Final Project: Character Motion Planning

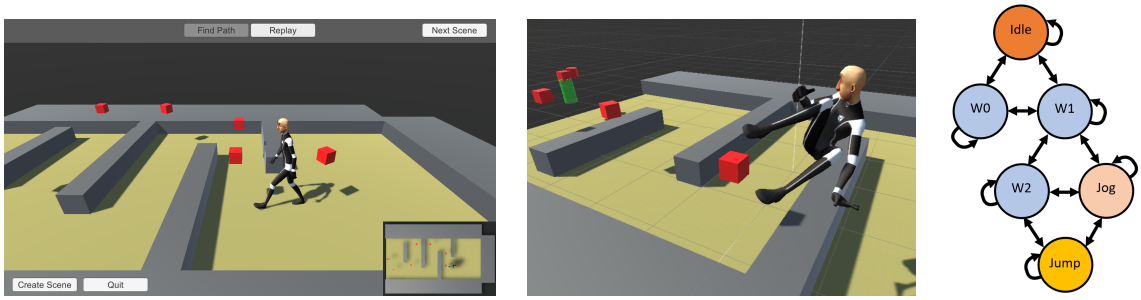
Ryan Canales

Given start and end positions, an environment, and a set of possible motions, this planning algorithm will search for an optimal sequence of motions for the character to undertake to reach the goal. The core of the planning algorithm is the A* shortest path search. In this project, it is used to search for a sequence of optimal "states" the character should take to reach a goal position. In this implementation, the states consist of the character's position, it's orientation, the time the state occurs, and a motion. The character can perform the following four motions to reach its goal: idle, walking, jogging, and jumping. In an early implementation, the probabilistic roadmap method (PRM) was used to sample a set of possible positions. However, I found this limiting and not ideal for the goal of the project, so I switched to implementing a method based on the behavior planning algorithm by Lau and Kuffner [1].

1 Implementation

1.1 Animation Preparation

One of the first things to do was obtain and prepare animation files to assign to each desired motion. The animation files that I used in this project are motion captured clips from the "Raw Mocap Data for Mecanim" asset. This asset was available through the Unity asset store but seems to have been removed within the past year. After selecting the appropriate motions from the mocap data set, some manual adjustments were made to them. All adjustments were made through Unity's interface for editing animations. First, the start and end times of each animation were adjusted so that the motions would loop smoothly. To help make the loop seamless, an option called "Loop Pose" was enabled for each motion. Next, the rotation of the root for each animation had to be adjusted so that the average velocity of the animation would point in front of the character.



(a) (b) (c)

Figure 1: (a) Example scene showing the planned path (red blocks), the walking character, and the UI. (b) The character in mid jump over one of the obstacles. (c) The finite state machine used to generate child states. This has no relation to the animation state machine used by Unity. W0, W1, and W2 represent the walking motion at three speeds, with W0 being the slowest and W2 the fastest.

1.2 Animation Control

In order to playback, loop, and blend between different motions in Unity, the animation state machine (called "Animator") must be used. The state machine I made is simple and consists of two connected states: a blend tree and a jump node. The blend tree is used to blend between the walking and jogging motions. The blend parameters used by the blend tree and when the animator exits the blend tree and triggers the jumping motion is based on the current state of the character (which is determined by the planner). In other words, the planning algorithm controls the animation state machine by determining which motion to play at what time and at what speed. Another point to clarify is that the state machine for playing back animations in Unity is not related to how the planner searches for a sequence of animations to play. In the following sections, I will describe the implementation of the behavior planner by the C# class name.

1.3 Planner

This class is the main class for the behavior planner. It implements the A* search algorithm and controls the playback of the animations. First, a starting state is created based on the characters position and orientation in the scene, and a goal state is created based on a given goal position. Then, in the function "FindPath", A* is used to search for an optimal sequence of states. If such a sequence is found, the function returns the sequence and playback begins. The cost of exploring a given state is the current state's cost plus the cost of the new motion (jogging and jumping have higher costs). The heuristic is the euclidean distance of a state from the goal.

Playback is achieved by iterating through the sequence of states obtained from "FindPath". At each frame during playback, the character is incrementally moved and rotated to reach the position and orientation described by the next state in the sequence. Once the next state is reached (i.e. the motion for the state has completed), a state index is incremented, and the process is repeated until the final state is reached.

1.4 State

This class contains the position, orientation (in degrees), motion, and timestamp (seconds). The position is stored as a 2D vector as the character only moves on the XZ plane. In addition to these parameters, additional information is stored, including the playback speed for the animation and the blend parameters to be passed to the blend tree. The playback speed is one by default, meaning it plays the animation at it's regular speed. As indicated in the state machine diagram (Figure 1 (c)), the walking motion can be played at three speeds. This class also contains several member functions, with the notable functions described in the following paragraphs.

GetChildStates: This function returns a list of possible states that can be obtained from a given state. The children are determined by the parent state's motion and the environment at the time each child state would occur. The motions that can be taken from a parent state and their associated transition costs are chosen according to the finite state machine shown in Figure 1 (c) (the transition costs are not shown as they are user defined). Next, child states are created for each motion. The position of a child state is based on the position of the parent state plus the velocity of the animation multiplied by it's length (in seconds). The velocity of each possible animation is rotated relative to the parent state's orientation from -90 to 90 degrees in intervals of 15 degrees, resulting in 13 new states. If the current animation being evaluated is the walking motion, the velocity is additionally scaled to 0.5 and 1.5 times the default velocity for each rotation. The orientation of a child state is the parent state's orientation plus the orientation used to generate the child state's position, which is from the set of orientations used to rotate the velocity. The timestamp for the child state is the parent state's timestamp plus the length of the child motion. Finally, each generated child state is checked

for collisions. The child states that result in no collisions are added to the list of possible states to explore.

Collision: Checks for collisions with obstacles given a state and a list of obstacles. Returns true if a collision is detected.

Equal: Checks the equality of two states. Returns true if the orientations and positions for the two states are equal.

1.5 Obstacle

This class describes a rectangular obstacle that lies on the XZ plane (ground). The obstacle can be static or moving. The movement of the obstacle is limited to horizontal movement along the global X-axis or movement along it's local X-axis. In both cases, the obstacle stays on the ground. The speed at which the obstacle moves and how far it moves can be set using the "speed" and "range" variables. The movement is a simply back and forth along whichever axis is chosen. All obstacles have the same height and their widths and lengths can be adjusted.

Collision: A simple disk-rectangle intersection test is used for collision checking. The character is represented as a fixed radius disk on the XZ plane. When collision testing for a static obstacle, the radius of the character, it's initial state, and it's new state need to be known. The disk at the new position is checked for intersections with the current obstacle. If none occur and the motion being performed is not a jump, then the edge between the initial position and the new position is checked for intersections. If no intersections were found in both cases, then no collision with the current obstacle is reported. If the current obstacle is moving, new final positions and obstacle positions are sampled at a regular interval between the timestamp of the initial state and the next state. At each timestep, the circle-rectangle intersection test is performed. If no intersection occurs during the time between the initial timestamp and the new timestamp, then no collision is reported.

2 Results

The character happily moves collision-free through the environment to the goal position. Although the motion of the character is often not natural, the main goal of the project was achieved. The build of the project, which is for Windows, includes 4 example scenes and a simple scene editor that allows the user to move the goal position and insert obstacles.

3 Challenges and Limitations

The transitions between states is not natural looking. This is because the movement of the character and it's rotation are simple interpolations between states. The speed of the character's translation and how quickly it rotates to a new orientation does vary depending on the animation playing. However, there is significant foot sliding when the state has been incremented because the character is rotated towards the new state's orientation while simultaneously being translated toward the next position.

A lot of time was spent attempting to make the motion look more natural. I tried adding more walking directions to the blend tree, allowing Unity to handle the characters movement, and various other smooth interpolation methods. Each technique I tried resulted in more natural looking walking motions, but the character would not exactly follow the sequence of states. This was because the speed of the character and it's angular velocity varied while interpolating between motions, leading to a different trajectory than that predicted by the planner. Another side effect is that the character would collide with obstacles because of the discrepancy between the speed of the character during playback and that expected by the planner. A further reason I abandoned these methods is that none of them helped make the transition from walking/jogging to jumping look any more natural.

The aforementioned issues with motion interpolation are, in part, due to a crucial limitation in the planner. That is, the planner assumes that the character moves in a straight line at a constant velocity from one state to the next. In order to achieve natural motion transitions and character movement, both the planner and the playback of the motions would require reworking. Due to time constraints (and for the sake of my sanity), I had to quit working on this aspect of the project.

4 Conclusion and Future Work

In conclusion, the basics of the project are achieved. The character can move from an initial position to a goal position while avoiding both static and moving obstacles. However, as described in the previous section, the movement of the character is not exactly natural. Future work could involve addressing the naturalness problem. Although this is one of the more challenging tasks, it is one worth tackling when designing a character motion planner. Other than that, this planner could be extended to include more motions, more complex obstacle movement, a varying terrain (as opposed to a flat plane), and 3D collision detection.

References

[1] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, page 271–280, New York, NY, USA, 2005. Association for Computing Machinery.