Social Navigation in Unity 3D

by

Karl Valdimar Kristinsson

Project report submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in Software Engineering

June 2015

Project Report Committee:

Hannes Högni Vilhjálmsson, Supervisor
Ph.D., Assistant Professor, School of Computer Science, Reykjavik University

J. Deon Garrett
Ph.D., Assistant Professor, School of Computer Science, Reykjavik University

Stephan Schiffel
Dr. rer. nat., Assistant Professor, School of Computer Science, Reykjavik University
Copyright
Karl Valdimar Kristinsson
June 2015
Social Navigation in Unity 3D

Karl Valdimar Kristinsson

June 2015

Abstract

Virtual environments such as those experienced in games are increasingly more common. Often these environments are populated by virtual characters that serve various purposes. In almost all instances they bring the environment to life, giving the user a deeper experience. These characters spend a lot of time moving around, either to achieve some goals or to simply give the environment an authentic feel of an active crowd. To maintain the illusion of life, how a character navigates an environment, full of static and dynamic obstacles, needs to reflect human intelligence or at least meet people’s expectations of human-like behavior. With many different navigation techniques available this project focused on implementing one that produces socially believable behavior. The chosen approach, a velocity-based method, aims to have characters make small adjustments long in advance before colliding with obstacles. This method was implemented within the Unity 3D game engine and extended to support more dynamic testing and use. The method was furthermore integrated into CADIA’s existing social simulation platform. Several social scenarios were used to stress test the implementation, resulting in a thorough review of strengths and weaknesses, compared to Unity’s own default navigation system. The results demonstrate the superiority of the method in several important scenarios but also point out an important weakness that emerges in very dense crowds.
Félagslegt flakk í Unity 3D
Karl Valdimar Kristinsson
Júní 2015

Útdráttur

Contents

List of Figures vii
List of Tables viii

1 Introduction 1

2 Background 3
  2.1 Navigation .......................................................... 3
  2.2 Social Navigation .................................................. 4
  2.3 SECOM Projects ................................................... 5
    2.3.1 Virtual Reykjavik .......................................... 5
    2.3.2 Impulsion .................................................... 5
  2.4 Unity Game Engine .............................................. 6
    2.4.1 Unity Game Objects and Components ...................... 6
    2.4.2 Path Finding in Unity ..................................... 6
    2.4.3 Locomotion in Unity ....................................... 8
    2.4.4 Unity Animator ........................................... 9
    2.4.5 Feeding the Unity Animator ............................... 10
  2.5 Terminology .................................................... 10

3 The Social Navigation System 13
  3.1 The Avoidance Algorithm ..................................... 14
    3.1.1 Step 1: Collisions ....................................... 14
    3.1.2 Step 2: Admissible Velocities ............................ 15
    3.1.3 Step 3: The Best Velocity ................................ 18
  3.2 Shortcomings .................................................. 19
    3.2.1 Intersections ............................................. 19
    3.2.2 Animator Problems ....................................... 20
  3.3 Optimizations ................................................ 21
3.3.1 Nav Resolution ........................................ 22
3.3.2 In Range ................................................. 22
3.4 Debugging .................................................. 23
  3.4.1 Navigation Path ....................................... 23
  3.4.2 Different Velocities .................................... 24
  3.4.3 Collision ............................................... 24
  3.4.4 Intersections .......................................... 25
  3.4.5 Deviations ............................................. 25
3.5 Integration with Impulsion .................................. 25

4 Experiments .................................................. 29
  4.1 Doorway .................................................... 30
    4.1.1 One on One ........................................... 30
    4.1.2 Three on Three ...................................... 30
  4.2 Small Corridor ........................................... 31
    4.2.1 One on One ........................................... 31
    4.2.2 Six on Six ............................................ 32
  4.3 Medium Corridor ......................................... 32
    4.3.1 One on One ........................................... 33
    4.3.2 Twelve on Twelve ..................................... 34
    4.3.3 Overtake ............................................... 34
  4.4 Large Corridor ........................................... 35
    4.4.1 Three on Three ...................................... 35
    4.4.2 Twelve on Twelve ..................................... 36
  4.5 Austurvöllur ............................................... 37

5 Conclusions .................................................. 39
  5.1 Future Work ............................................... 39
    5.1.1 2D to 3D ............................................... 39
    5.1.2 Social Group Awareness ................................ 40
    5.1.3 Intersections .......................................... 41
    5.1.4 Flow Adjustment ...................................... 41

Bibliography .................................................. 43
List of Figures

2.1 Unity Nav Mesh ................................................. 7
2.2 Unity Nav Mesh Agent ......................................... 8
2.3 Unity Animator ................................................ 9

3.1 Navigation System Flow ........................................ 13
3.2 Navigation Velocities ......................................... 14
3.3 Angle Deviation .............................................. 16
3.4 Unity Animation Curve ........................................ 16
3.5 Navigation Path ............................................... 23
3.6 Different Velocities Example .................................. 24
3.7 Intersections .................................................. 25
3.8 Deviations ..................................................... 26
3.9 Impulsion Behavior Tree ....................................... 27

4.1 Doorway, One on One ......................................... 30
4.2 Doorway, Three on Three ..................................... 31
4.3 Small Corridor, One on One ................................. 32
4.4 Small Corridor, Six on Six ................................... 33
4.5 Medium Corridor, One on One ............................... 33
4.6 Medium Corridor, One on One, Social Space ............. 34
4.7 Medium Corridor, Twelve on Twelve ....................... 35
4.8 Medium Corridor, Overtake .................................. 35
4.9 Large Corridor, Three on Three ............................ 36
4.10 Large Corridor, Twelve on Twelve, Initial Positions .... 36
4.11 Large Corridor, Twelve on Twelve ......................... 37
4.12 Austurvöllur .................................................. 38

5.1 Social Group Awareness ....................................... 41
5.2 Flow Adjustment ............................................... 42
List of Tables

3.1 Angle Deviation Default Values ........................................ 17
3.2 Velocity Ratings Constants Default Values .......................... 19
Chapter 1

Introduction

Virtual environments are becoming more and more popular and can be used for various purposes. Their best known use may be computer games but there are many more applications for them such as for training, learning and planning. A great deal of these virtual environments have some human characters. The role of those characters can differ greatly just like the environments that they belong in. For example they might be enemies of the player charging at him or running away from him. They might be his allies or someone they have to gather resources or information from. In most cases those characters are expected to behave in a lifelike manner. Characters can exhibit many different behaviors and one of them is how they navigate the environment. In this project the navigation behavior is brought as close to a lifelike performance as possible.

To produce a lifelike or human like character isn’t just about high quality graphics or appearances. Other factor can quickly make or break our illusion. This can be easily verified by imaging a highly appealing synthetic face and hearing it talk by only making O-shapes with its mouth. That perfectly built face is quickly broken by its unnatural movements. In fact a much worse looking face with better coordinated mouth flapping will look more appealing than the perfect face with unnatural mouth movements. The essence is that we find familiar movements and behaviors more appealing than exceptionally good graphics (Mori, MacDorman, & Kageki, 2012). From that notion this project aims to create a navigation behavior that resembles the way we walk around in our daily life.

The readers are encouraged to imagine themselves in situations that stress the navigation system. For example when walking towards someone head on, you don’t walk up to them until you’re a few centimeters away and then make a 90 degree turn to avoid them. Instead you make a small adjustment early on and miss them by a distance that you find acceptable, some like to keep a large distance while others don’t mind breezing up
against strangers. This is the main goal of the Social Navigation System, to make small adjustments early on instead of large changes just before colliding with someone.

The author is a member of the research group called Socially Expressive Computing (SECOM) within the Center of Analysis and Design of Intelligent Agents (CADIA) in Reykjavik University. The goal of SECOM is to create socially believable models and simulations. There are already some social simulation projects in development at SECOM such as the Icelandic Language and Culture Training in Virtual Reykjavik and the Impulsion project. The social navigation system is designed to be able to interact with these existing projects.

The structure of the report is fairly straight forward. This chapter gave an overview of the intended goals and what has to be kept in mind when reading the report. In the next chapter the work of others is examined, necessary background information given and overview of used tools provided. In chapter 3 the entire inner workings of the Social Navigation System is revealed and design notes are covered. In chapter 4 the system is put to the test and several cases are examined, flushing out the systems strong sides and weaknesses. Finally in chapter 5 the project is summed up and some future work is covered.
Chapter 2

Background

Navigation is a broad subject, the beginning of this chapter covers the work of others and how that affects the goals of this project. After that the necessary terminology is covered along with an extensive look at the Unity game engine sometimes referred to as Unity3D.

2.1 Navigation

There are many forms of navigation available and this project is about social navigation. Given a virtual environment filled with characters, every character should be able to find his way from one location to another without colliding with the environment and other characters. To make it a social navigation system real care has to be taken when avoiding other characters. Although there are many navigation or collision avoidance algorithms out there, some very fast ones, they might not exhibit a very socially believable behavior.

When referring to navigation or steering Reynolds work on Boids is widely cited. He gave virtual birds a set of simple steering rules to steer through an environment filled with other birds. This essentially created very lifelike flocking behaviors (Reynolds, 1987). In his next iteration, Reynolds described many more steering behaviors and added the definition of path following using his steering behaviors (Reynolds, 1999). His work on its own isn’t good enough to create a social navigation system but a lot of the concepts to be used are based on his work. The most important one is path following, with a known path from one location to another a character has to be able to follow that path adequately.
Following a path isn’t enough when navigating a dynamic environment filled with other independent characters. A well known solution to dynamic collision avoidance in robotics is using velocity obstacle (VO) (Fiorini & Shiller, 1998). In VO each character is able to observe the velocities of other characters. Every character can then check against every other character to see whether or not they will collide if they both maintain their current velocities. If a collision is imminent a character can find another velocity that won’t resolve in a collision. Many improvements have been done on VO, most notably the Reciprocal Velocity Obstacle (RVO) (Van den Berg, Lin, & Manocha, 2008). There characters assume that other characters are going to make similar collision-avoidance reasoning. This approach has been optimized by only solving the problem as a low-dimensional linear program. In fact Unity uses the RVO approach for its navigation system but as covered in the Experiments chapter RVO doesn’t provide very socially believable behaviors.

### 2.2 Social Navigation

In the previous section some of successful navigation techniques were mentioned but they tend to miss the social aspect. It is possible to use the VO method and add some sort of rating of the velocities. So when a character detects a collision with another character, instead of choosing the first collision free velocity available, a set of possible velocities are considered. Each of those velocities are then given a rating by some formula and the best one is chosen. The rating formula can be almost anything and it can give very diverse behaviors.

In (Karamouzas & Overmars, 2010) such a rating approach is used. They also define a very interesting way as to how and when alternative velocities should be considered. This is in fact the paper that this project is based on. The proposed model creates very socially believable behaviors and could easily be integrated into the Unity game engine. This report thoroughly describe how that model was used to create the Social Navigation System using Unity.

Many other navigation techniques are available, and were considered for this project. In (Duives, Daamen, & Hoogendoorn, 2013) an overview of different crowd motion models can be observed. All the models are categorized into categories such as Cellular Automata, Social Force models, Continuum models and Velocity Based models; see (Duives et al., 2013) for an extensive list. Each of these categories uses a different technique to achieve crowd simulation. In (Blue & Adler, 2001) they use Cellular Automata microsimulation for modeling pedestrian walkways. Social force models such as (Helbing
& Molnar, 1995) use social force fields to guide pedestrians through environments and avoid one and another. Models based on continuum dynamics such as (Treuille, Cooper, & Popović, 2006) can be used to create large crowds without the need for explicit collision avoidance. Many of these models produce socially believable behaviors but the velocity based model by (Karamouzas & Overmars, 2010) provided the best balance between good results and ease of implementation, as will be covered in this report.

It’s important to mention that (Karamouzas & Overmars, 2010) has been implemented before within SECOM (Oliva & Vilhjálmssson, 2014). That was done on a different platform, the Panda3D game engine and without the Impulsion behavior tree integration that this project strives for.

### 2.3 SECOM Projects

The social navigation system is intended to be used within two additional projects at SECOM. Firstly is Virtual Reykjavik, described in section 2.3.1 and secondly as mentioned before the social navigation system is expected to integrate fluently with the Impulsion project, described in section 2.3.2.

#### 2.3.1 Virtual Reykjavik

Virtual Reykjavik takes place at Austurvöllur in Reykjavik. There players can explore downtown Reykjavik and interact with the Icelandic people there. The people are of course computer controlled characters but a lot of work has gone into making them as human like as possible. The goal of the project is to allow foreign individuals to experience the atmosphere of downtown Reykjavik and learn some basic Icelandic before coming physically to Iceland. The social navigation system is to be used to guide the computer controlled characters between places giving the player a feel of busy downtown Reykjavik.

#### 2.3.2 Impulsion

Impulsion is mainly the work of Claudio Pedica (Pedica & Vilhjálmssson, 2010). It brings computer controlled characters to life by giving them spontaneous and human territorial behaviors. Impulsion has been implemented using the Unity game engine and state of the art behavior trees (Pedica & Vilhjálmssson, 2012). In Impulsion characters can form
groups and interact with one another. Characters are also very observant about their environment, gazing at other characters and so forth. The social navigation system is supposed to be integrated flawlessly with Impulsion so that characters being navigated through the environment are observant of one and another.

2.4 Unity Game Engine

A game engine is a computer program that offers users to create emergent virtual environments. Most games are created using some sort of a game engine, although the line between a game and a game engine can be very thin at times. Therefore many game engines are created for a specific genre of games. SECOM uses the free Unity game engine to deliver its virtual experiences. Unity hasn’t been created with any one genre of games in mind making it flexible in many situations. The social navigation system is developed in the Unity game engine. This gives the system a lot of flexibility, it can be used by almost all projects within SECOM. The system is also designed to be only dependent on core features of Unity, so that anyone using Unity should be able to use this implementation. In this section these core features will be covered, giving the reader an insight into how they function.

2.4.1 Unity Game Objects and Components

The core of any 3D program are objects that occupy the three dimensional space. These objects are twofold in Unity, a game object is a point in space and it can have any number of components. So an empty game object, with no components is just a point in space. To create a box, a game object with a box rendering component is needed. There are large numbers of components available, from all sorts of rendering shapes to a rigid body component to associate an object with Unity’s physics engine. User created components are also possible, such as scripts.

2.4.2 Path Finding in Unity

The core of the navigation system is path following but in order to perform path following there must be a path to follow. Path finding finds a path from one location to another without colliding with the static environment. The aim of this project is not to perfect a path finding algorithm. Unity has a fairly powerful path finding solution and there was no
need to look any further. The solution in Unity is two fold, the Nav Mesh which indicates
the walkable area in the environment and the Nav Mesh Agent to navigate through the
Nav Mesh.

Unity Nav Mesh

The Nav Mesh is baked prior to running a level in Unity and is associated with each level.
It uses all objects that have been marked as Nav Mesh static and as that implies the objects
should be unmovable. It is possible to define movable object within the Unity Nav system
but that is not within the scope of this project. An example of a Nav Mesh can be seen in
figure 2.1.

![Figure 2.1: An example of the Unity Nav Mesh. The floor is white (1) and the walls are
red (2). The blue area (3) is the walkable Nav Mesh.](image)

Unity Nav Mesh Agent

A Nav Mesh Agent is a component and is essentially just a cylinder that can be placed
on a Nav Mesh and used to navigate through the environment. In figure 2.2 a Nav Mesh
Agent can be observed in its simplest form. The path finding happens within the agent,
once it has been placed on the Nav Mesh it can be given a target to navigate to. It will
then find the shortest path and start navigating to that target. When a Nav Mesh Agent
component is placed on a simple game object, such as a box, it can navigate it using
a series of movements and rotations to bring it to its destination. In the case of more
complex objects, such as a character that isn’t good enough, since we want the character
to exhibit animations and have the animation control the displacement. How this is done
will be covered in section 2.4.3 about locomotion.
2.4.3 Locomotion in Unity

Locomotion is a change in position and is straightforward for simple objects such as spheres or boxes, which can just be moved. But when animating a character we want the animation to control the motion. For example, walking forward is an animation where we expect the character to move further along its forward direction. The animation is usually loopable, meaning it can be played indefinitely many times without looking unreal. It’s therefore possible to create an animation for walking forward and apply it to a character forever and he’ll just keep going forward. In the animation the walking speed might correspond to a constant speed, for example 1m/s. But what will happen if the character should move at a faster pace? There could be a different animation, but then if the character should move slower, does that require another animation as well? Instead of having different animations it is also possible to just slow down or increase the playback speed of the original animation, but as the speed increases dramatically and the character starts running the movement isn’t the same as when walking. This is in fact a well researched problem and the best solution so far is using motion blending, where different animations are blended together (Perlin, 1995). For example when walking one might blend together a standing idle animation with a walking one. As the weight of the walking animations increases the character walks faster. Once he’s reach the maximum walking speed the next step would be to blend the walking animation with a running animation. Fortunately all of this is very simple within Unity.
2.4.4 Unity Animator

Unity’s Animator is a tool to make motion blending as simple as possible. Different animations can be graphically placed on a sketch and blending parameters specified. In figure 2.3 a snapshot of the animator for this project can be found. It’s actually the same one as used for Impulsion but any animator could be used given that there is a good correlation between the input and output speed. The correlation is important because if the character wants to move a little faster the animator must be able to match the request. This correlation is described thoroughly in section 3.2.2.

There are two input parameters on the visible part of the animator in figure 2.3. Those are angular speed (Angular) and the forward speed (ZSpeed). The forward speed determines the weight between standing still, walking and running. The angular speed is to change the direction of the characters. There are different turning animations for turning either left or right and those must exist for both running and walking. There are also different variations as to how fast the angular change is executed, short, medium and wide. This results in quite many animations, not nearly all are visible in figure 2.3. Unity is smart enough to identify when to use each animation, that is, Unity automatically generates the thresholds as to when every animation is activated, this can be tweaked afterwards if required.

![Figure 2.3: Many animations can be easily blended together using Unity’s Animator. The box on the left is the blending unit and the boxes on the right represent different animations.](image-url)
2.4.5 Feeding the Unity Animator

When using the Unity Nav Mesh Agent from section 2.4.2 on a simple object such as a box the agent can just move the object but when using the animator an extra locomotion step is needed. Instead of allowing the agent to move the object, in this case a character, the agent can output its desired velocity. This velocity can then be broken down into the required parameters for the animator. The forward speed is the magnitude of the velocity and the angular speed is the angle between the current orientation of the character and the desired velocity.

Unity does provide collision avoidance for its Nav Mesh Agents, in fact it uses RVO (Van Den Berg, Guy, Lin, & Manocha, 2011) as mentioned earlier. RVO however tends to not look very socially believable and behaves very badly when coupled with an Animator. The goal of this social navigation system is to take the desired velocity from the Nav Mesh Agent and apply additional collision avoidance to get socially believable navigation from the characters.

2.5 Terminology

Following is a list of terms that the reader is expected to know and understand throughout the report. Most of them should have been covered already but this can be used to lookup when in doubt.

Social Navigation System The system described in this report, often abbreviated as SNS.

Animator The component within the Unity game engine that transforms the velocity from vector to animations to move a character.

Nav Mesh A mesh in the Unity game engine that marks the static environment and allows Nav Mesh Agents to perform path finding.

Nav Mesh Agent A component within the Unity game engine to perform path finding and navigate through the environment.

Game Object A point in 3D space in Unity which can have any amount of components attached to it.

Component Something that can be added to a game object in Unity, can be a verity of things.

Path Finding The operation of finding a path from one location to another.
**Path Following**  The operation of following a given path from one location to another.

**Collision Avoidance**  Something that is done in order to avoid an imminent collision.

**Collision Detection**  A calculation that is done to find a possible collision.
Chapter 3

The Social Navigation System

At this point all necessary features of Unity have been covered. This chapter will cover how those features make up the Social Navigation System. In figure 3.1 a very simple flow of the system can be observed. Unity’s Nav Mesh Agent is used for path finding. Usually the information from the Nav Mesh Agent is sent straight to the locomotion part where it is converted so that the animator understands it. Instead that information is passed to the Social Navigation System where some extra social collision avoidance is applied. This chapter will cover in detail what happens in the Social Navigation part.

Figure 3.1: The basic flow of the system.

This flow will result in three different velocities at any given time. The first one is the desired velocity of the Nav Mesh Agent. Second is a desired velocity from the Social
Navigation System that exists if the Nav Mesh Agent velocity results in a collision. Finally there is the actual velocity of the character. Figure 3.2 shows the velocities. The colors are the same as when the velocities are displayed in the executable version as described in section 3.4.2.

![Nav Mesh Agent Desired Velocity](image1)

![Navigation Desired Velocity](image2)

![Character Current Velocity](image3)

Figure 3.2: The three different velocities within the Social Navigation System.

### 3.1 The Avoidance Algorithm

This first section describes the collision avoidance algorithm presented in (Karamouzas & Overmars, 2010) and how it has been adopted to the Unity game engine. There are 3 steps in the avoidance algorithm. First all possible collisions with agents must be found. Secondly all admissible velocities must be calculated and finally the best velocity must be chosen. These three steps will each be covered in a separate subsection.

This algorithm is performed separately by every character. How often it is performed is explained later in section 3.3.

#### 3.1.1 Step 1: Collisions

This first step of collision avoidance is to check if there are any collisions to avoid. This is done with a simple velocity obstacle formula observable in equation 3.1. For simplicity the agents move on a 2D plane and therefore the two dimensional radius is enough to check for a collision. The Unity Nav Mesh Agent navigates the agents through the environment so that collision with walls and other static obstacles don’t need to be accounted for.

\[ ||(x_j + v_j t) - (x_i + v_i^{des} t)|| \leq r_j + (r_i + \mu_i) \]  

(3.1)
The environment is filled with \( n \) heterogeneous agents \( A_1, A_2, A_3, \ldots A_n \). An agent \( A_i \) is considered to be colliding with another agent \( A_j \) if their radii \( r_j \) and \( r_i \) intersect. Each agent also has a personal space \( \mu \) that they maintain for themselves but do not have knowledge about one another. In equation 3.1 an agent \( A_i \) is at position \( x_i \) traveling with a desired velocity (Nav Mesh Agent) \( v_i^{des} \) and agent \( A_j \) is at \( x_j \) traveling with a velocity of \( v_j \). Note that each agent only knows the current velocity of other agents but not their desired one. If equation 3.1 is solvable there exists a time \( t_{ij} \) where agent \( A_i \) and \( A_j \) will collide if they maintain their velocities. If \( t_{ij} \geq 0 \), \( A_j \) is inserted into the set of the agents that are on collision course with agent \( A_i \).

The set of collisions is then sorted by the time to the collision. Only the N most imminent collisions are kept and others are discarded. That both reduces calculation time but it is also what happens in real life when we avoid others, keeping track of only the most imminent future collisions. Experiments have showed that \( N = 5 \), most imminent collisions is suffice enough (Karamouzas & Overmars, 2010).

### 3.1.2 Step 2: Admissible Velocities

When a collision has been detected a different velocity has to be picked. In this step all possible velocities are calculated. The main idea is that agents should make small changes to their velocity early to avoid one another instead of making dramatic changes right before a collision. In (Karamouzas & Overmars, 2010) experimental analysis were used to determine the appropriate angle and speed changes an agent might take to avoid a collision. From the set of collisions declared at the end of section 3.1.1 the minimum time to collision can be retrieved. This time is used to determine the maximum angle and speed deviation an agent can take. The closer a collision is the more deviations are allowed. Angle and speed deviations are calculated a bit differently so they are covered separately.

**Angle Deviations**

From the minimum time to collision the maximum orientation deviation can be calculated. That is done with a piecewise function observable in figure 3.3. If a collision is imminent an agent can make a very large deviation but as the time to collision increases only a smaller deviation is allowed.

A piecewise function can be easily programed but Unity offers a better way through curves. In Unity there is a data type called *Animation Curve* that allows a designer to
create and dynamically alter curves, even at run time. In figure 3.4 the angle deviation can be seen as an animation curve in Unity. The curve is created in a range between 0 to 1 and can be scaled using two constant, the maximum angle deviation and the maximum time to collision. Notice that the curve is similar to the angle deviation in figure 3.3 but the x axis has been flipped.

The agents are not supposed to be able to turn around so their maximum angle deviation is set to $\pi/2$. As discussed in (Karamouzas & Overmars, 2010) the experimental analyses showed that average time of the maximum collision time was 4.2s but in the case of an algorithm a larger value yielded better results. The default values can be seen in table 3.1.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_{\text{max}}$</td>
<td>$\pi/2$</td>
</tr>
<tr>
<td>$\delta_{\text{mid}}$</td>
<td>$\pi/6$</td>
</tr>
<tr>
<td>$t_{c_{\text{min}}}$</td>
<td>$2.5s$</td>
</tr>
<tr>
<td>$t_{c_{\text{mid}}}$</td>
<td>$6s$</td>
</tr>
<tr>
<td>$t_{c_{\text{max}}}$</td>
<td>$8s$</td>
</tr>
</tbody>
</table>

Table 3.1: The default values for the angle deviation in figure 3.3 as given in (Karamouzas & Overmars, 2010).

**Speed Deviations**

The speed deviation is done in a very similar way but instead of a piecewise function, it is determined by equation 3.2.

$$U_i = \begin{cases} 
    u & | u \in [0, u_{\text{max}}^i] \\
    u & | u \in [u_{\text{pref}}^i, u_{\text{pref}}^i \pm \Delta u_{\text{max}}^i] \\
    u_{\text{pref}}^i , & \text{if } t_{c_{\text{max}}} < t_c < t_{c_{\text{max}}} \\
    u_{\text{pref}}^i , & \text{if } t_{c_{\text{max}}} < t_c 
\end{cases} \quad (3.2)$$

$u_{\text{max}}^i$ is the maximum speed every agent can achieve set to $2m/s$, $\Delta u_{\text{max}}^i$ is the maximum speed deviation set to $0.4m/s$ and $u_{\text{pref}}^i$ is the speed that the agent prefers to maintain. Each agent can prefer a different speed.

This is set up as an Unity Animation Curve as well where the y position can be either below 0, between 0 and 1 or over 1, imitating all three conditions in equation 3.2.

**Combining Angles and Speed**

Until now both the speed and angles have been considered as a range of values and therefore contain infinitely many feasible velocities. To counter this the angle deviation is discretized with a step of 0.13 radians and the speed set with a step of 0.1$m/s$. Both of these step size can be changed within the Unity environment. With the set of all possible speeds and angles a set of admissible velocities can be constructed. This combined set contains all angle deviations traveling with all possible speeds. The size of the set will therefore be the multitude of the number of angle and speed deviations. In the next step all of these velocities get rated and the best one is picked.
3.1.3 Step 3: The Best Velocity

With the set of admissible velocities from section 3.1.2 the next step is to choose which velocity is the best one to follow. This is done by rating each velocity. The goal of the rating is to find an optimal velocity without deviating dramatically from the original path. The rating equation, equation 3.3 is taken directly from (Karamouzas & Overmars, 2010). In it $V^{cand}$ is taken from the set of all admissible velocities $FAV_i$ explained in section 3.1.2. The best velocity is the one with the lowest rating.

$$v^{new}_i = \arg\min_{v^{cand} \in FAV_i} \left\{ \alpha \left(1 - \frac{\cos(\Delta \phi)}{2}\right) + \frac{\beta \left\|v^{cand}\right\| - \left\|v\right\|}{u_{max}} + \gamma \frac{\left\|v^{cand} - v^{des}_i\right\|}{2u_{\text{max}}} + \delta \frac{t_c}{t_{c\text{max}}} \right\}$$

(3.3)

This equation is quite complicated but has been split into three sections, energy, deviation and collisions. Each section has one part except the energy section which has two parts. The two parts represent the angle and speed energy expenditure. Every part has a constant to level it is weight in the rating, the constants are $\alpha, \beta, \gamma$ and $\delta$. Let’s examine each part individually:

- **Energy Angle.** $\Delta \phi$ is the angle between the agents current velocity and $v^{cand}$. This part favors small or no changes of the agents orientation. Changing the orientation requires energy and should therefore be avoided if possible.

- **Energy Speed.** Here the absolute value of the difference between the candidate and agents current velocity is divided by the maximum speed an agent can have. This part favors small or no changes of the agents speed. Similar to the angle energy, changing the speed requires energy and should therefore be avoided.

- **Deviation.** This part favors small or no changes from the desired velocity. This is to keep the agent from deviating to far from it is desired path.

- **Collisions.** This part favors long time until collision. $t_c$ is the time to collision with this velocity and $t_{c\text{max}}$ is the maximum time to collision. This part is of course to avoid collisions.

When calculating the time to collision in the velocity rating only characters that are on a collision course are checked, instead of all agents such as in section 3.1.1. It’s also
important to not pick a velocity that steers the character into the environment. Usually
the Nav Mesh Agent takes care of avoiding the environment but that is not the case when
the Social Navigation System needs to deviate the character. Therefor the collision time
against the environment is also calculated when rating the velocities. The calculation is
done by querying the Nav Mesh. This for example causes the character to choose the left
side when doing an overtake as described in section 4.3.3 instead of the right one which
leads straight into the environment.

The constants in equation 3.3 can provide very different behaviors. The default constants
from (Karamouzas & Overmars, 2010) can be found in table 3.2. Those constants give a
very smooth looking avoidance. In order to give each character a different look they can
be given varying constants. One way is to give each character the default values with a
random $-10\%$ to $10\%$ deviation.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>5.0</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0.5</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>1.0</td>
</tr>
<tr>
<td>$\delta$</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 3.2: The default values of the constants in equation 3.3 from (Karamouzas & Over-
mars, 2010).

3.2 Shortcomings

The algorithm from section 3.1 works fairly well out of the box. However there are
some shortcomings, both with the algorithm and with the implementation in Unity. This
sections covers those shortcomings and what can be done to fix them.

3.2.1 Intersections

This section covers the biggest weakness of the system. This system isn’t impeccable and
in some instances the best velocity can be the one through another character. This happens
especially in very crowded environments; imagine being surrounded by other people and
finding your way without touching anyone. But since each character has a social space to
maintain, measures can be taken when another character intersects with the social space.
In (Karamouzas & Overmars, 2010) they provide a solution to this problem. When such
an intersection takes place a new rating formula is used:
In this formula the angle and speed energy factors have been left out since it is of most importance to escape the intersection. The desired velocity in the deviation term has been set to zero. Finally $t_c$ in this formula represents the time it takes to get out of the intersection. This rating causes characters to choose the velocity that quickest gets them out of the intersection. This approach works really good but can result in robotic like behavior of the characters. Since the shortest way out of an intersection might be the opposite direction or require a very harsh turn. How sudden changes can have poor affects on the animator is covered in section 3.2.2. Also, when coming out of an intersection the best velocity might be the same one as before, therefor leading the character back into an intersection. In section 5.1.3 alternative ways to fixing this problem are covered.

### 3.2.2 Animator Problems

The animator plays a huge role in making the system believable. Turning the numbers outputted by the algorithms to humanoid character movements. When designing this system two animators were tried. The first one was provided by Unity and was quickly deemed unusable since it did not have a correlation between input and output. Some time went into trying to adjust it but designing an animator is very challenging to get right and also requires good animations. It was then decided to use the same animator as Impulsion used, only minor adjustments had to be made to get it working. But still it isn’t perfect, the main complications are described in the next subsections.

#### Animator Correlation

As mentioned in section 2.4.4, it is important that the animator is able to match the requested speed. This is because if the Social Navigation System decides that a certain speed is required to avoid a collision it is important that the character actually moves at that speed. For the Impulsion animator this was a problem since a requested speed of less then $1.55 \text{m/s}$ resulted in up to half as slow speed. To counter that a small adjustment was made in the locomotion class:

$$\text{Speed}_{\text{new}} = \text{Speed}_{\text{req}} + \text{Speed}_{\text{req}} \times (1.0 - (\text{Speed}_{\text{req}}/1.55)) \quad (3.5)$$
This equation is applied only if the requested speed is less than 1.55, higher speeds do not need this adjustment. This equation adjusts the new speed linearly with the requested speed. Faster speeds will need less adjustments and slower speeds will need more adjustments.

**Animator Latency**

Another problem with the animator is the latency in it. Just like when a human starts running he can’t go from standing still to running in an instance, he has to take the time to accelerate to his running speed. The same happens in the animator, accelerating and turning both takes time. This can be problematic, especially in very crowded environments were constant changes to the velocity are required. The system might decide to turn 90 degrees to avoid a collision but because of the time it takes to turn the character the collision has already happened. It’s possible to design an animator that behaves like an athlete, accelerates very fast and turns in an instance, but that just isn’t how we usually behave. It could however be possible to create a hybrid solution where sprinting away to avoid a collision is an option.

**Drunk Characters**

In some instances the characters can be seen behaving like they are drunk. Tilting from one side to the other. This is because in the animations to turn the characters lean a bit when turning. That coupled with turning a bit to the right and then adjusting again to the left can cause the characters to wobble quite a bit, like they are drunk. To counter this would require changing the animations with ones that don’t lean into a turn. New animations would either have to be created or purchased, and this is beyond the scope of this project.

### 3.3 Optimizations

Since each cycle of the algorithm involves calculations against all the other characters it is therefore very dependent on the number of characters. That makes it also very important that there are no unnecessary calculations being done since they multiply fast with the number of characters. Optimization was of course one of the design goals when developing the system and it does reflect that. However there are two optimizations that stand up.
3.3.1 Nav Resolution

This one is very straightforward and was created early on in the project. It simply makes sure that the navigation code is run at a fixed interval instead of every frame. There is a global variable that defines the resolution of the calculations. It’s a floating point variable that defines the time between each calculation, for example 0.1 would result in 10 calculations per second. The weakest point of this approach is that all agents use the same resolution meaning that they usually perform their calculations at the same time. So given that they are set to perform the calculation once per second, if there are many characters the game will slow down every second. Now this is one of the simplest optimization possible, but it could easily be expanded to some extent.

One way would be to have some sort of scheduling so if for example calculations are done every second. Instead of all the characters do their calculation at the same time a scheduler would make sure that they do their calculation one at a time over the course of one second.

Another way would be to have a separate resolution for each character. They might then control the resolution them self. So when there are relatively few collision occurring the resolution can be increased. While if there are many collisions the resolution would have to be decreased. That’s a very simplified version of such an optimization. Finding the correct way of detecting when to increase and decrease the resolution can be difficult.

3.3.2 In Range

This optimization was added much later then the resolution. At first when doing collision detection against other characters a list of all characters was iterated. Then a simple in range calculation was done, if the characters were in range a collision detection would occur between them. With this approach if a character was wandering far away from all the others he would still have to iterate over the all the others to check if they’re in range.

To counter this Unity’s built in physics engine came in handy. Each character gets their own sphere collider, with the radius of their desired speed multiplied with the maximum time to collision constant. Each character is then able to keep track of the characters in range of him. A message is sent to each character when another one enters and exits their range collider.
3.4 Debugging

Since this is a very visual system debugging can be quite difficult. Some visual aids were developed to ease the understanding of the functionality of the system. In this section these different techniques will be covered. All of the debugging aids where developed into an optional, separate component. That means it is not necessary to include the debugging tools to have an operational character. It’s also possible to disable and enable each aid for every character as to not fill the environment with debug information.

3.4.1 Navigation Path

This was the first aid developed, serving both the purpose of viewing the path that the character has traveled and to compare the system with Unity’s navigation system. The class logs a characters position and speed according at a defined interval. This log can then be rendered as a path and colored according to the speed. This is very helpful in detecting whether a character is slowing down or speeding up too much. The path of a character controlled by the Social Navigation System can also be compared to a path of a character controlled by Unity’s navigation system.

In figure 3.5 an example of the navigation path can be seen. In that example character 1 waited at the doorway while character 2 passed through it. It can be seen in the path as it is red right before the door. The color should indicate how good the speed is. Green is when the character maintains its desired speed, yellow is a little slow but still acceptable while red is very slow and unacceptable. Although it is acceptable in this case since there was no other alternative.

Figure 3.5: A navigation path for a character (1) that waited at the doorway.
3.4.2 Different Velocities

A lot can be observed about the behavior of a character from his different velocities. As explained in the beginning of chapter 3 every character has three distinct velocities. There is the Nav Mesh Agent desired velocity rendered as red, the Social Navigation Systems desired velocity rendered as yellow and finally the current velocity of the character rendered as blue. These different velocities were presented in figure 3.2. In figure 3.6 is an example of how viewing the different velocities comes in handy. In the example character 1 is about to collide on his Nav Mesh Agent desired velocity, the red one. The Social Navigation System has then found a new velocity for him, the yellow one. His current velocity then is the blue one and it is progressing from the red one to the yellow one. Recall how the animator can exhibit latency as described in section 3.2.2. That latency can be seen as he’s progressing from his current velocity (blue) to the Social Navigations Systems desired velocity (yellow).

Figure 3.6: An example of how a character maintains three different velocities at any given moment. Red: Nav Mesh Agent desired velocity. Blue: Characters current velocity. Yellow: Social Navigation System desired velocity.

3.4.3 Collision

As already seen in figure 3.6 a special capsule is created when a collision is imminent. The capsule is intended to look unrealistic as it doesn’t represent a physical object but an immaterial collision. At the moment only the most imminent collision is displayed. Recall from section 3.1.1 that a character can be avoiding many collision simultaneously. This could easily be extended to see every collision an agent is avoiding.
3.4.4 Intersections

As already mentioned in section 3.2.1, characters can intersect with one another’s social space. Then a new rating formula takes over to guide the characters out of the intersection. It’s good to be able to see when this interaction takes place and when this new rating formula is applied. In figure 3.7 an example of this aid can be seen. A red sphere is placed over a character when another one has entered his social space.

![Figure 3.7: A red sphere is placed over a character when someone enters his social space.](image)

3.4.5 Deviations

When perfecting the rating formula or understanding why a certain velocity is considered the optimal velocity, it is good to see all possible velocities and their ratings. This is possible with the deviations aid. It renders all the velocities and their ratings next to them. It list all the sections in the rating formula and then finally the total rating. That makes it possible to notice if one section is out of order and giving a to larger or small value. The velocities tend to get many so when viewing them it is good too increase the step size to decrease the number of velocities. In figure 3.8 a large set of possible velocities can be observed. Recall that the number of velocities increases as a collision gets closer.

3.5 Integration with Impulsion

In order to give the characters more life like behaviors the Social Navigation System was integrated with Impulsion (Pedica & Vilhjálmsson, 2012). Impulsion uses Unity’s built in navigation system which is used as comparison in chapter 4. At first a passive Social
Navigations System character was created. That allowed an Impulsion character to behave just as he normally would but characters that were a part of the Social Navigation System would be aware of him. That way Impulsion characters behaved just like they always had and the Social Navigation Characters avoided them like as if they were part of the Social Navigation System. This passive character isn’t Impulsion specific and could be applied to any character that uses Unity Nav Mesh Agent for navigation.

A passive character is only a half integration. The goal was to have Impulsion characters walk around and avoid each other using the Social Navigation System. To do that the avoidance behavior described in section 3.1 had to be ported to a separate class. That means that the avoidance behavior was exposed using only a single function and the user doesn’t have to have further knowledge about the functionality. That single function takes the Nav Mesh Agent desired velocity and returns an optimal velocity.

Impulsion is built using behavior trees as described in (Pedica & Vilhjálmsson, 2012). Behavior trees are a broad topic and won’t be covered in detail but they are considered to be ideal for next generation game AI (Champandard, 2007). In order to incorporate the Social Navigation System into Impulsion it had to be ported to a behavior tree decorator, one type of a behavior tree node. The main author of Impulsion, Claudio Pedica took care of setting up this decorator. It’s then possible, when creating Impulsion behavior trees to use this decorator to achieve the effect of the Social Navigation System. Since all the avoidance code is in a separate class, if changes are done to it, they’ll affect all projects that use it, including Impulsion.

Behavior trees are not a core feature of Unity, instead a third party plugin, Behave ¹ is used to create behavior trees. In figure 3.9 an example use of the Social Navigation

¹ http://angryant.com/behave/
System can be observed. In the figure the Social Navigation System decorator is named *AvoidObstacles* and is followed by a *Navigate* action, which is a part of the Impulsion project. This is a simple example of how the Social Navigation System can be used but much more complex behavior trees can be built.

![Diagram](Image)

**Figure 3.9:** An example of using the Social Navigation System (AvoidObstacles) within Impulsion.

With the integration, characters can now exhibit the strength of both systems. They can walk around avoiding each other and then join a group of other characters in an Impulsion conversation. Further integration of the systems is also possible. Such as adding a social group awareness described in section 5.1.2.
Chapter 4

Experiments

In this chapter a few scenarios will be covered. During development of the social navigation system these scenarios where used as unit tests. As to be expected from unit tests, some would fail while others looked astonishing, the goal is to make all scenarios work at the same time.

It is important to see that this system is an improvement over other systems. In (Karamouzas & Overmars, 2010) they compare to the RVO algorithm. As already mentioned Unity uses RVO for its Nav Mesh Agent making it easy to compare this system to RVO. There are two forms of observing both systems. They can be used to move basic objects such as boxes or they can be used to move animated characters. The goal is of course to have animated characters so that form will be used for comparison.

All of the scenarios can be observed and tested on a web version of the system on the SECOM development server. Go to http://secom-dev.ru.is/nav to experience the scenarios. There it’s possible to run all the scenarios with the social navigation system as well using Unity’s built in navigation system. It’s also possible to use a cylinder object instead of the characters.

The scenarios are split into two, there is a set of levels that define different environments, then each level can have their own set of configurations. Every level tries to emphasize some fundamental environmental factor so that the system gets tested on a broad spectrum of environments. The configurations also try to strain the system as much as possible in every level. In the following text each level gets its own section.
4.1 Doorway

This level has broad space all around except in the middle where there is a doorway. The doorway can only comfortably fit one character, although two can squeeze through at a time, it’s not a favorable outcome. This level stresses the fact that character must be able to wait for their turn instead of powering through and hoping for the best.

4.1.1 One on One

In this configuration there are two agents starting at an equal distance from the doorway and they travel at the same speed. This means that without any intervention they’d arrive in the doorway at the same time. In figure 4.1 a comparison of the SNS and Unity’s native RVO implementation can be observed. The SNS handles this very well. Either character, non deterministically, will slow down and allow the other to pass. RVO however tries to power through and hopes for the best, both characters fit in the doorway but their bodies collide a little bit.

(a) SNS. Character 1 waits while character 2 enters through the doorway.
(b) RVO. Both characters try to enter through at the same time, causing their bodies to collide a little bit.

Figure 4.1: Doorway, One on One

4.1.2 Three on Three

This configuration adds more complexity than the previous one. Now there are three characters on either side of the doorway. There is a character in every corner and they are all within an equal distance from the doorway. The last two are in the middle heading straight for the doorway, which means they’ll arrive there before the ones in the corners. Figure 4.2 compares this scenario. In the comparison characters 1 and 2 arrive in the
doorway first. There is a lot to notice in this comparison, firstly it’s obvious that in figure 4.2a character 2 not only waited for character 1 but also one other after him. Then character 2 was the third one through the doorway and following him are both the other characters from his side. It’s interesting how they queue up for crossing through the doorway. This is specially mentioned in (Karamouzas & Overmars, 2010) as a strong feature of their system and that this sort of queuing actually happens in real life. In Unity’s RVO case seen in figure 4.2b character 1 and 2 are an equal distance from the doorway as they solved their interaction the same way as in the previous scenario. Then the four corner characters arrive after them, all at the same time, causing a big group collision with severe collisions of bodies.

(a) SNS. Characters exhibit queuing to enter through the doorway.
(b) RVO. Characters try to resolve their collisions within the doorway causing their bodies to collide.

Figure 4.2: Doorway, Three on Three

4.2 Small Corridor

This level stresses the weak point of the system, little space. Without the wast space to deviate, finding an optimal velocity can be problematic. In tight places character can even intersect with one another as described in section 3.2.1. It’s still interesting to look at this level to observe the systems weak points.

4.2.1 One on One

As before in the doorway level the characters start the same distance from the middle, causing them to meet there. They slow down a bit before meeting, reflecting the difficulty of finding a collision free velocity. Remember that they also have a social space that they try to maintain. In the end they are capable of squeezing through without intersecting.
Although the slowdown can be considered a flaw of the SNS another view might be that they slow down because they’re unsure they’ll both fit through. Unity’s RVO doesn’t exhibit this slowdown and no intersection can be observed for the characters. However as seen in figure 4.3b only character 2 moves out of the way while the other just powers through leaving them coming unnecessarily close together. The deviation also happens very late, just as the characters are about to collide, something a human wouldn’t do to a stranger in a tight corridor.

![Figure 4.3: Small Corridor, One on One](image)

(a) SNS. Both characters deviate. (b) RVO. Only character 2 deviates giving both characters little space to meet.

### 4.2.2 Six on Six

This is the one of hardest configuration for the SNS. Although the characters do align to queues they don’t posses the knowledge to form a single queue. Instead two queues in the same direction form and block the path. This does cause the agents to intersect but with the intersection avoidance described in section 3.2.1 the models don’t collide but he characters can exhibit a bit of a robotic behavior. Amazingly Unity’s RVO handles this very well as seen in figure 4.4b without any major intersections. That’s however one possible outcome, it’s just as likely to see them just power through with severe collisions of the bodies. They also squeeze much closer together compared with the social space in the SNS.

### 4.3 Medium Corridor

With the increased space the SNS can really start to enjoy itself. Although the level still isn’t big enough to support a large amount of characters the systems parameters can be tweaked a bit to exhibit some interesting features.
4.3.1 One on One

In this configuration the social space setting for each character will be explored but before that the usual comparison with Unity’s RVO. In figure 4.5 that comparison can be observed. As per usual Unity’s RVO solves their collision right before they happen while the SNS does so in a timely advance. As observed before the characters controlled by RVO tend to collide a bit. In figure 4.5a the default social space of 0.5 is used. Recall that each character is only aware of his own social space but not the others. That means that the distance between the characters will amount to just 0.5 instead of their double social space of 1.0. This also means if one of them sets his social space to zero they’ll keep the same distance as before because the other one still maintains his social space.

In figure 4.6 the affects of changing the social space can be observed. In figure 4.6a the social space has been completely removed causing the characters to only avoid their physical radius. On the opposite the social space has been maximized in figure 4.6b,
there really isn’t enough space for the characters to maintain a larger social space. Having a large social space also overlaps with the core goal of the system, to make small changes early on. With a large social space, big deviations are required to maintain that space.

![Image of characters in a corridor with social space](image)

(a) SNS. Social Space: 0.0. Characters avoid just the physical size.  
(b) SNS. Social Space: 1.5. Characters avoid a large social space.

Figure 4.6: Medium Corridor, One on One, Social Space

### 4.3.2 Twelve on Twelve

Although the level is a little bit too small to support this many characters it’s still of interest to see how the SNS handles it. For this configuration Unity’s RVO won’t be covered, since it behaves very much as before, with a lot of collisions. In figure 4.7 the scenario can be seen in two different figures. At first as seen in figure 4.7a all the characters clutter together in one big group intersecting quite a bit with each other and exhibiting robotic behavior. Once each character reaches his destination at the other side of the level he returns to his original location and keeps going back and forth. Then after awhile the scenario reaches a sort of equilibrium with minimum intersections as seen in figure 4.7b.

### 4.3.3 Overtake

This configuration was of huge importance when designing the system. When a character wishing to travel faster than another one he must be able to pass or overtake him. The process of doing so must also look very convincing. At the beginning of the scenario character 1 is behind 2 and tries to maintain a faster speed than 2. As they walk through the corridor character 1 should overtake character 2. In figure 4.8 the differences between SNS and RVO can be seen. SNS performs great by deviating character 1 around character...
2, however RVO is unable to perform this maneuver. In some cases RVO is able to power through, pushing character 2 a bit to the left side and squeezing through on his right side.

Figure 4.8: Medium Corridor, Overtake

4.4 Large Corridor

This is where the SNS really gets to shine, with enough space to deviate the algorithm can really show its strength. Many of the features already observed will be exhibited in this level.

4.4.1 Three on Three

All the characters start straight in front of one and another. Each trio is on opposite sides of the level, with just enough space for the characters to walk in a side by side line. In
the SNS, once they meet in the middle some gaps in the line will have occurred allowing
the characters to pass through. Queuing can often be observed in these circumstances.
In RVO the characters try to fit through the original space between the characters, which
is barely big enough to fit a character. That usually causes at least one collision to hap-
pen.

(a) SNS. Characters queue up for gaps in the
(b) RVO. Characters try to squeeze through the
lines.

Figure 4.9: Large Corridor, Three on Three

4.4.2 Twelve on Twelve

With a large level, having a lot of characters finally gets interesting. To fully understand
this scenario it’s necessary to know the initial position of all the characters. In figure 4.10
the initial positions can be seen.

(a) SNS. Characters queue up for gaps in the
(b) RVO. Characters try to squeeze through the
too small space between characters.

Figure 4.10: Large Corridor, Twelve on Twelve. The initial position of all characters.

The idea is to have a large collision in the middle while still have characters walking on
the sides. The comparison can be seen in figure 4.11. The SNS handles this very well,
the characters spread over the map making it easier to navigate. Several queues appear in
the scenario and interactions rarely happen, it they do they’re hard to detect. RVO on the
other hands pushes all characters at full speed causing them to form three groups, trying
to solve the interactions. Large amount of body collisions can be observed.

(a) SNS. Characters spread out and form queues. (b) RVO. Characters try to resolve all the colli-
sions in the middle.

Figure 4.11: Large Corridor, Twelve on Twelve

4.5 Austurvöllur

This is the final level. It’s the one that’s used in the Virtual Reykjavik project. This level
demonstrates that the Social Navigation System works in any compatible Unity Scene
that has a Nav Mesh. In figure 4.12 a busy day at Austurvöllur can be seen using the
SNS. In this configuration there are 30 characters placed manually around the statue of
Jón Sigurðsson and then another 50 are placed at random anywhere in the level. That
amounts to a total of 80 characters, showing that the Social Navigation System is capable
of a high number of characters.
Figure 4.12: Busy day at Austurvöllur.
Chapter 5

Conclusions

The original goal of this project was to create a socially believable navigation system. The previous work from (Karamouzas & Overmars, 2010) proved to provide good results and the implementation was made easy using the tools within the Unity game engine.

Since the implementation relies on only core features of the Unity engine the system can easily be integrated into any Unity project. The integration with Impulsion and Virtual Reykjavik is a good example of that.

In the last chapter we observed some good results. The Social Navigation System is an improvement on Unity’s own navigation system and is also capable of producing socially believable behaviors. There are some shortcomings as described in section 3.2 and some problems were also observed and discussed along with the examples. To conclude the report some possible future work is covered in section 5.1.

5.1 Future Work

The system has it’s weaknesses and there are some ways to improve and expand it. This section describes some of the ideas that have sprung up during the development of the project.

5.1.1 2D to 3D

Throughout the project and in this report all of the functionality has been simplified to a two dimensional plane. Porting the project to support a full 3D environment with multiple floors and stairs between them is a possibility. However there are some challenges that
arise. The most obvious one is to animate the characters in that way. Since the current animator doesn’t support walking up and down stairs or slopes, it would require a set of new animations along with changes to the animator. Without going into too much detail this is much harder to get right than simply traversing a plane. Special care has to be taken to have the feet place correctly on the stairs without colliding with them. None the less it’s a solved problem and only requires time to tune correctly.

The Unity Nav Mesh and the Unity Nav Mesh Agent support slopes or stairs and multiple floors so path finding is possible. The biggest challenge would be to do the additional social collision detection since it isn’t possible to simplify the velocity to a single vector. Imagine standing in front of stairs walking towards them, at that point the velocity is on a two dimensional plane, but as you walk up the stairs the velocity changes to a three dimensional velocity up the stairs.

As always it depends on how good end results are required. It would be possible to solve stair traversal by employing a strict right hand traffic only, making it much easier to solve collisions in the stairs or simply unnecessary. That way characters can be expected to only enter and exit the stairs at a single location. Each side of the stairs would have an entry location and an exit, like a tunnel. These locations could then be considered as a character on the two dimensional floors, indicating the time to the nearest character in the stairs, either coming or going. Allowing others to predict whether or not someone is coming out from the stairs.

Some sort of partitioning like that would make the problem a lot simpler. It wouldn’t be enough to just add the height variable to all the current equations. A lot more logic would have to be added such as collision detection through multiple velocities.

### 5.1.2 Social Group Awareness

As the system gets integrated with Impulsion some interesting problems arise. The biggest one is social territorial awareness, where characters notice when others are employed in a social activity, conversation for example. This is depicted in figure 5.1. As two characters are engaged in a conversation there is space between them that is meant for socializing. In real life we are fully aware of this and never walk between people engaged like that, except there is absolutely no other way. The Social Navigation Systems rating formula should be adjusted to account for this social space so velocities that go around it are rather chosen. Ground work for this already been represented in (Oliva & Vilhjálmsson, 2014), all that has to be done is apply it to the Social Navigation System with respect to Impulsion.
5.1.3 Intersections

As already covered in section 3.2.1 the biggest drawback of the system are intersections. Caused when a character picks a velocity that goes through another character, which is likely in a crowded environment. There is a fix for this by using another rating formula but that tends to produce robotic behaviors. One way would be to improve the rating formula opting to choosing a less diverging velocity. Another way would be to have characters slowing down or simply stopping when an intersection is detected. Or the combination of both, if a very dramatic change is required to escape the intersection, simply stop and hope the other character will solve the intersection. To make it more likely to function properly a check could be made to decide which character is better suited to solve the intersection while the other one stops and waits.

5.1.4 Flow Adjustment

The solution to intersections as described in the previous sections can be thought of as solving a problem while the actual navigation system can be thought of as problem avoiding. Intersections are something that in perfect world should never happen and the system tries to avoid it at all costs. However if it accidentally happens there should be a solution out of it. The idea of flow adjustment is to add more emphasis on problem avoidance. It’s considered to be useful in a crowded environment. The idea comes from the scenario in the small corridor with six on six characters. In figure 5.2 the results of the Social Navigation System can be seen in that scenario.
In this scenario there is enough space for two and only two queues, one in each direction would be optimal but the characters aren’t that clever. It would be nice if they’d figure out that if they all keep to their right there would be enough space for everybody to pass through the corridor. The idea for flow adjustment is to make the characters aware that if there is limited space the best bet might be to follow someone else going in a similar direction. There are two problems that need to be solved in order for this to function properly. First is to know when there is limited space and that this approach should be activated. Second is how to implement the actual flow adjustment.

This must be a very simple calculation since it might be done against many characters, multiple times per second. When checking this for two characters one way might be to subtract their velocities. If the magnitude of the resulting outcome is small it means that they are going in a similar direction at similar speed. On the other hand a larger result indicates that they are not so parallel. This is untested but might provide a simple enough way to decide which character to follow in case there is limited space.
Bibliography


