Using AI to Bring Open City Racing to Life

Angel Studios' MIDTOWN MADNESS 2 for PC and MIDNIGHT CLUB for Playstation 2 are open racing games in which players have complete freedom to drive where they please. Set in "living cities," these games feature interactive entities that include opponents, cops, traffic, and pedestrians. The role of artificial intelligence is to make the behaviors of these high-level entities convincing and immersive: opponents must be competitive but not insurmountable. Cops who spot you breaking the law must diligently try to slow you down or stop you. Vehicles composing ambient traffic must follow all traffic laws while responding to collisions and other unpredictable circumstances. And pedestrians must go about their routine business, until you swerve towards them and provoke them to run for their lives. This article provides a strategy for programmers who are trying to create AI for open city racing games, which is based on the success of Angel Studios' implementation of AI in MIDTOWN MADNESS 2 and MIDNIGHT CLUB. The following discussion focuses on the autonomous architecture used by each high-level entity in these games. As gameplay progresses, this autonomy allows each entity to decide for itself how it's going to react to its immediate circumstances. This approach has the benefit of creating lifelike behaviors along with some that were never intended, but add to gameplay in surprising ways.

AI Map: Roads, Intersections, and Open Areas

At the highest level, a city is divided into three primary components for the AI map: roads, intersections, and open areas (see Figure 1). Most of this AI map is composed of roads (line segments) that connect intersections. For our purposes, an intersection is defined as a 2D area in which various roads join. Shortcuts are just like roads, except they are overlaid on top of the three main component types. Shortcuts are used to help the opponents navigate through the various open areas, which by definition have no visible roads or intersections. Each of these physical objects is reflected in a software object.

The road object contains all the data representing a street, in terms of lists of 3D vertices. The main definition of the road includes the left/right boundary data, the road's centerline, and orientation vectors defined for each vertex in the definition. Other important road data includes the traffic lane definitions, the pedestrian sidewalk definition, road segment lengths, and lane width data. A minimum of four 3D vertices are used to define a road, and each list of ver-

JOE ADZIMA | *Joe has been an AI programmer at Angel Studios for three years. During that time, he architected and implemented the entire AI system for MIDTOWN MADNESS 1 and 2 for PC and MIDNIGHT CLUB for Playstation 2. Joe thanks Robert Bacon, Angel Studios' technical writer, for the exceptional editorial efforts Robert has applied to this article.*

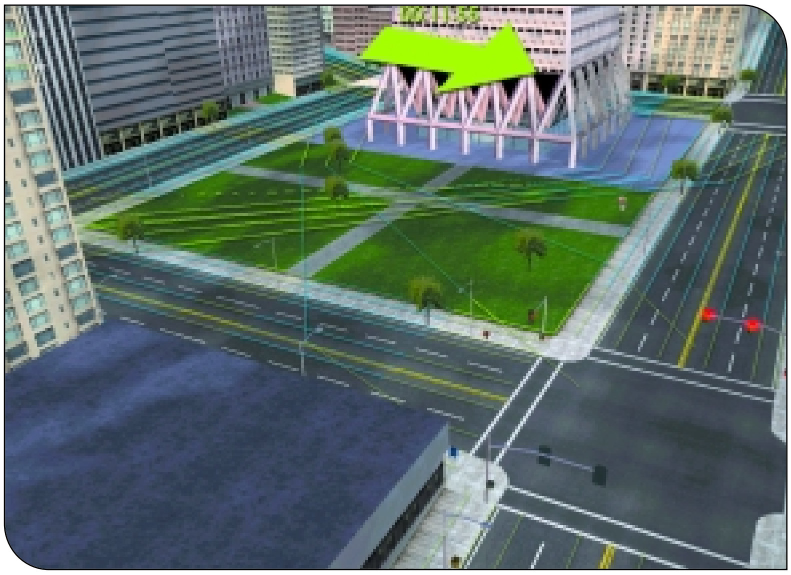


FIGURE 1. The AI map elements appear as green and blue line segments for roads and sidewalks, 2D areas for intersections, and additional line segments for shortcuts across open areas.

tices (for example, center vertices, boundary vertices, and so on) has the same number of vertices.

The intersection object contains a pointer to each connected shortcut and road segment. At initialization, these pointers are sorted in clockwise order. The sorting is necessary for helping the ambient traffic decide which is the correct road to turn onto when traversing an intersection. The intersection object also contains a pointer to a "traffic light set" object, which, as you might guess, is responsible for controlling the light's sequence between green and red. Other important tasks for this object include obstacle management and stop-sign control.

Big-city solutions: leveraging the City Tool and GenBAI Tool. Angel's method for creating extremely large cities uses a very sophisticated in-house tool called the City Tool. Not only does this tool create the physical representation of the city, but it also produces the raw data necessary for the AI to work. The City Tool allows the regeneration of the city database on a daily basis. Hence, the city can be customized very quickly to accommodate new gameplay elements that are discovered in prototyping, and to help resolve any issues that may emerge with the AI algorithms.

The GenBAI Tool is a separate tool that processes the raw data generated from the City Tool into the format that the run-time code needs. Other essential tasks that this GenBAI Tool performs include the creation of the ambient and pedestrian population bubbles and the correlation of cull rooms (discrete regions of the city) to the components of the road map.

Based on the available AI performance budget and the immense size of the cities, it's impossible to simulate an entire city at once. The solution is to define a "bubble" that contains a list of all the road components on the city map that are visible from each cull room in the city, for the purpose of culling the simulation of traffic and pedestrians beyond a certain distance. This collection of road components essentially becomes the bubbles for ambient traffic and pedestrians.

The last function of the GenBAI tool is to create a binary version of the data that allows for superfast load times, because binary data can be directly mapped into the structures.

Data files: setting up races. The AI for each race event in the game is defined using one of two files: the city-based AI map data file or the race-based AI map data file. The city file contains defaults to use for all the necessary AI settings at a city level. Each race event in the city includes a race-based AI map data file. This race file contains replacement values to use instead of the city values. This approach turns out to be a powerful design feature, because it allows the game designer to set defaults at a city level, and then easily override these values with new settings for each race.

Some examples of what is defined in these files are the number and definition of the race's opponents, cops, and hook men. Also defined here are the models for the pedestrians and ambient vehicles to use for a specific race event. Finally, exceptions to the road data can be included to change the population fill density and speed limits.

Curves Ahead: Creating Traffic

Following rails and cubic spline curves. During normal driving conditions, all the ambient vehicles are positioned and oriented by a 2D spline curve. This curve defines the exact route the ambient traffic will drive in the XZ-plane. We used Hermite curves because the defining parameters, the start and end positions, and the directional vectors are easy to calculate and readily available.

Since the lanes for ambient vehicles on each road are defined by a list of vertices, a road subsegment can easily be created between each vertex in the list. When the ambient vehicle moves from one segment to the next, a new spline is calculated to define the path the vehicle will take. Splines are also used for creating recovery routes back to the main rail data. These recovery routes are necessary for recovering the path after a collision or a player-avoidance action sent the ambient vehicle off the rail. Using splines enables the ambient vehicles to drive smoothly through curves typically made up of many small road segments and intersections.

Setting the road velocity: the need for speed. Each road in the AI map has a speed-limit parameter for determining how fast ambient vehicles are allowed to drive on that road. In addition, each ambient vehicle has a random value for determining the amount it will drive over or under the road's speed limit. This value can be negative or positive to allow the ambient vehicles to travel at different speeds relative to each other.

When a vehicle needs to accelerate, it uses a randomly selected value between 5 and 8 m/s^2. At other times, when an ambient vehicle needs to decelerate, perhaps because of a stop sign or red light, then the vehicle calculates a deceleration value based on attaining the desired speed in 1 second. The deceleration is calculated by

$$\frac{\left(V^2 - V_0^{\,2}\right)}{2\left(X - X_0\right)}$$

where V is the target velocity, V_0 is the current velocity, and $(X - X_0)$ is the distance required to perform the deceleration.

Detecting collisions. With performance times being so critical, each ambient vehicle can't test all the other ambient vehicles in its obstacle grid cell. As a compromise between speed and comprehensiveness, each ambient vehicle contains only a pointer to the next ambient vehicle directly in front of it in the same lane. On each frame, the ambient checks if the distance between itself and the next ambient vehicle is too close. If it is, the ambient in back will slow down to the speed of the ambient in front. Later, when the ambient in front becomes far enough away, the one in back will try to resume a different speed based on the current road's speed limit.

By itself, this simplification creates a problem with multi-car pileups. The problem can be solved by stopping the ambient vehicles at the intersections preceding the crash scene.

Crossing the intersection. Once an ambient vehicle reaches the end of a road, it must traverse an intersection. To do this, each vehicle needs to successfully gain approval from the following four functional groups.

First, the ambient vehicle must get approval from the intersection governing that road's "traffic control." Each road entering an intersection contains information that describes the traffic control for that road. Applicable control types are `NoStop`, `AllwaysStop`, `TrafficLight`, and `StopSign` (see Figure 2). If `NoStop` is set, then the ambient vehicle gets immediate approval to proceed through the intersection. If `AllwaysStop` is set, the ambient never gets approval to enter the intersection. If `TrafficLight` is set, the ambient is given approval whenever its direction has a green light. If `StopSign` is set, the ambient vehicle that has been waiting the longest time is approved to traverse the intersection.

The second approval group is the accident manager. The accident manager keeps track of all the ambient vehicles in the intersection and the next upcoming road segment. If there are any accidents present in these AI map components, then approval to traverse the intersection is denied. Otherwise, the ambient vehicle is approved and moves on to the third stage.

The third stage requires that the road which the ambient is going to be on after traversing the intersection has the road capacity to accept the ambient vehicle's entire length, with no part of the vehicle sticking into the intersection.

The fourth and final approval comes from a check to see if there are any other ambient vehicles trying to cross at the same time. An example of why this check is necessary is when an ambient vehicle is turning from a road controlled by a stop sign onto a main road controlled by a traffic light. Since the approval of the stop sign is based on the wait time at the intersection, the vehicle that's been waiting longest would have permission to cross the intersection — but in reality that vehicle needs to wait until the cars that have been given permission by the traffic light get out of the way.

Selecting the next road. When an ambient vehicle reaches the end of the intersection, the next decision the vehicle must make is which direction to take. Depending on its current lane assignment, the ambient vehicle selects the next road based on the following rules (see Figure 2):

• If a vehicle is in the far-left lane, it can go either left or straight.
• If it's in the far-right lane, it can go either right or straight.
• If it's in any of the center lanes, then it must go straight.

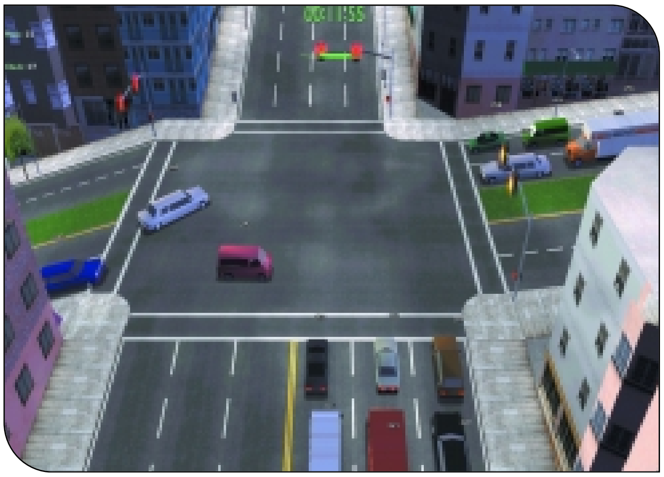


FIGURE 2. In this case, the `TrafficLight` class is set to red for some vehicles, which stop and wait. The other vehicles with green/yellow lights get permission to cross the intersection. The vehicle crossing in the left lane decides to turn left, while the vehicle in the right lane goes straight.

• If it's on a one-way road, then it picks randomly from any of the outgoing roads.
• If it's on a freeway intersection where on-ramps merge with the main freeway traffic, then it must always go right.
• U-turns are never allowed, mostly because a splined curve in this situation would not look natural.

Since the roads are sorted in clockwise order, this simplifies selection of the correct road. For example, to select the road to the left, just add 1 to the current road's intersection index value (the ID number of that road in the intersection road array). To pick the straight road, add 2. To go right, just subtract 1 from the road's intersection index value.

Changing lanes. On roads that are long enough, the ambient vehicles will change lanes in order to load an equal number of vehicles into each lane of the road. When the vehicle has traveled to the point that triggers the lane change (usually set at 25 percent of the total road length), the vehicle will calculate a spline that will take it smoothly from its current lane to the destination lane.

The difficulty here is in setting the next-vehicle pointer for collision detection. The solution is to have a next-vehicle pointer for each possible lane of the road. During this state, the vehicle is assigned to two separate lanes and therefore is actually able to detect collision for both traffic lanes.

Once a vehicle completes the lane change, it makes another decision as to which road it wants to turn onto after traversing the upcoming intersection. This decision is necessary because the vehicle is in a new lane and may not be able to get to the previously selected road from its new lane assignment.

Orienting the car. As the ambient traffic vehicles drive around the city, they are constantly driving over an arbitrary set of polygons forming the roads and intersections. One of the challenges for the AI is orienting the ambient vehicles to match the contour of the road and surfaces of open areas. Because there are hills, banked road surfaces, curbs separating roads and sidewalks, and uneven open terrain, the obvious way to orient the vehicles is to shoot a probe straight down the Y-axis from the front-left, front-right, and rear-left corners of the vehicle. First, get the XZ position of the vehicle from the calculated spline position and determine the three corner positions in respect to the center point of the vehicle. Then, shoot probes at the three corners to get their Y positions.

Once you know the three corner positions, you can calculate the car's orientation vectors. This approach works very well, but even caching the last polygon isn't fast enough to do all the time for every car in the traffic bubble. One way to enhance performance is to mark every road as being either flat or not. If an ambient vehicle drives on a flat road, it doesn't need to do the full probe method. Instead, this vehicle could use just the Y value from the road's rail data. Another performance enhancement is to orient the vehicles that are far enough from the player using only the road's rail-orientation vectors. This approach works well when small vehicle-orientation pops are not noticeable.

Managing the collision state. When an ambient vehicle collides with the player, or with a dynamic or static obstacle in the city, the ambient vehicle switches from using a partially simulated physics model to a fully simulated physics model. The fully simulated model allows the ambient vehicle to act correctly in collisions.

A vehicle manager controls the activities of all the vehicles transitioning between physics models. A collision manager handles the collision itself. For example, once a vehicle has come to rest, the vehicle manager resets it back to the partially simulated physics model. At this point, the ambient vehicle attempts to plot a spline back to the road rail. As it proceeds along the rail, the vehicle will not perform any obstacle detection, and will collide with anything in its way. A collision then sends the vehicle back to the collision manager. This loop will repeat for a definable number of tries. If the maximum number of tries is reached, the ambient vehicle gives up and remains in its current location until the population manager places it back into the active bubble of the ambient vehicle pool.

Using an obstacle-avoidance grid. Every AI entity in the game is assigned to a cell in the obstacle-avoidance grid. This assignment allows fully simulated physics vehicles to perform faster obstacle avoidance.

Since the road is defined by a list of vertices, these vertices make natural separation points between obstacle-avoidance buckets. Together, these buckets divide the city into a grid that limits the scope of collision detection. As an ambient vehicle moves along its rail, crossing a boundary between buckets causes the vehicle to be removed from the previous bucket and added to the new bucket. The intersection is also considered an obstacle bucket.

Simulation bubbles for ambient traffic. A run-time parameter specifies the total number of ambient vehicles to create in the city. After being created, each ambient vehicle is placed into an ambient pool from which the ambients around the player are populated. This fully simulated region around the player is the simulation bubble. Relative to the locations of the player, remote regions of the city are outside of the simulation bubble, and are not fully simulated.

When a player moves from one cull room to another, the population manager compares the vertex list of the new cull room against the list for the old one. From these two lists, three new lists are created: New Roads, Obsolete Roads, and No Change Roads. First, the obsolete roads are removed from the active road list, and the ambient vehicles on them are placed into the ambient pool. Next, the new roads are populated with a vehicle density equal to the total vehicle length divided by the total road length. The vehicle density value is set to the default value based on the road type, or an exception value set through the definition of the race AI map file.

As the ambient vehicles randomly drive around the city, they sometimes come to the edge of the simulation bubble. When this happens, the ambient vehicles have two choices. First, if the road type is two-way (that is, ambient vehicles can drive in both directions), then the vehicle is repositioned at the beginning of the current road's opposite direction. Alternatively, if the ambient vehicle reaches the end of a one-way road, the vehicle is removed from the road and placed into the pool and thereby becomes available to populate other bubbles.

Driving in London: left becomes right. London drivers use the left side of the road instead of the right. To accommodate this situation, some changes have to be made to the raw road data. First, all of the right lane data must be copied to the left lane data, and vice versa. The order of each lane's vertex data must then be reversed so that the first vertex becomes the last, and the lane order reversed so that what was the lane closest to the road's centerline becomes the lane farthest from the center.

Given these changes, the rest of the AI entities and the ambient vehicle logic will work the same regardless of which side of the road the traffic drives on. This architecture gave us the flexibility to allow left- or right-side driving in any city.

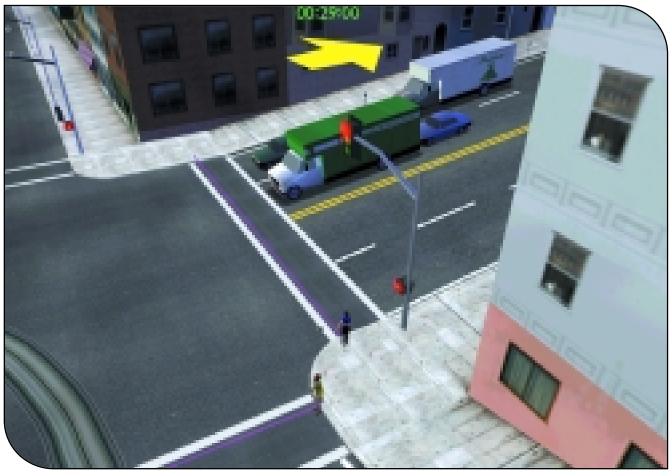
City People: Simulating Pedestrians

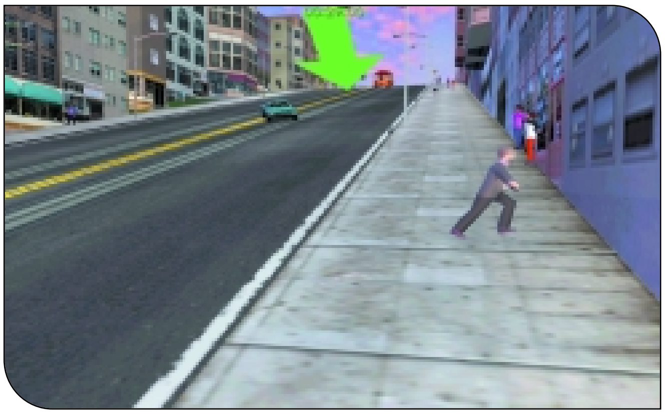
In real cities, pedestrians are on nearly every street corner. They walk and go about their business, so it should be no different in the cities we create in our games. The pedestrians wander along the sidewalks and sometimes cross streets. They avoid static obstacles such as mailboxes, streetlights, and parking meters, and also dynamic obstacles such as other pedestrians and the vehicles controlled by the players. And no, players can't run over the pedestrians, or get points for trying! Even so, interacting with these "peds" makes the player's experience as a city driver much more realistic and immersive.

Simulation bubbles for pedestrians. Just as the ambient traffic has a simulation bubble, so do the pedestrians. And while the pedestrian bubble has a much smaller radius, both types are handled similarly. During initialization, the pedestrians are created and inserted into the pedestrian pool. When the player is inserted into the city, the pedestrians are populated around him. During population, one pedestrian is added to each road in the bubble, round-robin style, until all the pedestrians in the pool are exhausted.

Pedestrians are initialized with a random road distance and side distance based on an offset to the center of the sidewalk. They are also assigned a direction in which to travel and a side of the street on which to start. As the pedestrians get to the edge of the population bubble, they simply turn around and walk back in the opposite direction from which they came.

Wandering the city. When walking the streets, the pedestrians use splines to smooth out the angles created by the road subsegments. All the spline calculations are done in 2D to increase the performance of the pedestrians. The Y value for the splines is calculated by probing the polygon the pedestrian is walking on in order to give the appearance that the pedestrian is actually walking on the terrain underneath its feet.





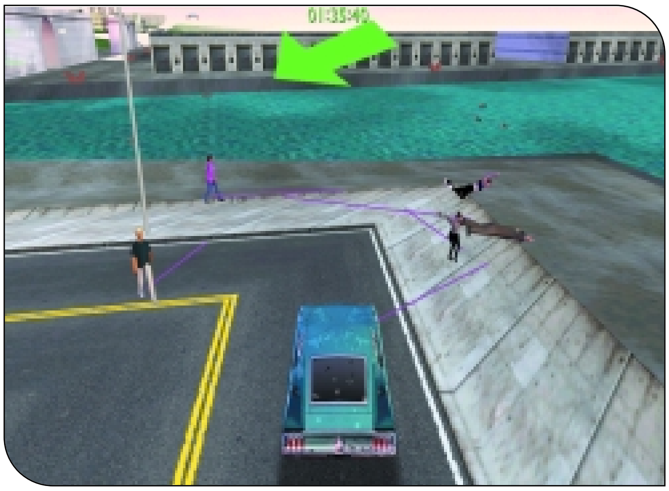


FIGURE 3 (above). In this situation, the `PreCrossStreet` state has moved the pedestrians next to the street curb, and now the `WaitToCrossStreet` state is holding the peds in place until the light turns green. FIGURE 4 (top right). When the oncoming player vehicle threatens these pedestrians, they decide to hug the wall after sending out a probe and finding a wall nearby. FIGURE 5 (bottom right). The pink lines visualize the direction the peds intend to walk. When a player vehicle introduces a threat, the pedestrians decide to dive right or left at the last moment, since no wall is nearby.

Each pedestrian has a target point for it to head toward. This target point is calculated by solving for the location on the spline path three meters ahead of the pedestrian. In walking, the ped will turn toward the target point a little bit each frame, while moving forward and sideways at a rate based on the parameters that control the animation speed. As the pedestrian walks down the road, the ped object calculates a new spline every time it passes a sidewalk vertex.

Crossing the street. When a pedestrian gets to the end of the street, it has a decision to make. The ped either follows the sidewalk to the next street or crosses the street. If the ped decides to cross the street, then it must decide which street to cross: the current or the next. Four states control ped navigation on the streets: `Wander`, `PreCrossStreet`, `WaitToCrossStreet`, and `CrossStreet` (see Figure 3). The first of these, `Wander`, is described in the previous section, "Wandering the City." `PreCrossStreet` takes the pedestrian from the end of the street to a position closer to the street curb, `WaitToCrossStreet` tells the pedestrian waiting for the traffic light that it's time to cross the street, and `CrossStreet` handles the actual walking or running of the pedestrian to the curb on the other side of the street.

Animating actions. The core animation system for the pedestrians is skeleton-based. Specifically, animations are created in 3D Studio Max at 30FPS, and then downloaded using Angel's proprietary exporter. The animation system accounts for the nonconstant nature of the frame rate.

For each type of pedestrian model, a data file identifies the animation sequences. Since all the translation information is removed from the animations, the data file also specifies the amount of translation necessary in the forward and sideways directions. To move the pedestrian, the ped object simply adds the total distance multiplied by the frame time for both the forward and sideways directions. (Most animation sequences have zero side-to-side movement.)

Two functions of the animation system are particularly useful. The `Start` function immediately starts the animation sequence specified as a parameter to the function, and the `Schedule` function starts the desired animation sequence as soon as the current sequence finishes.

Avoiding the speeding player. The main rule for the pedestrians is to always avoid being hit. We accomplish this in two ways. First, if the pedestrian is near a wall, then the ped runs to the wall, puts its back against it, and stands flush up against it until the threatening vehicle moves away (see Figure 4). Alternatively, if no wall is nearby, the ped turns to face the oncoming vehicle, waits until the vehicle is close enough, and then dives to the left or right at the very last moment (see Figure 5).

The pedestrian object determines that an oncoming vehicle is a threat by taking the forward directional vector of the vehicle and performing a dot product with the vector defined by the ped's position minus the vehicle's position. This calculation measures the side distance. If the side distance is less than half the width of the vehicle, then a collision is imminent.

The next calculation is the time it will take the approaching vehicle to collide with the pedestrian. In this context, two distance zones are defined: a far and a near. In the far zone, the pedestrian turns to face the vehicle and then goes into an "anticipate" behavior, which results in a choice between shaking with fear and running away. The near zone activates the "avoid" behavior, which causes the pedestrian to look for a wall to hug. To locate a wall, the pedestrian object shoots a probe perpendicular to the sidewalk for ten meters from its current location. If a wall is found, the pedestrian runs to it. Otherwise, the ped dives in the opposite direction of the vehicle's rotational momentum. (Sometimes the vehicle is going so fast, a superhuman boost in dive speed is needed to avoid a collision.)

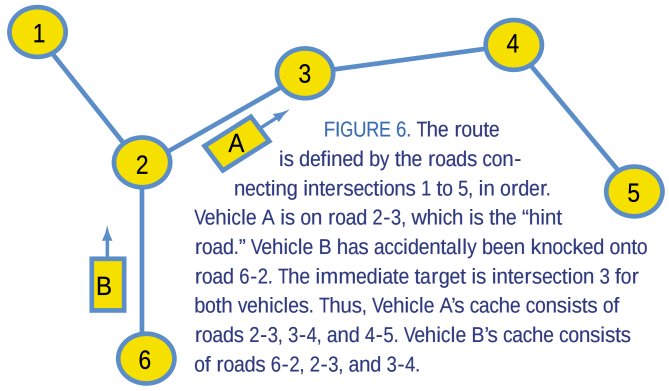


FIGURE 6. The route is defined by the roads connecting intersections 1 to 5, in order. Vehicle A is on road 2-3, which is the "hint road." Vehicle B has accidentally been knocked onto road 6-2. The immediate target is intersection 3 for both vehicles. Thus, Vehicle A's cache consists of roads 2-3, 3-4, and 4-5. Vehicle B's cache consists of roads 6-2, 2-3, and 3-4.

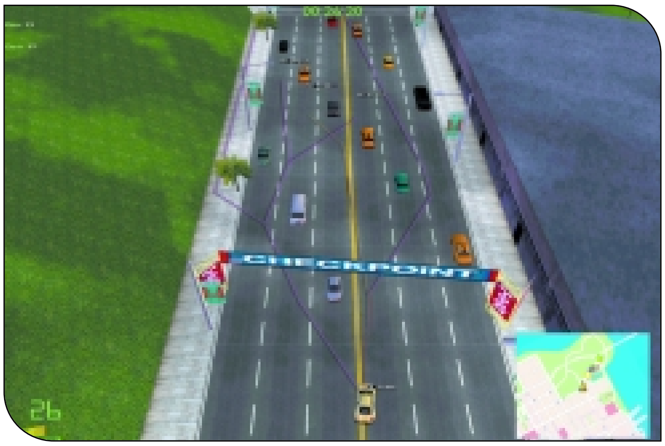


FIGURE 7. The purple lines on the road show the tree of possible routes that this opponent vehicle is considering. The orange line shows the best route — which is typically the one that isn't blocked, stays on the road, and goes as straight as possible.

Avoiding obstacles. As the pedestrians walk blissfully down the street, they come to obstacles in the road. The obstacles fall into one of three categories: other wandering pedestrians; props such as trash cans, mailboxes, and streetlights; or the player's vehicle parked on the sidewalk.

In order to avoid other pedestrians, each ped checks all the pedestrians inside its obstacle grid cell. To detect a collision among this group, the ped performs a couple of calculations. First, it determines the side distance from the centerline of the sidewalk to itself and the other pedestrian. The ped's radius is then added to and subtracted from this distance. A collision is imminent if there is any overlap between the two pedestrians.

In order to help them avoid each other, one of the pedestrians can stop while the other one passes. One way to do this is to make the pedestrian with the lowest identification number stop, and the latter ped sets its target point far enough to left or right to miss the former ped. The ped will always choose left if it's within the sidewalk boundary; otherwise it will go to the right. If the right target point is also past the edge of the sidewalk, then the pedestrian will turn around and continue on its way. Similar calculations to pedestrian detection and avoidance are performed to detect and avoid the props and the player's vehicle.

Simulating Vehicles with Full Physics

The full physics simulation object, `VehiclePhysics`, is a base class with the logic for navigating the city. The different entities in the city are derived from this base class, including the `RouteRacer` object (some of the opponents) and the `PoliceOfficer` object (cops). These child classes supply the additional logic necessary for performing higher-level behaviors. We use the term "full-physics vehicles" because the car being controlled for this category behaves within the laws of physics. These cars have code for simulating the engine, transmission, and wheels, and are controlled by setting values for steering, brake, and throttle. Additionally, the `VehiclePhysics` class contains two key public methods, `RegisterRoute` and `DriveRoute`.

Registering a route. The first thing that the navigation algorithm needs is a route. The route can either be created dynamically in real time or defined in a file as a list of intersection IDs. The real-time method always returns the shortest route. The file method is created by the Race Editor, another proprietary in-house tool that allows the game designer to look down on the city in 2D and select the intersections that make up the route. The game designer can thereby create very specific routes for opponents. Also, the file method eliminates the need for some of the AI entities to calculate their routes in real time, which in turn saves processing time.

Planning the route. Once a route to a final destination has been specified, a little bit more detailed planning is needed for handling immediate situations. We used a road cache for this purpose, which stores the most immediate three roads the vehicle is on or needs to drive down next (see Figure 6).

At any given moment, the vehicle knows the next intersection it is trying to get to (the immediate target), so the vehicle can identify the road connecting this target intersection with the intersection immediately before the target. If the vehicle is already on this "hint road," then the cache is filled with the hint road and the next two roads in the route.

If the vehicle isn't on the hint road, it has gotten knocked off course. In this situation, the vehicle looks at all the roads that connect with the intersection immediately before the target. If the vehicle is on one of these roads, then the cache is filled with this road and the next two roads the vehicle needs to take in order to get back on track. If the vehicle isn't on any of these roads, then it dynamically plots a new route to the target intersection.

Determining multiple routes. If there are no ambient vehicles in the city, then there is only one route necessary to give to an opponent (the computer-controlled player, or CCP), the best route. In general, however, there is ambient traffic everywhere that must be avoided if the CCP is to remain competitive. The choice then becomes which path to pick to avoid the obstacles. At any given moment, this choice comes down to going left or right to avoid an upcoming obstacle. As the CCP plans ahead, it determines two additional routes for each and every obstacle, until it reaches the required planning distance. This process produces a tree of routes to choose from (see Figure 7).

Choosing the best route. When all the possible routes have been enumerated, the best route for the CCP can be determined. Sometimes one or more of the routes will take the vehicle onto the sidewalk. Taking the sidewalk is a negative, so these routes are less attractive than those which stay on the road. Also, some routes will become completely blocked, with no way around the obstacles present, making those less attractive as well. The last criterion is minimizing the amount of turning required to drive a path. Taking all these criteria into account, the best route is usually the one that isn't blocked, stays on the road, and goes as straight as possible.

Setting the steering. The CCP vehicle simulated with full physics uses the same driving model that the player's vehicle uses. For example, both vehicles take a steering parameter between –1.0 and 1.0. This parameter is input from the control pad for the player's vehicle, but the CCP must calculate its steering parameter in real time to avoid obstacles and reach its final destination. Rather than planning its entire route in advance, the CCP simplifies the problem by calculating a series of Steering Target Points (STPs), one per frame in real time as gameplay progresses. Each STP is simply the next point the CCP needs to steer towards to get one frame closer to its final destination. Each point is calculated with due consideration to navigating the road, navigating sharp turns, and avoiding obstacles.

Setting the throttle. Most of the time a CCP wants to go as fast as possible. There are two exceptions to this rule: traversing sharp turns and reaching the end of a race. Sharp turns are defined as those in which the angle between two road subsegments is greater than 45 degrees, and can occur anywhere along the road or when traversing an intersection. Since the route through a sharp turn is circular, it is easy to calculate the maximum velocity through the turn by the formula

$$V = \sqrt{ugR}$$

where V is equal to the velocity, u is the coefficient of friction for the road surface, g is the value of gravity, and R is the radius of our turn. Once the velocity is known, all that the CCP has to do is slow down to the correct speed before entering the turn.

Getting stuck. Unfortunately, even the best CCP can occasionally get stuck, just like the player does. When a CCP gets stuck, it throws its car into reverse, realigns with the road target, and then goes back into drive and resumes the race.

The Road Ahead

In the wake of the original MIDTOWN MADNESS, we wanted open city racing to give players much more than the ability to drive on any street and across any open area. In order for a city to feel and play in the most immersive and fun way possible, many interactive entities of real cities need to be simulated convincingly. These entities include racing opponents, tenacious cops, ambient traffic, and pedestrians, all of which require powerful and adaptive AI to bring them to life. MIDTOWN MADNESS 2 and MIDNIGHT CLUB expand on the capabilities of these entities, which in turn raises the bar of players' expectations even further.

The future of open city racing is wide open — literally. Angel Studios and I are planning even more enhancements to the AI in any future games of this type that we do. Some ideas I'm planning to investigate in the future include enhancing the opponent navigation skills of all AI entities, and creating AI opponents that learn from the players. Additionally, I'd like to create more player interaction with the city pedestrians, and have more interaction between AI entities. Anyone wanna race?