection point of two linear constraints
- *dest* itself, if it's inside all the linear constraints

That is if the problem is feasbile.

With this in mind we can create our naive algorithm for this problem (Algorithm 1).

The naive algorithm is already not so bad, it is polynomial in running time ($O(n^3)$ with n being the number of constraints).

But we can do even better, there is actually an efficient randomized algorithm for this that is not too difficult to write [10]. The algorithm adds the constraints one by one in randomized order while maintaining the current best solution (Algorithm 2).

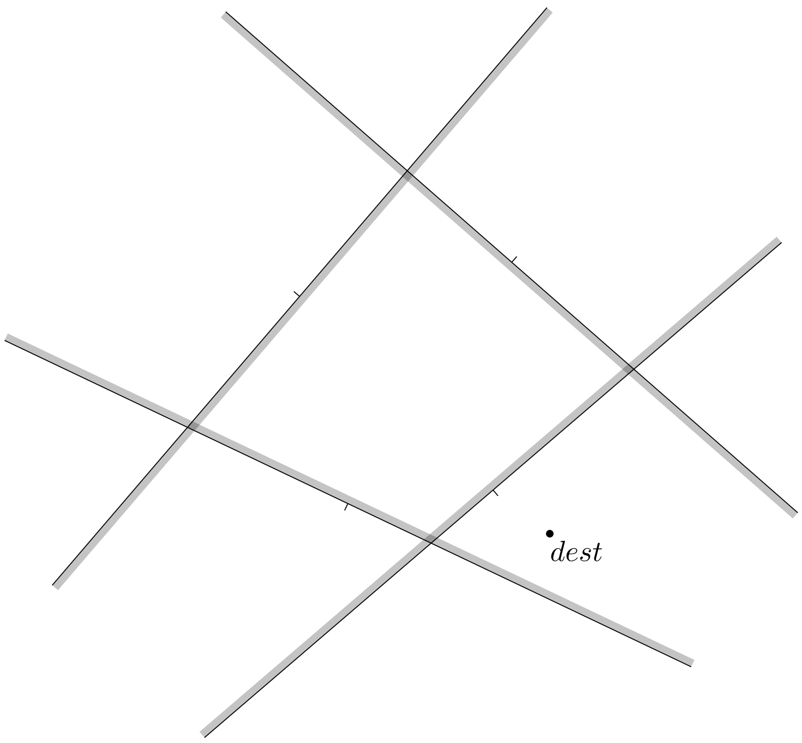

Figure 2.1: Closest Point Linear Programming, the grey area represents the allowed half-plane for a linear constraint

Algorithm 1 Naive CPLP algorithm

```
1: $C \leftarrow$ set of all linear constraints
2: $dest \leftarrow$ the destination point
3: $feasible \leftarrow false$
4: $closestYet$ : the closest of checked points
5: update $closestYet$ with $dest$, and set $feasible$ to $true$ if better and satisfies all $c \in C$
6: **for** $c_1 \in C$ **do**
7:     $point \leftarrow$ orthogonal projection of $dest$ on $c_1$
8:     update $closestYet$ with $point$ ... (line 5)
9:     **for** $c_2 \in C$, $c_2 \neq c_1$ **do**
10:        $point \leftarrow$ intersection of $c_1$ and $c_2$
11:        update $closestYet$ with $point$ ... (line 5)
12:    **end for**
13: **end for**
14: **return** $closestYet$, $feasible$
```

Algorithm 2 Randomized CPLP algorithm

```
1: $C \leftarrow$ set of all linear constraints
2: $dest \leftarrow$ the destination point
3: $feasible \leftarrow true$
4: $closestYet \leftarrow dest$
5: $checkedC \leftarrow \varnothing$
6: for next random $c_1 \in C$, $c_1 \notin checkedC$ do
7:     if $closestYet$ satisfies $c_1$ then
8:         $checkedC \leftarrow checkedC \cup \{c_1\}$
9:         continue
10:    end if
11:    $feasible \leftarrow false$
12:    $point \leftarrow$ orthogonal projection of $dest$ on $c_1$
13:    update $closestYet$ with $point$, and set $feasible$ to $true$ if better and satisfies all $c \in checkedC$
14:    for $c_2 \in checkedC$, $c_2 \neq c_1$ do
15:        $point \leftarrow$ intersection of $c_1$ and $c_2$
16:        update $closestYet$ with $point$ ... (line 13)
17:    end for
18:    if $feasible = false$ then return
19:    end if
20:    $checkedC \leftarrow checkedC \cup \{c_1\}$
21: end for
22: return $closestYet$, $feasible$
```

The basis of the algorithm is that when a new constraint violates the current best solution, it means that the new best solution must be on that new constraint somewhere.

The randomized algorithm has a linear expected running time ($O(n)$, n : number of constraints). Berg et. al [9] also mentions they included a circular constraint for the maximum speed, which does not alter the algorithm significantly, or its running time.

The question arises though what point to choose when the optimization problem is infeasible (the intersection of the constraints is empty).

In that case, there was a constraint whose orthogonal projection point, and all other constraint intersection points did not satisfy the already checked constraints. If that happens, the constraints are relaxed, and the problem can be solved using 3D linear programming.

A linear constraint can be seen in Figure 2.2. Its equation can be derived from a normal vector $n(A, B)$ and a point on the edge $Q(x_0, y_0)$, and $P(x, y)$ denoting the points inside:

$$\begin{aligned} n \cdot \overrightarrow{QP} &< 0 \\ (A, B) \cdot (x - x_0, y - y_0) &< 0 \\ Ax + By &< Ax_0 + By_0 = C \end{aligned} \tag{2.2}$$

In this case, the (A, B) vector's direction points away from the available points of the half-plane. When a constraints is relaxed, its edge moves d distance in the direction of (A, B): $n \cdot d$ (for this the normal vector (A, B) needs to be normalized otherwise $n \cdot d$'s length

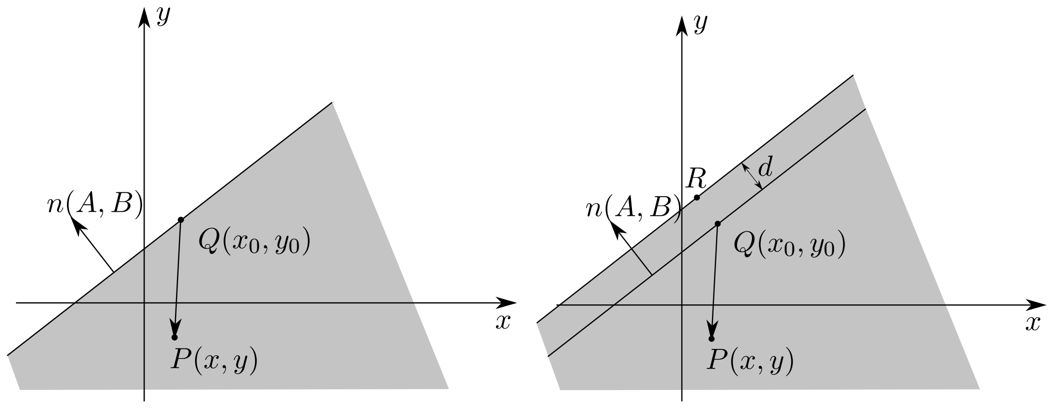

Figure 2.2: Linear constraint, and its relaxation

will be $d \cdot \sqrt{A^2 + B^2}$). Let R represent Q translated by $n \cdot d$. The equation will be the following:

$$\begin{aligned} Ax + By &< A(x_0 + d \cdot A) + B(y_0 + d \cdot B) \\ Ax + By - d \cdot (A^2 + B^2) &< C \\ Ax + By - d &< C \end{aligned} \tag{2.3}$$

This is how 2D constraints become 3D with the third variable being d. In the 3D problem, we need to find the smallest possible d value which makes the intersection of the ORCA lines not empty. If that value is found, then the closest point to *dest* is selected in the emerging convex shape (usually a point, but with parallel lines a line or a line segment is possible) using our regular 2D algorithm. The resulting point can be considered the safest possible velocity which penetrates the unrelaxed constraints minimally (another way of saying this is : the constraints are moved in the direction of their normal vector (A, B) with equal velocity until the 2D problem becomes feasible).

The d value can be found by checking the relevant points of the constraint that caused infeasibility (let's call it failConstraint). Such relevant points include angular bisector intersections (incircle center) of three constraints (one of them being failConstraint), and the orthogonal projection of *dest* on an angular bisector of two constraints (one of them being failConstraint).

The 3D problem is always feasible (with a great enough d distance, all constraints must be satisfied), and while it is more complex to solve, the overall expected complexity remains $O(n)$.

2.2 Extension

A unit's movement usually involves constraints. My extension for the ORCA above is handling maximum accereration constraints, and maximum velocity constraints as well (although the latter is said to be included already in Berg et al.[9]).

In my extension the unit is a circle, it has a velocity, it has symmetrical acceleration capabilities, and it has a maximum velocity. Its rotating abilities and any other constraints

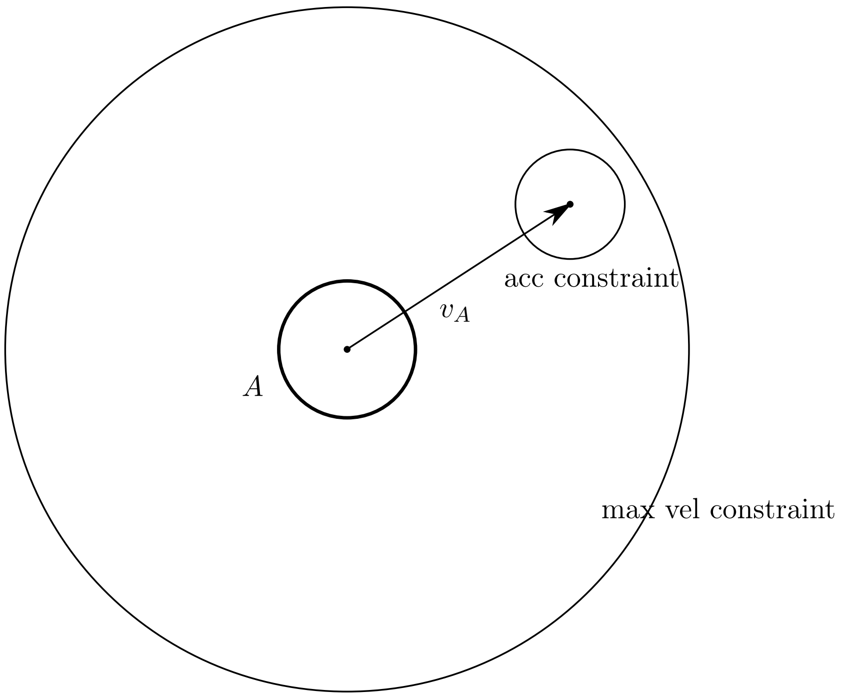

Figure 2.3: Maximum velocity and acceleration constraints

that could be present are ignored. Without ORCA constraints the unit and these attributes can be viewed in Figure 2.3.

The interpretation of Figure 2.3 is that we regard unit A, like it can choose any velocity for itself that is inside the acceleration circular constraint, and the maximum velocity constraint (also circular). For example if a unit was set to have a 100 m/s^2 maximum acceleration, and the delta time was 15 ms, then the acceleration circular constraint's radius (the maximum allowed change of velocity) will be 100 * 0,015 = 1,5 m/s, centered at its velocity as always. The maximum velocity constraint is not affected by the elapsed time, it will be a constant radius around A's center.

All the constraints including maximum velocity, acceleration, and ORCA constraints can be viewed in Figure 2.4. The darker grey convex area represents the intersection of all the constraints in which we need the closest point (v_A^{new}) to v_A^{pref}, which is in this case an intersection point between the acceleration constraint , and the top-right ORCA constraint.

For this optimization problem, I extended the original randomized CPLP algorithm (Algorithm 2) to one that can handle any number of circular constraints in addition to the linear ones, but instead of (or as an addition to) a circle shape, any convex shape could be integrated provided that you can calculate:

- the closest point of the shape from another point
- intersection points of any two convex constraint type edges
- closest point of the shape from a line (this is only needed for the relaxed 3D problem)

Keep in mind however, that the number of intersection calculation types increase quadratically in the number of the shape types, so having many types is not advised. Also, polygon shaped constraints can be perfectly put together using simple lines, and many shapes can

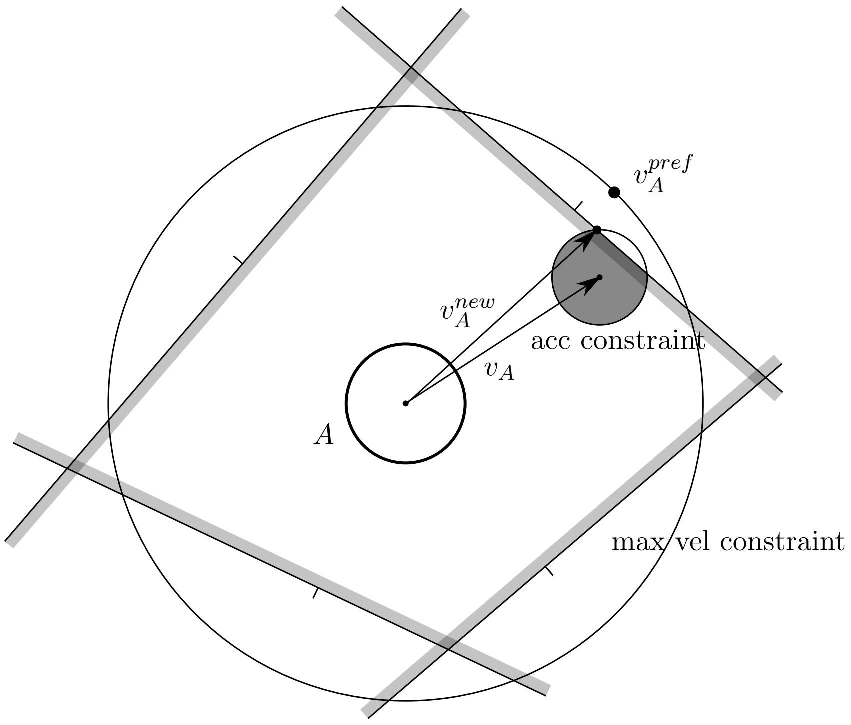

Figure 2.4: Maximum velocity, acceleration constraints and ORCA constraints, $dest = v_A^{pref}$, v_A^{new} is the solution

be approximated well-enough with polygons. I think new shapes can become convenient when using some shape with round edges, like circles, ellipses, generally shapes that may need many lines to be approximated fairly.

The intended purpose for this extension is to handle the constraints which reflect the unit's movement capabilities too, these constraints must not be relaxed when solving the relaxed 3D problem (only linear ORCA constraints are relaxed). Since these constraints are not relaxed, it is essential that the 3D problem is always feasible so that there is always a good enough velocity chosen given the circumstances. This is true for having just maximum velocity, and acceleration as circular constraints (the acceleration circle will always have a not empty intersection with the maximum velocity circle).

The new, extended algorithm can be seen in Algorithm 3. It is much the same as before, except

- orthogonal projection is replaced with a more general name as closest point (which works for any convex shape)
- the inner for loop now goes through all intersection points of two constraint edges (two convex edges can only have a maximum of 2, or ∞ intersection points, but ∞ means they have a straight line segment in their edges, which should be a line constraint, but even with a line segment, only its two endpoints are relevant)

The relaxed, 3D problem is also similar to the one without circle constraints, and can be implemented similarly.

Algorithm 3 Randomized CPLP algorithm with any convex shaped constraints

```
 1: $C \leftarrow$ set of all constraints
 2: $dest \leftarrow$ the destination point
 3: $feasible \leftarrow true$
 4: $closestYet \leftarrow dest$
 5: $checkedC \leftarrow \varnothing$
 6: for next random $c_1 \in C$, $c_1 \notin checkedC$ do
 7:     if $closestYet$ satisfies $c_1$ then
 8:         $checkedC \leftarrow checkedC \cup \{c_1\}$
 9:         continue
10:     end if
11:     $feasible \leftarrow false$
12:     $point \leftarrow$ closest point of $dest$ on $c_1$
13:     update $closestYet$ with $point$, and set $feasible$ to $true$ if better and satisfies all
    $c \in checkedC$
14:     for $c_2 \in checkedC$, $c_2 \neq c_1$ do
15:         for each $point \in$ {intersection points of $c_1$ and $c_2$'s edges} do
16:             update $closestYet$ with $point$ ... (line 13)
17:         end for
18:     end for
19:     if $feasible = false$ then return
20:     end if
21:     $checkedC \leftarrow checkedC \cup \{c_1\}$
22: end for
23: return $closestYet$, $feasible$
```

Chapter 3

Implementation

I have chosen to implement collision avoidance in Unreal Engine (developed by Epic Games). It is a state of the art game engine with many features suitable for indies and pros as well. A project's logic can be written in two ways: C++, and/or blueprints (the two can be combined). Blueprint visual scripting interface is a tool that allows quick prototyping, but entire games can also be written by it.

For obvious performance reasons, I chose C++ to implement collision avoidance. I did use blueprints for level editing, and creating the test scenarios, but the important parts are all in C++.

For each unit computing the constraints induced by others, and solving the optimization problem could be done parallel, but for simplicity I chose to implement everything single threadedly. The program could easily be ported to multiple threads though.

3.1 Avoidance Component

I succeeded to create an attachable component (ActorComponent) that has several properties:

- radius - the radius to use in collision avoidance, it can make sense to use a bigger radius than the actual physics collider
- maximum velocity - the maximum velocity of the unit
- maximum acceleration - the maximum acceleration of the unit
- current target - this is a 2d location where the unit aims to go
- acceptance distance - if the unit is closer to the target than the acceptance distance, it's considered to have reached its target, so its preferred velocity will be 0
- slowdown distance - if the unit is farther to the target than the slowdown distance, its preferred velocity will point towards the current target, and its magnitude will be the unit's maximum velocity. If the unit is closer than this distance, it will decrease its preferred velocity gradually to 0 until it reaches the target (acceptance distance)

The avoidance component is able to calculate its preferred speed according to the the properties above, and it can set the new velocity of its owner unit when calculating the avoidance maneuvers are done.

3.2 ORCA Manager

ORCA Manager handles the avoidance components that register to it. It queries agents for their preferred speed, it uses a kd-tree for nearest neighbour computations, an ORCA Solver for computing new velocities, and sets the new velocities after simulation.

3.3 ORCA Solver

ORCA Solver stores the agent data, and computes new velocities for every agent by computing the constraints, and adding them to the CPLP Solver.

3.4 Agent

An agent is a minimal representation of a unit for computing the avoidance maneuvers. It has a position, a velocity, a preferred velocity, a radius, maximum velocity, and maximum acceleration magnitude constraints. It also stores the nearby agent ids that the kd-tree computed.

3.5 CPLP Solver

CPLP Solver solves the optimization problem of selecting a velocity close the the preferred velocity in the available convex region permitted by ORCA (linear), and circle constraints.

The randomized algorithm mentioned in Chapter 2 works by maintaining its current best solution, and some side computation when a new constraint violates it. This means the algorithm will work faster if the constraints that define (an intersection of constraints, etc.) the optimal point are found sooner. It follows that the acceleration circular constraint being as small as it usually is, needs to be checked first, since it will almost always have something to do with the optimum. The maximum velocity circular constraint, if not intersecting the acceleration constraint, can be left out completely. Including these two insights into the solution the following modifications were made to the randomized algorithm (Algorithm 3):

- constraints can be marked as fixed - meaning they will not be among the constraints that are randomized, and they will be in the beginning of the order of the constraint elements in the order they are added
- when adding a circle constraint, it is checked if it is completely inside another circle constraint (in which case it can replace the other), or if it has a circle completely inside it (in this case it can be ignored)

3.6 Math library: MathUtils

The CPLP Solver program code boils down to lots of mathematics. The core of it was my MathUtils library which contains mostly geometric functions for calculating intersections.

It was very important that these functions work well without flaw. Bugs can cause NaN numbers emerging in the ORCA constraints, and as a consequence in the velocities. Eliminating these bugs is crucial, as they may cause performance issues, and glitches. An effort must be made to handle the special cases for the geometric functions, like the orthogonal projection of a point to a circle when the point is at the center of the circle.

When implementing a math library, make sure to handle every possible case even if it's possibility is close to 0.

3.7 KD Tree

Any one unit could induce an ORCA constraint in another unit, this means that for n units, there can be $n(n-1)$ ORCA constraints that need to be calculated. However, it is unnecessary to count for ORCA constraints induced by far away units. For this purpose, a kd-tree can be used, which can be built in $O(n \cdot log(n))$ time. After the tree is built, it allows searching for nearby units in $O(log(n))$ time, so setting each unit's nearby units still only takes $O(n \cdot log(n))$ time instead of $O(n^2)$. If the nearby units are known (c of them), then only $O(c \cdot n)$ ORCA constraints need to be computed.

3.8 SVG exporter

There were several times where a strange behavior needed debugging, and it usually took a long time reproducing them, and finding the source of it. To facilitate debugging, I created an svg exporter which writes program state out to svg files to visualize units, their positions, and their constraints. This significantly reduced the time needed to understand what the problem was when inspecting a strange behavior, and helped a great deal to fix the problem.

Chapter 4

Test Scenarios

All the tests were run on an Intel i5 3570k CPU, with 16 GB RAM, running Windows 7.

4.1 Circle-n Test

The most commonly used test scenario for collision avoidance is the Circle-n test. In this test there are n units on the perimeter of a circle, and they have to move to their antipodal position while avoiding collisions with each other.

In the first case I tested my implementation for up to 1000 units. Some screenshots can be seen in Figure 4.1. For such large numbers the units usually get crowded in the middle, and then slowly the units on the edge break away from the rest, until the core is no more. This test was done using a simple pawn unit which looks like a cylinder with a blue material, has a capsule collider (for physics interaction), and a floating pawn movement component.

I measured computation step times, deltatimes between frames, fps, minimum fps, and the time to complete the cirlce-test (every unit reaches its destination) for various unit numbers. The measured data can be read in this table:

Unit count	50	100	200	400	600	800	1000
Avg computation per step (ms)	0,32	0,55	1,01	2,25	3,25	4,36	5,45
Avg deltatime between frames (ms)	16,13	16,13	16,13	16,29	29,44	23,78	28,29
Avg frames per second	62	62	62	61,39	51,44	42,05	35,35
Minimum frame per second	61,9	61,97	57,98	45,63	29,35	20,84	17,97
Time to complete test (sec)	12	20	30	61	92	125	174

On the charts I will refer to my solution as ORCA ext, since I consider it as an extension of ORCA.

I compared my solution's time to complete with unreal engine's integrated RVO solution's time in Figure 4.2. It can be seen that the two solutions do not deviate much for lower unit

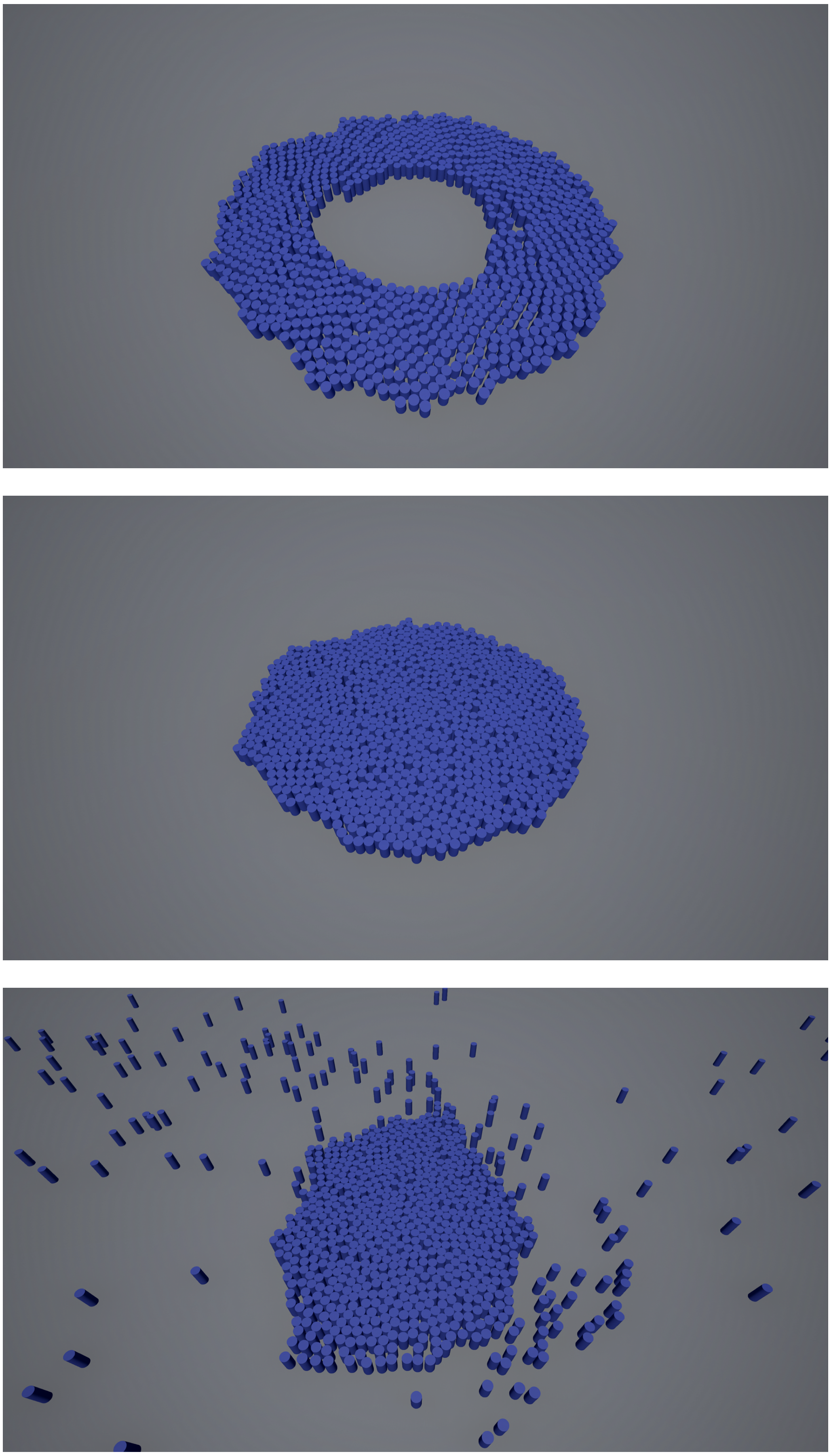

Figure 4.1: Circle with 1000 units moving to their opposite positions across

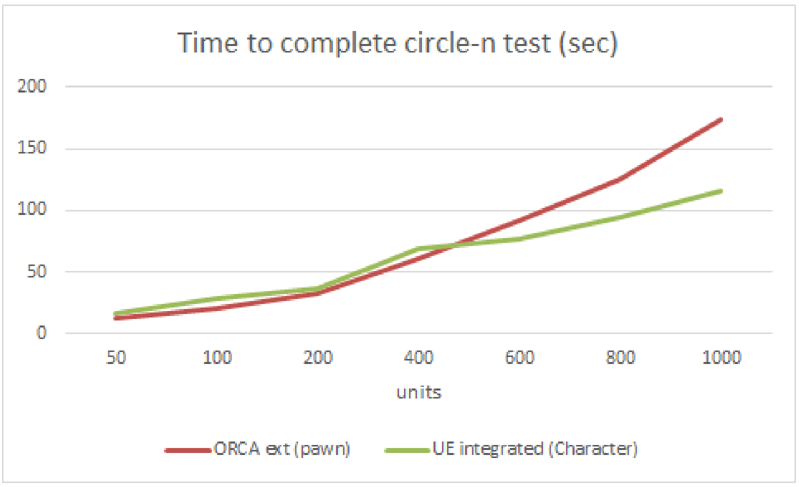

Figure 4.2: Time to complete circle-n test

numbers, but for larger numbers (> 500) the integrated solution clearly finishes sooner. When watching the simulation, the integrated solution's behavior did seem more natural.

It must be noted, that the engine currently only supports collision avoidance for Characters (humanoid-style Pawn with human-like movement). I tried my attachable component on a Character, but it only works well for low number of units (max. 50), for more units, the units that get close to each other in the center get stuck easily. So the integrated solution works for Characters, and my solution works well for Pawns. Characters are able to do much more than Pawns, and their movement capabilities can be quite different than that of simple Pawns. Characters benefit from their more complex components, they are able to find a path in the environment using the navmesh.

4.2 Narrow Passage

Narrow passage (also known as blocks) is a test where four groups (each having 5×5 units) have to move to their mirrored positions from the center, where four square shaped blocks (static obstacles) are in the way. For static obstacles the units have to take full responsibility for avoiding collision (the smallest change vector: u should not be halved, see in Chapter 2).

Screenshots can be seen in Figure 4.3 and 4.4. I tried this test without any obstacle support, but the units obviously got stuck in the middle. I also tried to query the engine for a path to the destination so that units may consider the blocks at least in their preferred velocities, but still it was too crowded in the middle. Finally I succeeded when I added obstacle support. In my solution, obstacles are just like units, but they don't move (and can't be seen). I placed obstacles on the edge of the blocks to approximate the sharp edge of the blocks by the circle shaped obstacles. This method worked really well, but it did require quite a lot of obstacles, when in fact it can be done with just a few line segments (Section 5.4 in Berg et al. [9]).

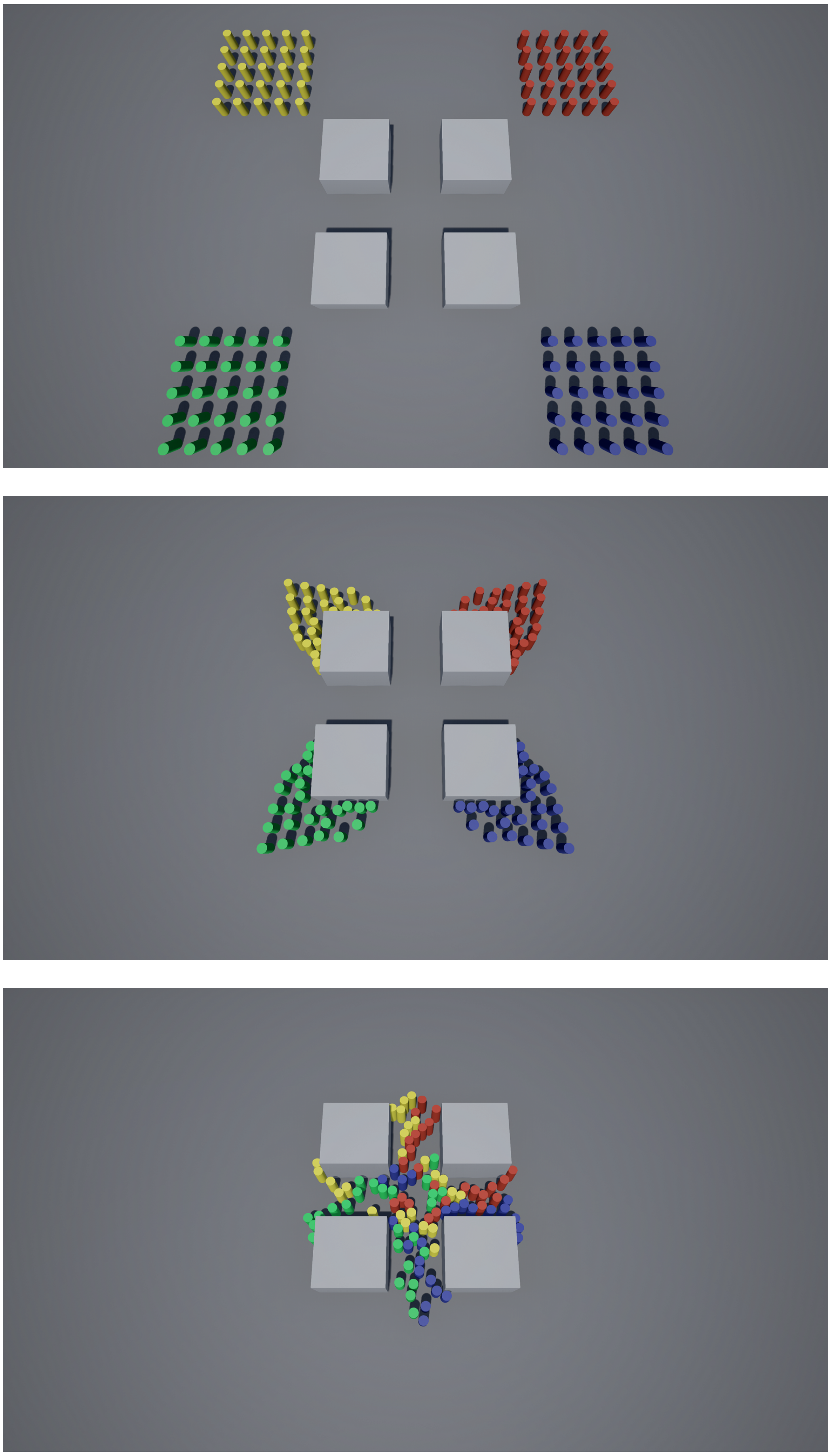
Figure 4.3: Narrow passage test 1/2

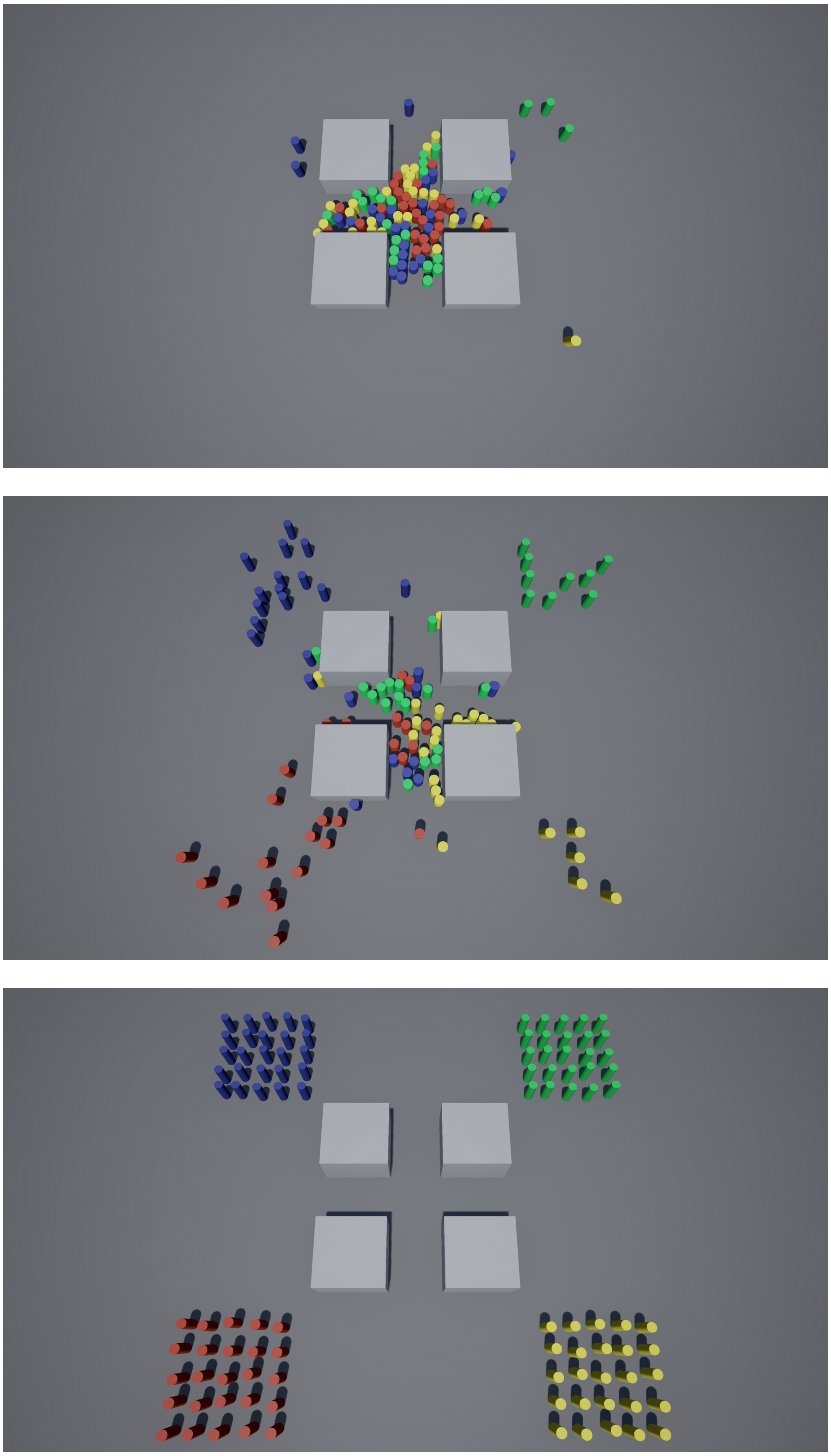
Figure 4.4: Narrow passage test 2/2

Chapter 5

Scalability

I have measured the milliseconds it took to simulate collision avoidance maneuvers for the Circle-n test in my solution. I have compared my results with the results in Snape et al.[4], the comparison can be viewed in Figure 5.1. Keep in mind, that my solution works on only one thread, while RVO2 is parallelized with OpenMP.

It can be seen that my solution scales linearly with the number of agents (n). RVO2 is also linear (although that may be not clear from the figure), but it is at least an order of magnitude faster than my solution. For 1000 units, it could simulate the step in an average of $240.7\mu s$.

I have also compared how the unreal engine integrated RVO compares with my solution in terms of delta times between frames (the inverse of FPS). This comparison can be seen in Figure 5.2. This of course might not mean much, since my solution can only use basic pawns, and the UE integrated RVO uses characters with skeletal mesh components which might easily require more performance to animate and render, than their RVO simulation.

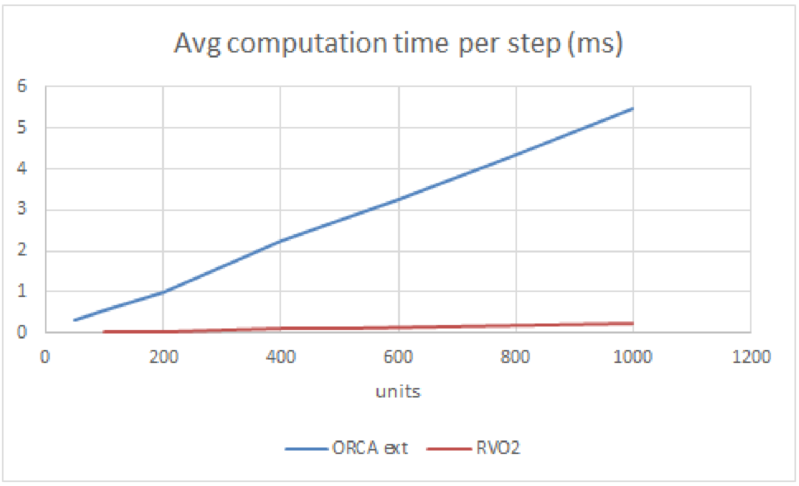

Figure 5.1: Average computation time per step (in ms)

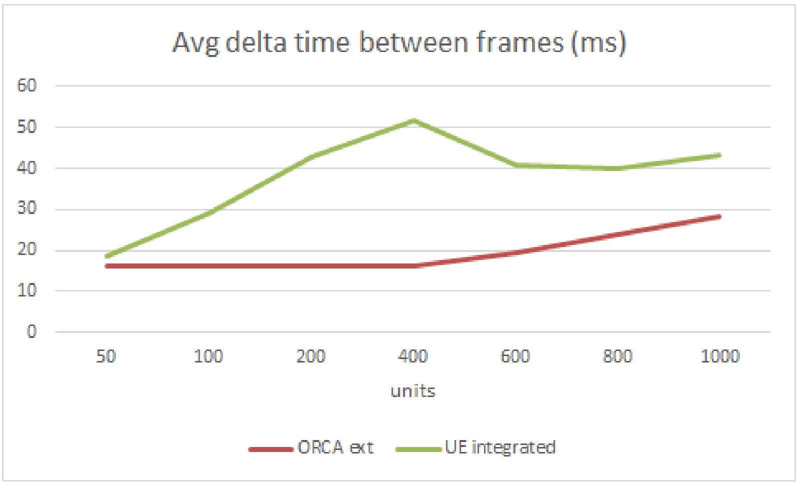

Figure 5.2: Average delta time between frames in millis

Chapter 6

Conclusion

I have extensively studied the currently available realtime collision avoidance solutions, and found ORCA to be one of the best methods. I have implemented my own ORCA solution based on articles about it from zero, gained a keener insight into low dimensional convex optimization problems while doing so, and have successfully extended it to be able to use circular constraints in addition to linear ones. I have also presented a pseudo algorithm for handling any kind of convex shape instead of, or in addition to circles. This circle addition allowed me to consider a unit's maximum velocity and acceleration constraints while calculating the avoidance maneuver.

I have adapted my ORCA extension with circles in an actual game engine proving its feasability with the available computing power of today's users with interactive framerates. The performance shows promising results for further development, and the visual experience is pleasing. I was able to implement a much better solution for collision avoidance than my previous effort using Reynolds' flocking behavior, and I dare say a valuable addition to the integrated solution in the current version of the engine (4.9.2).

My solution has the following advantages compared to the integrated solution:

- better scalability (for the measured range)
- it takes acceleration constraints into account in the maneuver calculation (although it is unclear for me, whether the UE solution handles acceleration, or not, and how - from what I've seen in it, it locks the calculated speed for a short time)
- mostly independent from the engine - the solution could be ported to another game-engine or a standalone application quite easily

Possible disadvantages:

- takes more time to resolve the circle-n scenario
- it does not work (well) for Characters (walking humanoid pawns)
- worse integration with the engine, and its features (the engine is huge, and there are a number of things my solution ignores, like static obstacles, or the navmesh) - in an ideal integration it wouldn't be necessary to set many of the properties seen in my Avoidance Component, the RVO integrated avoidance is just a checkbox in CharacterMovementComponent

For future work I propose optimizing my solution, since the authors of ORCA succeeded in a much faster algorithm, and my extension does not increase the overall complexity. Although their library may be parallel, I'd like to optimize it first on a single thread before parallelizing. Later, I'd like to implement obstacle support with line segments (this was done in Berg et. al[9]), and pathfinding including traffic calculations to avoid high unit density where the solution seems to show its weak side. (The units move in towards the center even though it should be clear that it's better to go around the crowded core of units).

I'd also like to explore other real time methods, especially ones for dense scenarios (crowd simulation). An interesting method which works well for dense scenarios as well is Continuum Crowds [11]. To rough it out, it combines pathfinding with collision avoidance by calculating grid tiles, and where the units average velocity points in them, then pathfinding takes these grid directions into account when finding a path.

It would be also interesting to see the new integrated collision avoidance in Unreal Engine, and if it has any way to handle accelerations. As far as I know, for the next version (4.10) they are planning to integrate or implement the RVO2 library, which is the official ORCA solution developed by the authors of the ORCA publications [12].

My final thoughts are that Unreal Engine did help a lot to develop all that I accomplished to do, and for developing games it is really worth using it, but when trying to create tests, a lot of work goes into creating not only those tests, but other aspects of navigation that are not the subject of the paper, but needed nevertheless. In addition comparing the results of those tests with another solution's results and fairly compare them can be a difficult task.

Addressing these problems, not long ago I found Menge[13], a crowd simulation framework that has all the usual tests (called component subproblems), and it helps researchers focus on the single task at hand instead of having to implement a crowd simulator. The other aspects of navigation are handled by the framework instead. For papers on movements of crowds, and related fields, it could prove to be worthwhile getting to know such a system, and implementing a solution that can be plugged into that system.

Acknowledgements

I would like to express my thanks to my supervisor dr. Blázovics László for his encouragement, and his support for making this work happen, and for setting me on this path.

This work has been created for the Students' Scientific Conference 2015 of the Budapest University of Technology and Economics.

Bibliography

[1] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, pp. 25–34, August 1987.

[2] C. W. Reynolds, "Steering behaviors for autonomous characters," in *Game developers conference*, vol. 1999, pp. 763–782, 1999.

[3] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *The International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.

[4] J. Snape, S. J. Guy, D. Vembar, A. Lake, M. C. Lin, and D. Manocha11, "Reciprocal collision avoidance and navigation for video games," in *Game Developers Conf., San Francisco*, 2012.

[5] J. Van den Berg, M. Lin, and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 1928–1935, IEEE, 2008.

[6] J. Snape, J. Van den Berg, S. J. Guy, and D. Manocha, "Independent navigation of multiple mobile robots with hybrid reciprocal velocity obstacles," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pp. 5917–5922, IEEE, 2009.

[7] J. Snape, J. van den Berg, S. J. Guy, and D. Manocha, "The hybrid reciprocal velocity obstacle," *Robotics, IEEE Transactions on*, vol. 27, no. 4, pp. 696–706, 2011.

[8] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, and P. Dubey, "Clearpath: highly parallel collision avoidance for multi-agent simulation," in *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 177–187, ACM, 2009.

[9] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics research*, pp. 3–19, Springer, 2011.

[10] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications.* Santa Clara, CA, USA: Springer-Verlag TELOS, 3rd ed. ed., 2008.

[11] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 1160–1168, 2006.

[12] J. van den Berg, S. J. Guy, J. Snape, M. C. Lin, and D. Manocha, "Rvo2 library: Reciprocal collision avoidance for real-time multi-agent simulation," 2011.

[13] S. Curtis, A. Best, and D. Manocha, "Menge: A modular framework for simulating crowd movement," *University of North Carolina at Chapel Hill, Tech. Rep*, 2014.