Kite: a 2D bounding shape for human crowd simulations

Yu Wang
Department of Computer Science
University of Maryland
College Park, MD 20740
ywang185@terpmail.umd.edu

December 2019

Abstract

Kite is a 2D bounding shape initially for human crowd simulation. It was inspired by my personal experience as a pedestrian that a pedestrian's perception to its neighboring environment is not evenly distributed to different directions. To implement a bounding shape with this idea, four sensing directions are chosen and the sensing distances can be customized, which gives us a kite-shaped bounding shape. Based on this bounding shape, repulsive forces are computed to adjust the distances between different agents.

Keywords— bounding shape, crowd simulation, Unity, geometry

1 Introduction of purpose

With the development of building constructions and the scale of public transportation, the capacity of evacuation is becoming more and more important as a concern for the designers and operators. For a building with more than a hundred floors, testing the evacuation with real humans is not very practical. As an obvious substitution, a computer-based evacuation simulation can be used to evaluate a building/vehicle's design on the safety side, which is cheaper and more convenient. In addition, this procedure can be done before a real one is built or a 1:1-scaled model is created, so that the designers can change the plan before it is too late. To simulate the evacuation, a key part is to simulate how the crowds of people in certain structures move under certain emergencies. Since we are humans, perfection is never a feature of us, and being calm can only happen on a portion of people, so using a perfect crowd simulation for evacuation simulations may only predict the best case, but that is far from the reality. There need to be a more realistic method to simulate the potential chaos in a crowd, not guiding the agents with omnipotent views and skills.

Figure 1: agent based evacuation simulation -Pathfinder

Another corresponding industry to crowd simulations is entertainments. With no doubt that the video game industry needs crowd simulations to move groups of NPCs (non-player characters) around, such as Starcraft and other RTS (real-time strategy) games. One other good application would be CGI (computer-generated imagery) in movies. For those legendary battles with huge scales, hiring enough people to get in the battlefields is neither practical nor controllable. Using computer-based simulation is again a good substitution.

Figure 2: Starcraft II, a good example of multi-agents controlling

2 Methods and implementation

Kite itself is a 2D bounding shape. Instead of creating a primitive of the bounding shape, I represent the bounding shape by four axes: front (CE in Fig. 3), back (CD in Fig. 3) and side (CA and CB in Fig. 3). The side axis is considered symmetrically to the left and to the right of the agent, so I consider it as two axes. These four axes support the shape of kite from the standing point of the agent, and the bounding shape is vertical to the Transform.Up vector of each agent. Although this shape is not center symmetric, I am going to call the standing point (point C in Fig. 3) as the "center" for simplicity for the rest of this paper. In the following, I am going to describe how this bounding shape works in different procedures.

2.1 Initialization

The bounding shape is wrapped in a class called "boundingstuff". To create a Kite bounding shape, the length of front, back and side axes are passed in together with the current Transform of the agent so that the shape can find its position and direction. Since the agents are not static, the bounding shape moves and rotates with the agent by updating the Transform of the bounding shape.

2.2 Collision detection

To check the collision detection, certain vertices are used to check whether they are in the boundary of the corresponding shape. To do so, a function called Steparound would iterate around the four vertices of the kite and return the vertices that are closer to the center of corresponding kite compared with the distance between two centers.

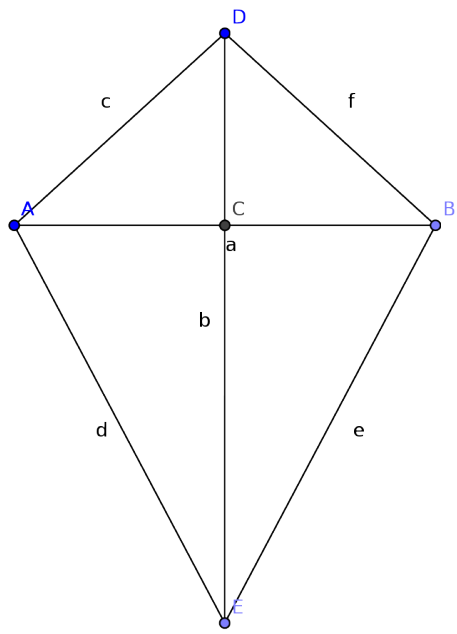

Figure 3: a picture to show the idea of Kite

From the geometric features of kite, it is obvious that Steparound can either return one or two vertices. Then we are going to use them for collision detection.

There are three functions created in "boundingstuff" for the purpose of collision detection: CD, CDline and CDs. When there is a collision, these functions would return a reaction force to avoid the potential collisions. CD is good to handle the collision detection of one vertex against a kite bounding shape. But for the case of handling the collision detection of two vertices against a kite bounding shape, there can be several ways to do: 1) get the mid-point of these two vertices and do collision detection on this point instead, or 2) get the reaction forces of these two vertices individually and compute the averaged reaction force, or 3) compute the reaction force to the line segment using CDline. All these three options are implemented, and CDs can do the corresponding option indicated by a given parameter called op that takes the indices above.

2.2.1 CD

CD handles the computation of checking one vertex against a kite bounding shape. It projects the vertex in the coordinate space that the origin is the center of the kite, one axis is the front, and the other axis is the right. When the point is not in the boundary of the bounding shape, a force of 0 would be returned, otherwise, separate reaction forces can be computed based on the penetrations on each axis. As mentioned, there

are four axes in a kite, and the four axes split the region of Kite into four triangular areas. Each triangular area is bounded by two perpendicular axes. The reaction force F is computed as:

$$F = V_1 \times P_1 + V_2 \times P_2 \tag{1}$$

*The subscribes are two bounding axes for the triangular area that the point falls in. P stands for the penetration ratio on the given axis, which is computed by dividing the penetration depth on that axis by the axis length.

*For the case that the vertex is exactly overlapping the center of Kite, I finally changed my mind to add an extra parameter for the center of Kite that the point belongs to, so that the function can figure out which region should include this point and how the reaction force should be computed.

2.2.2 CDline

CDline is a more complicated function comparing with CD. It computes the foot of perpendicular of a line that goes through the center of Kite and the connection between the foot of perpendicular and the center of Kite on the given line segment. A similar idea is used to compute the penetration ratio, then the reaction force F is computed as:

$$F = \sqrt[2]{P_1^2 + P_2^2} \times LC \tag{2}$$

*LC stands for the vector of the connection between the center of Kite and the foot of perpendicular on the given line segment. It is the vector that starts from the center of Kite and points to the foot of perpendicular on the given line segment (for some cases, the foot of perpendicular can be on the line instead of the line segment, but this doesn't really matter).

*Since CDline uses a different strategy to compute the reaction force, it doesn't need to use center point to take care of the corner case. So I keep the initial version without the change that is similar to the one made on CD.

2.2.3 CDs

CDs takes a list of points (List¡Vector3¿), the center point of another Kite(Vector3) and an option indicator (int) as the input. As described in the earlier paragraph, this function works as a function selector that handle either single or two vertices in the given list. When there is only one vertex, it invokes CD directly; otherwise, it does the corresponding manipulation and invokes either CD or CDline following the indicator. When the indicator is not valid (not 1, 2 or 3) or the point list is somehow empty, this function would return a force of 0 as default.

2.3 Loop-runner

In my current implementations, a central controller of all agents is necessary, and there is a dual-loop procedure in the update function to compute the corresponding reaction forces for each agent. Since there are speed limits for all the agents (which are also the lengths of their front axes of the bounding shapes), I can predict the lower-bound

of collision-free time by dividing the distance by the sum of the speed limits. In the period of collision-free time, collision detection procedure can be saved on this pair.

For those pairs that are not skipped due to the prediction, the corresponding reaction force is computed mutually, which is due to the arbitrary direction of Kites. To make the reaction forces fair for each agent, they are scaled to the speed limit of each agent, so that the slower agent takes a smaller force and a faster agent takes a larger one. And since the reaction forces are computed with different reference Kite, to uniform them, averaging on the forces are needed. So the reaction forces are averaged first, then rescaled with the speed limit.

After the dual-loop, each agent has a corresponding sum of reaction forces. These forces then decide how the agents move together with the precomputed velocities. For different conditions, there can be different heuristic functions to manipulate the agents' movements based on the forces. More details will be elaborated in the "Demos and experiments" section.

2.4 Navigation

To boost the performance of crowd simulation, navigation information is pre-computed as the baked Navmesh component in Unity. So for every agent that stands on the Navmesh, it already has a path and knows the right velocities to get to its destination from an arbitrary position on the Navmesh. Thanks to the supportive components in Unity, I don't really need to worry about this part.

3 Demos and experiments

In this project, I have created two types of demos. One is the common demo that a given number of agents stand on a circle uniformly and are required to go to the corresponding symmetrical points of their initial position about the center of the circle, I call it "Circle demo" for short. The other one is a house-like demo with floors and stairs. Different agents are initialized randomly on different floors, and their movements generates crowds that move against each other. I call it "House demo".

A fundamental idea in this project is, how to avoid stagnation in a moving crowd. By simply adding a reaction force to an agent, it is very likely that the force balances out the velocity, then causes stagnation. According to my personal experience and daily observation, I have found that in daily life, an obstacle in the way would cause a turning of the pedestrian if the collision can be avoided by doing so. Then, I decide to convert the reaction force to a force that makes the agent turn around the potential collision point. Such a rotational velocity can be easily computed with the cross production of vectors, then, an extra dragging force is applied to drag the touching agents apart. I have tuned the parameters so long to improve the result, without using machine learning stuff, it has taken a really long period, and the current parameters are not the best ones I think.

Since more tuning can be made to improve the performance, I am going to provide the open formulas with some parameters that don't have exact values but have some constrains on them:

$$Shift = D \times (F \times \hat{Up}), D \in \{-1, 1\} \tag{3}$$

$$C = \hat{Shift} \times r_S + \hat{F} \times r_F + \hat{Forward} \times r_D \tag{4}$$

$$Velocity_{new} = Velocity_{pre} \times r_V + \hat{C} \times SL \times r_C, r_V \geq 0, r_C \geq 0, r_V + r_C \leq 1 \tag{5}$$

*Shift is the value that the force is turned into a turning, which changes linear accelerations/decelerations to a rotational accelerations/decelearations. C is short for "Change", it combines shift, force, and a potential dragging force together to give a more complicated adjustment to the velocity. For velocity, the new one is computed based on the precomputed one and the change. With the consideration of SL (Speed limit) for each agent, the new velocity is guaranteed to be no larger than SL by using the constraint that $r_V + r_C \leq 1$. I call the ratio of r_C/r_V "boldness" since this shows how aggressive the formula is to change the precomputed velocity.

3.1 Circle demo

In this demo, all agents are distributed on a circle uniformly. Their destination are set properly in the initialization of each agent. An obvious feature of this demo is, the one and only intersection of all the precomputed paths is the center of the circle. Then before reaching this intersection points, the pressure from an agent's neighbors (peer pressure) would keep growing along the way. When there are not a lot agents on the circle, dodging is good enough to let the agents avoid reaching the intersection point;

but when the number of agents grows, how to handle the peer pressure becomes the new concern.

As the circle shrinks and compress the agents more and more, some agents are pushed outward and break the ring. When there are enough agents, agents can be distributed on different layers: the early push-out agents form the outer layer, which walks around the circle until they can go to their destination directly; the later-push-out agents form the inner core of the rotating disk: they rotate around the center and get pushed around by other inner agents and the outer layer. The outer layer will get wider and wider while rotating, and the inner layer will come out and form the new outer layer when the previous outer layer is far enough from the center. Eventually, all the agents will pass the outer layer process and get to their destination.

In my implementation of this demo, when the angle between the precomputed velocity and the total force is less than 90 degree, the force would be ignored since this is a pushing force from the back, and in daily life, people don't usually adjust their speeds based on the reaction from the back. This idea enable the outer layer to use the precomputed velocities more often, which makes the pattern straight forward.

*For this demo, the parameters I am using are:

$$D = 1, r_S = 1, r_F = 0, r_D = -0.1, r_V = 0.9, r_C = 0.05 \tag{6}$$

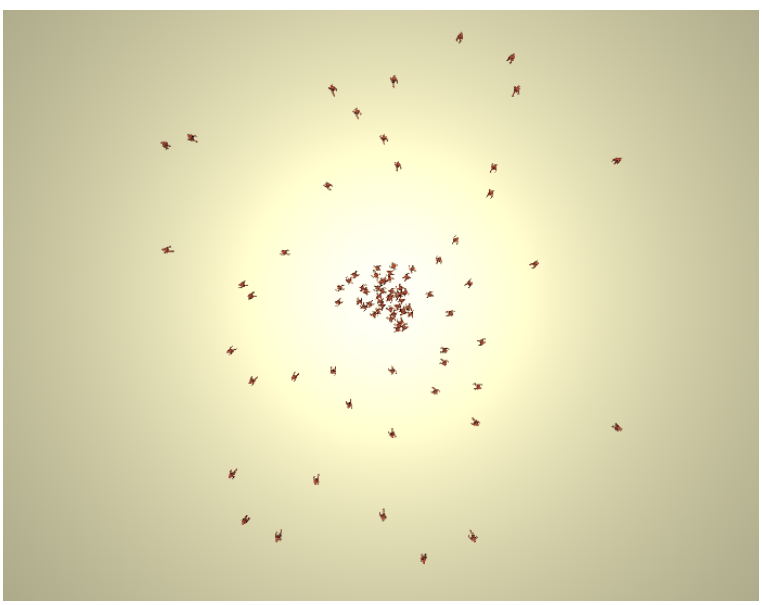

Figure 4: a snapshot for Circle demo

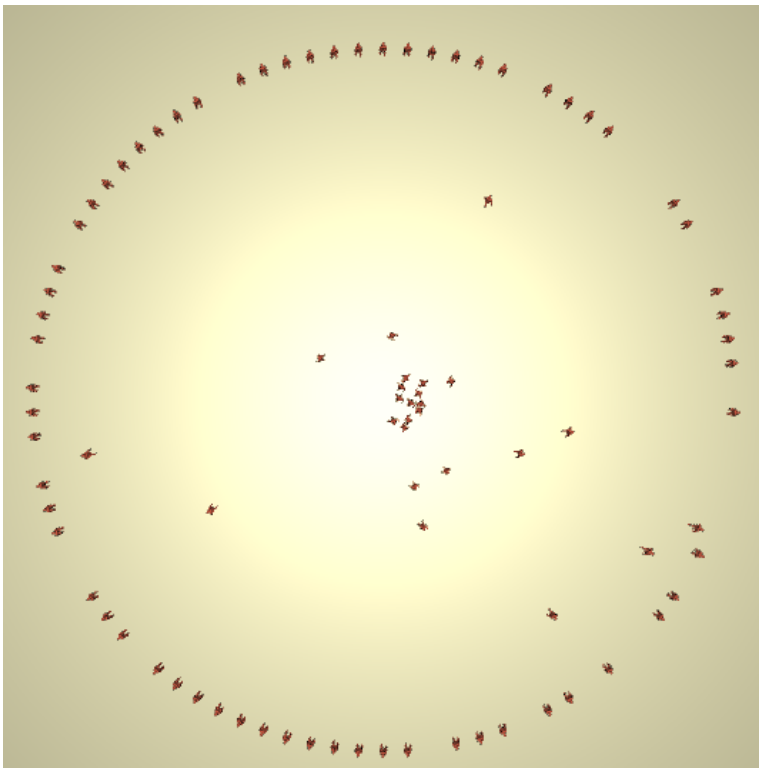

Figure 5: another snapshot for Circle demo

3.2 House demo

In this demo, there are three floors in the constructed structures, and there are two stairways created to connect these floors. It is a basic demo to test evacuation. But it is not interesting to let all people go downstairs, so I want to challenge myself a little bit to let the flows of people go to different floors. (The demo can still test evacuation with some modification on the settings in the code, not a lot.)

In this demo, to boost the performance, I have created a structure called "skipper" to avoid CD check on a pair of agents that are far away. The estimated lowerbound for such a pair's earliest collision time is computed and stored, before that time arrives, the CD check on such a pair is skipped (this is why I call it skipper). This is an extension for the broadphase checking, so it doesn't require to compute any extra stuff. When that time arrives, the broadphase will generate a new estimation. If this time the estimation is still larger than one, it will be stored in skipper as a new value and the narrowphase would be skipped; otherwise, the value will be stored as well, but the narrowphase would not be skipped.

This improvement works well for the case that many pairs of agents are not very likely to collide in a short period of time. It is not that useful when there aren't many agents.

There are two trials that I haven't decided whether they should be used in the demo (they are not straight forward in the formulas, and the corresponding results are not obviously-improved). I am going to put them here in the case if the readers are curious.

3.2.1 Swing-D

By "Swing-D", I mean parameter D should be swing between 1 and -1 based on the round. It is kinda like, if everybody turns left in this round to avoid collision, then in the next round, everybody turns right, so that no continuous turnings can happen and cause some stagnation. Another option would be choose D randomly from -1 and 1.

3.2.2 Half-life

By "Half-life", I mean there are only half of the agents are active to change their velocities, and the other half will stick to their previous computed velocities no matter what. And the two halves are shifting on each round. It is more like a "try-this-out" for my "what-if" ideas.

*For this demo, the parameters I am using are:

$$D = -1, r_S = 0.05, r_F = 0.1, r_D = -0.05, r_V = 0.9, r_C = 0.1 \tag{7}$$

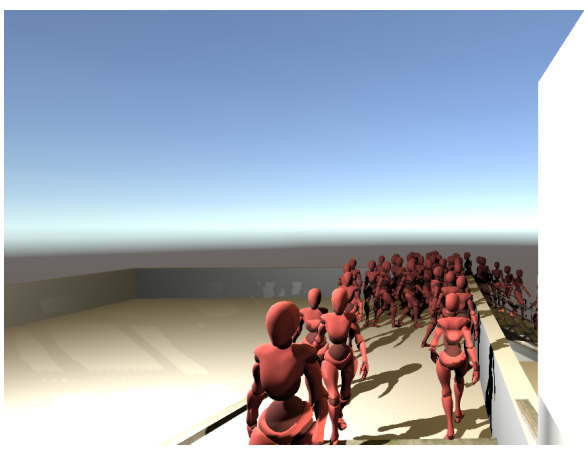

Figure 6: First person view of House demo

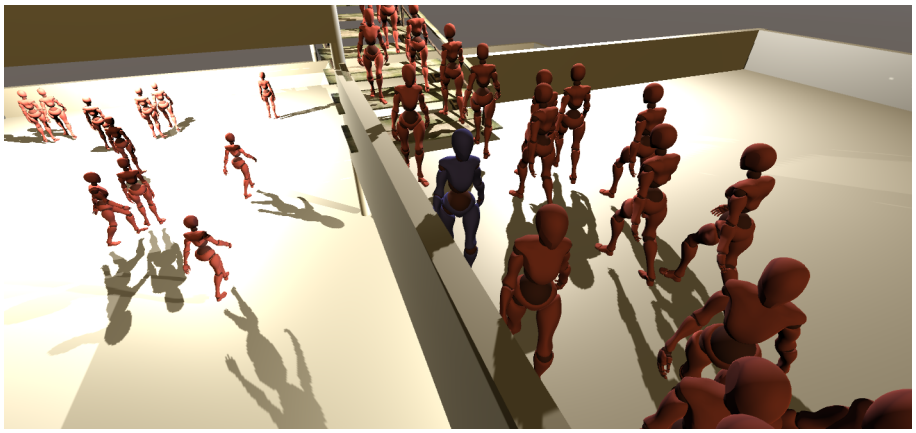

Figure 7: Third person view of House demo

4 Future work

This current implementation comes with some open formulas computing the reaction forces and adjusting the velocities, in which the parameters are not guaranteed to be optimal. I have struggled a lot to construct the formulas and tune the parameters. A more elegant way would be plugging in machine learning methods to find the optimal formulas.

Another potential change can be letting left-side and right-side be separated parameters, which enables more special condition to be considered. For now, as a design for individual-based crowd simulation, the current kite bounding shape works just fine.

A more aggressive change would be bounding shape merging. I thought it would be a reasonable design to save time for a flow of people walking towards the same direction, but it becomes a waste of time when the shapes need to be merged and de-merged pretty often. For certain scenario like a crowd moves in the same direction all the time without worrying some counter-crowds' incoming, this can save a lot of computation time with the loss of flexibility in a crowd.