Jefferson TDR

Section 11 : Movement
Jamie Doornbos
9/18/97
Revision 2.

11. Movement

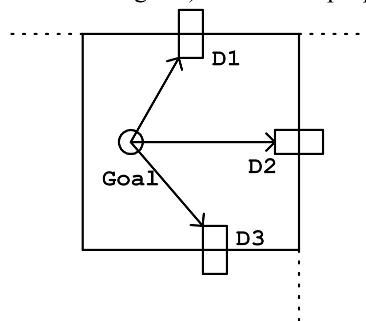
This section describes how people move around in the world. At the high-level, primitives are issued from the person's behavior. The primitive handlers invoke routines to find the best door for room-to-room routes, find the best path around obstacles inside a room, and move the person along line segments.

11.1 Routing. This section describes a path is found from a person's current location to a destination. First of all, this is split into two stages: getting into the room of the goal and getting to the goal. If the room numbers of the start and goal match, the within-room routing is used, section 11.1.2. If not, the room-to-room routing is used, section 11.1.1. (for how rooms are assigned, see Rooms, Section 2.4).

11.1.1 Room-to-room routing. If the start and goal lie in different rooms, the portal tree of the "best" door object is pushed with interrupt onto the person's stack (see also Doors, section 11.4). The portal tree is expected either to get the person into the opposite room of the door, or fail. To find the best door, recursion is used to assign a score to each door in the world. The score represents how close the door is to the goal. Currently doors are enumerated by finding all objects of type "kDoor".

11.1.1.1 Best-Door Algorithm summary.

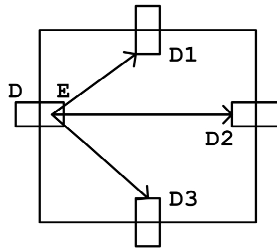
11.1.1.1.1 Initialization and top-level (routine "cXObject :: MakeRoute"). First, find an unused route ID for the person finding the route and stash the value in the route ID field of the person. (See 11.4.2 Score Table in Doors section.) An unused route id is found by taking the first one that is not used by any person. From here on, any mention of a door's score is implied to be its score for this route ID. (See section 11.4, Doors, for a more detailed explanation of route ids.) Then set the score for every door in the world to 0 to signify that it is not yet connected to the goal. Let D1…DN be all the doors in the same room as the goal. Then for each door, Di, recursively propagate to Di with a score of MAXSCORE - floor[distance(Di, goal)]. See below for what happens in this recursive step. "floor" is the integer floor function. MAXSCORE is a constant that is high enough not to produce final negative scores. (MAXSCORE is actually cXDoor::kBestRouteScore. Currently its value is 200. TBD TODO: should be higher.) This initial propogation is demonstrated in the figure below.



Let E1..EM be the doors in that same room as the start. Then for each door, Ei, subtract the distance(Start, Ei) from the its score. This is necessary to take into account the position of the starting point within its room. Then the best door is the door with the highest score and in the same room as the start.

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

11.1.1.1.2 Recursion. (routine "cXDoor :: PropogateRoute" sp.). Propagate a route to a door, D, with score, S: If the score S is less than the score of D, just return. This is the base case of the recursion, i.e. how it is known that this door has been previously visited. Let E be the opposite side of D. Set the score of D and E to S. Let D1..DN be all doors in the same room as E. For each door, Di, propagate the route on Di with score S - floor[distance(E, Di)] - 1. The -1 is used to make sure the score always goes down when a room is passed over, since the floor function does not guarantee this for close doors. This propogation is demonstrated in the figure below.



11.1.1.2 Example.
In figure 1, there are four rooms. Doors A,B,C,D and E connect the rooms. Also shown for reference are the distances between doors in the same room and start and goal. Assume for simplicity that MAXSCORE is 100. The enumeration order for doors is alphabetical. This is analogous to the order of the object list.

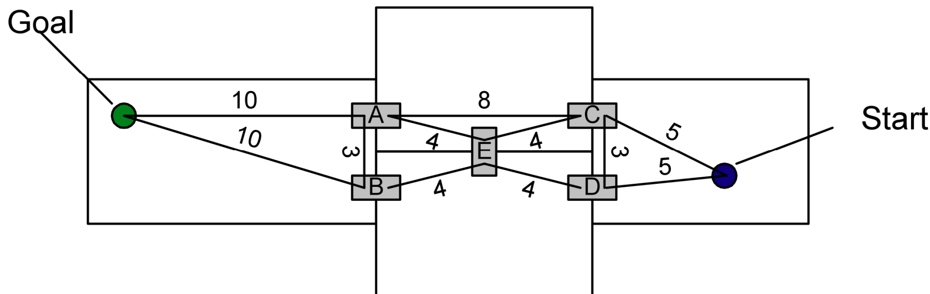


Figure 1. A sample room map.

The algorithm starts out by propagating the route on A, the first door in the same room as the goal. Door A gets a score of 90, since it is 10 units away from the goal. Propagation continues to C, with a score of 81 (90 from score of A minus 8 for distance from A to C minus 1 for passing through a room). Similarly, C propagates to D with a score of 77, which then propagates to B with 67. When B attempts a propagation to A, A rejects the lower score of 63, terminating the recursion. Then D propagates to E with 72, which propagates to A and C, both of which reject the lower score. The result of the process so far is shown in figure 2. In the recursion diagram on the right, the gray nodes represent unvisited branches, and the inequalities show why the recursion was terminated.

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

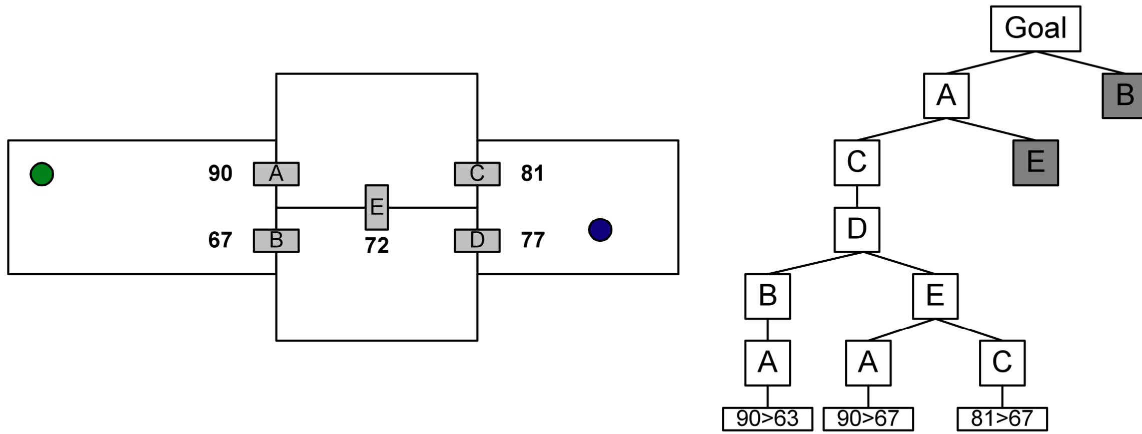


Figure 2. Intermediate scores and partially complete recursion tree beginning "Goal-A-C".

The recursion on C being complete, the propagation continues from A to the next door, E, with score 85 (90 minus 4, the distance from A to E, minus 1 for the room). This score beats the previous 72, so it is replaced and propagation continues to B with a score of 80. B takes on the better score of 80, then propagates to A, which rejects the lower score of 76. Similarly, E propagates to D with a score of 80. D adopts this better score, and propagates to C, which rejects the lower score of 76. The results after this recursion are shown in figure 3.

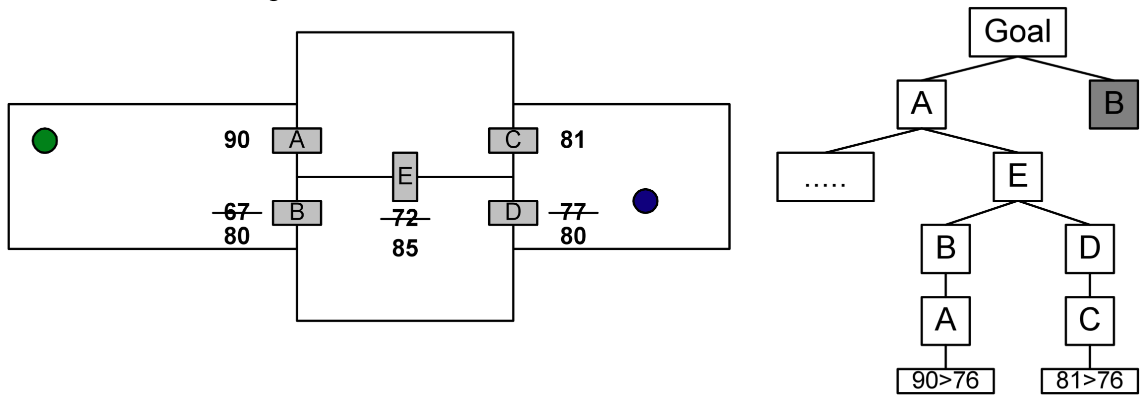


Figure 3. Intermediate scores and partially complete recursion tree beginning "Goal-A-E". (Previous left branch omitted for brevity).

The next step continues back in the initial room search. B is found and propagates with score 90, since it is 10 units away from the goal, and 90 is better than the previous score of 80. B propagates on to D with a score of 81 = 90 - 8 - 1. This wins over the previous 80. D propagates to C, which rejects the lower score of 77, completing the recursion of B on D. Then B propagates to E with a score of 85, which beats the previous score of 85 (yeah, I know. See "inefficiency"). E propagates losing scores to both A and C, completing the recursive process. Figure 4 shows the final numbers.

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

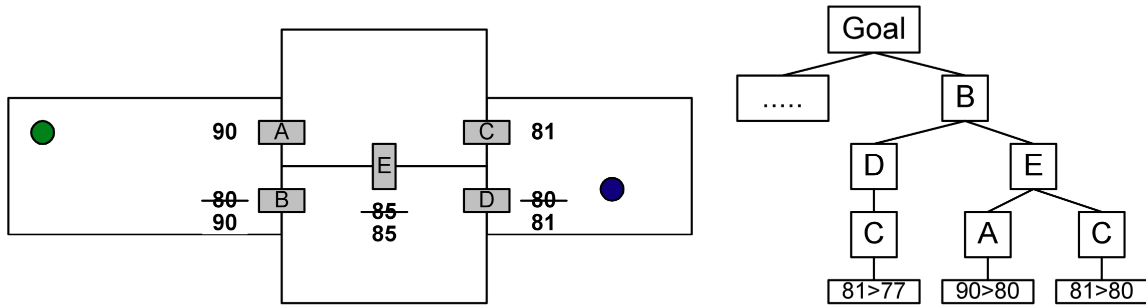


Figure 4. Final recursive result.

Finally, the distance to start is subtracted from the scores of C and D since they are in the same room as the start. The best door is a tie between C and D. These final scores are shown in Figure 5. With the current code, the algorithm would choose C because it is first in the object list.

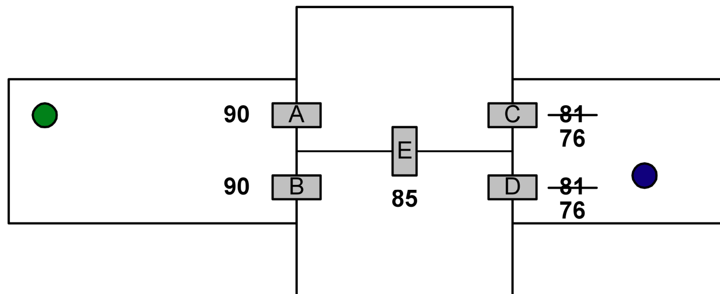


Figure 5. Final result after subtracting distance to start.

11.1.1.3 Inefficiencies. TBD

The current algorithm has some possible inefficiencies, though in the grander scheme, I doubt very much time is spent here.

1. Enumerating the doors. Currently this is done with a straight traversal of the object list, testing the type of each one. Keeping a list of doors separate from objects would probably be a big improvement.
2. Each door that can be arrived at by more than one path is usually visited at least twice. This is because the "depth first" recursion will usually find the worst path first, generating an initial low score for non-direct paths. To fix this, the score assignment could be done in two passes, one to reject and assign, and another to recurse, effectively a "breadth first" recursion. This would allow quicker rejection of doors in rooms that had already been visited. This would be faster if the time to enumerate all the doors does not outweigh the eliminated recursive steps.
3. Another way to improve inefficient paths being found first would be to terminate recursion once a door in the starting room is reached. This would still find all connected doors and would only limit the re-visiting when there are multiple direct routes.
4. TBD TODO: An oversight in the algorithm lets propagation continue when a tie occurs. This appears to be completely unnecessary.

11.1.1.3 Fudging Distances.

A straight line distance is used when scoring doors, and not the route planning distance. This is mainly to save cycles, since "within-room routing" would be excessive. At worst, a non-optimal door will be chosen.

11.1.1.4 Improvements. TBD.

With this algorithm, people will possibly go outside through a side door to get to another room in the house. This could easily be fixed with an individual score subtraction value for doors. Also, doors may be locked, so they may need to be especially discarded.

11.1.1.5 Failure

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

If the portal tree for a door fails, the score of that door is set to 0, and the door with the highest score and in the same room as the person is chosen. If no more doors are available with non zero scores, the route is failed overall.

11.1.2 Within Room routing. Once a person is in the right room, he is moved step by step towards the destination. First, the current wall and room grid and set of objects determine obstacles that the person cannot walk over. These blockages are passed into a low-level route planner in the form of a list of rectangles. The planner does two things: uses a rectangular covering technique together with an A-star algorithm to find the shortest path to the destination, and secondly, constructs the shortest possible path through the chain of rectangular areas. The route planner returns a list of points, which are then followed to reach the destination.

11.1.2.1 Obstacles. The first step in finding the route is to determine where not to step.

11.1.2.1.1 Outer boundary: the room + 1 tile. (routine XPath :: BuildRoomBounds). Using the room of the starting point, the minimal room box is found. This is expanded by 1, and then four 1 tile thick rectangles around the box are added to the obstacle list.

11.1.2.1.2 Wall scan (routine XPath :: BuildRoomBounds). The area within the outer boundary is scanned (top to bottom and left to right) for tiles that are not routable. A routable tile is defined to be one with the same room as the starting room, or a door style. When a non-routable tile is found, it is tested for intersection with existing obstacles. If it intersects, the scan is advanced to its right edge. If not, a rectangle is created for this tile, then expanded to include as many non-routable tiles as possible to the right and then down. TBD: this doesn't account for stray walls, so it will probably change to add walls to the list, not just other rooms.

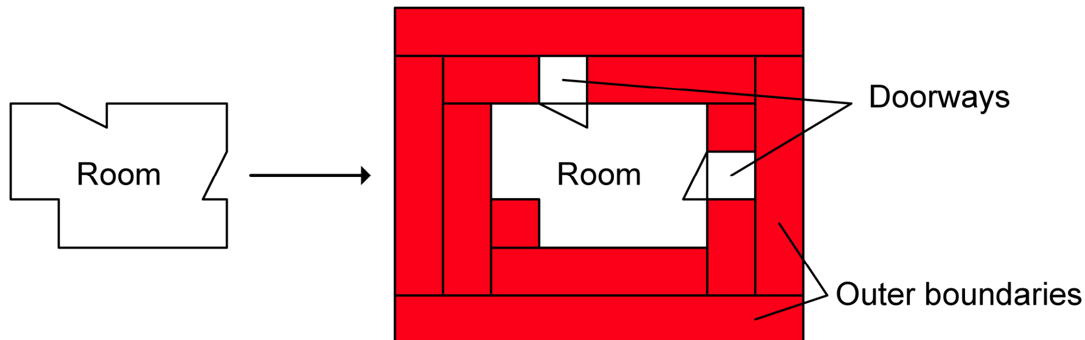


Figure: how a room may look after steps 1 and 2 in 11.1.2.1. Solid rectangles are the generated room bounds.

11.1.2.1.1 Object scan. (routine XPath :: FindPath). Object rectangles are added to the obstacle list as follows:

1. The starting object is skipped.
2. The destination object is skipped if the "route near destination" parameter is set. (This parameter is used to find a route to a locus of points which is the boundary of a rectangle around a destination point).
3. Objects in other rooms are skipped.
4. Objects contained in other objects are skipped, since the parenting object is assumed to produce an obstacle at the location.
5. Objects that can be walked over are skipped.
6. People are skipped if the "skip person" parameter is set and they are within the "width of ignorance" parameter.
7. All others are added.

11.1.2.2 A-Star. The low-level route planner uses the A Star algorithm (see W. Bryan Stout. Computer Game Developer's Conference Proceedings. 1996). In Jefferson, the class "Path" consolidates the low level route planning. A-Star uses the concept

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

of "nodes" and their "successors" to sniff out the shortest path. A Path object performs the route planning step by step with the "AdvancePath" method so the results at each step can be easily examined.

11.1.2.2.1 Nodes. A "node" is essentially a possible location for the routing object. It's "successors" are all nodes that can be reached from it. In this version of the algorithm, rectangles are used as the nodes.

11.1.2.2.2 Inputs and outputs. The inputs to the low-level A-Star algorithm are the list of obstacles, instructions about routing near the destination or on top of it, the width of the routing object, and various tuning values. The tuning values may be viewed in the Demo App, see section 11.1.2.2.5. The output is a list of points (the structure used is STD::vector). Internally, each obstacle rectangle is inflated by the width of the routing object to reduce the problem to that of routing a point, which makes intersection tests more efficient. The particular parameters used in Jefferson are consolidated in class "XPath", which configures the parameters to the low-level based on game structures.

11.1.2.2.3 Dynamic nodes. The main cool feature of this version of the algorithm which saves a lot of time is that successors to a node are found dynamically by maximally expanding seed rectangles around the node. This process uses the current set of nodes and the obstacles to limit the expansion. Other parameters, such as the maximum expanded dimensions, are also available for limiting the expansion. The set of nodes generated for the bounded room is referred to as a "rectangular cover".

11.1.2.2.4 Penalties. Currently the obstacle list is separate from the node list created by the A-Star search. Since the obstacles are rectangles just like nodes, this is theoretically like having two discrete penalty states for a node: zero and infinity. The A-Star algorithm is good at taking into account penalties and finding the overall best path. TBD: Merging the obstacle list in with the node list using a penalty field with maximum value to denote impassable nodes. This will enable terrain penalty boundaries to be used as a node expansion limit and different penalties assigned to different types of terrain. This will also enable rare "jumps" if a high, but not infinite, penalty is used on objects that can be jumped over. However, the details of this with respect to animation and registration are frightening.

11.1.2.2.5 Demo app. Please see the route testing application ($/routetest in the Jefferson/CTG source safe data base) for a visual demonstration of all the parameters, and the A Star process. The app can load and save route files (extension rte) that contain all the low level parameters. Currently the game writes out "routefile.rte" every time the route planner is called. TBD: Add a routes directory and come up with a naming scheme for depositing game routes here. Make a menu check to enable and disable it.

11.1.2.3 Smoothing. The second task of the low level route planning is to construct the final list of points from the shortest node path. Since the node path is one of touching rectangles, this is just choosing a point on each intersection of adjacent nodes, and including the start and goal points. A trivial algorithm for this is apparent, which is to choose the upper left point on each intersection. The algorithm actually used is one that examines the effects of perturbing each point by a distance in both directions along the intersection, which is always a line segment. If the perturbations produce a valid point which decreases the total route length, the new point is substituted for the old one. The process is repeated until no increases are found. Then the perturbation value is lowered and the process repeats. When the perturbation value is zero, smoothing is complete. This process can also be observed in the route testing application.

11.1.2.4 Route following. Once the list of points is found, they must be followed. A walking person keeps track of a current sub-goal which is an index into the list of found points. Each tick, a step is taken toward the sub-goal. If the person is within

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

one step, he is moved to the sub-goal and the sub-goal incremented. The route is done when the final point is reached. If, at any step, the person cannot be placed at the next desired position, the whole route finding process is repeated from the top level and a route counter decremented. If the route counter is zero, then the routing has failed. The route following is controlled at the primitive level, so a behavior with a routing instruction is needed to get a person to move. TBD: this might be too constraining: we may want to have people perform routing without executing a behavior tree. TBD: the route following may need to be buffed up. Some issues and ideas for this are:

1. Sharp turns. Smoke-and-mirrors smoothing of the corners using circular, polynomial, or Bezier curves. Basically this would have to be tweaked, possibly per person, so that legs do not appear to move through obstacles.
2. Abrupt start and stop. Near the start and goal, change the step size along a ramp to simulate acceleration.
3. Abrupt turning. Use an angular velocity value to turn when the desired rotation is more than a threshold value away from the current one, instead of just setting the rotation.
4. Frozen in walk position at unanticipated stops. Look ahead for blockage in the path, so re-routing can be attempted and/or an animation seam can be planned. This could be quite a hard problem.
5. Colliding with idle people. Idle people blocking a route can be modally pushed aside if there is room, while running a shove animation or something.
6. Colliding with moving People. Stall for a while.
7. Head-on collision deadly embrace. Generate a new route for each person with blocking on the space just to the left of him.

11.3 Primitives. A person is routed when a "Goto Relative" or "Delta Move" primitive is issued from a behavior tree.

11.3.1 "Goto Relative." This very common primitive incorporates all of the route finding and following into a single instruction. It attempts to moves a person to a destination specified relative to the stack object (see Simulation chapter for explanation of stack object). Any single side of the object may be chosen, or a special "anywhere near" value can be used, which will allow any valid route that ends a certain distance from the object. Also specified is a maximum number of times to try to reroute. TBD: this has not proven very useful, and is typically left at the default by the tree programmers. Each time the routing is attempted, if the route planner fails, people near the destination are eliminated from the obstacle list. This was done to help avoid crowding at doors, which is where a person would be left standing on an inter-room route that fails after a door has been reached. TBD: this was a quick hack and made only marginal improvements and a more long term solution may be needed.

11.3.2 "Delta Move." Moves a person a specified number of fractional tile positions in their current direction. This primitive works by just putting two points into the person's point list instead of planning a route, and then using the same move code as goto relative. Currently this primitive is only used by doors to get through them once the person has routed successfully to one side.

11.4 Doors.

11.4.1 Object Class. A door is a special C++ subclass, cXDoor, of the multi-tile object class, cXMTObject.

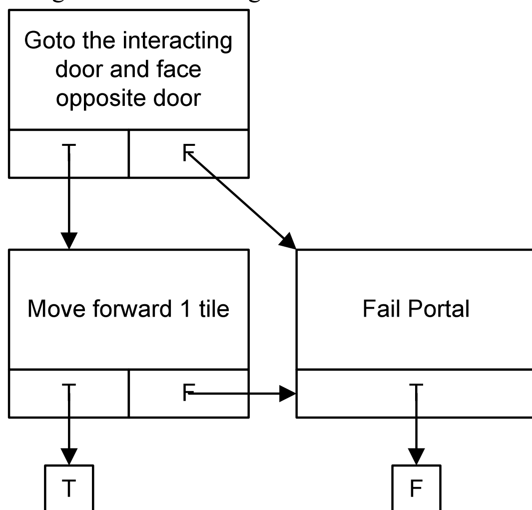
11.4.2 Score Table. The only member added by class cXDoor is a routing score table. This maps route ID numbers to scores. Currently, this uses the TreeTable class where "motive advertisements" are replaced with "route ids". [TBD: This is a strange re-use of code. If anything needs to change about the way route scores are set, like going to floating point scores, this will probably become a normal array.] The number of availabe route id's is a constant, currently set to 8. A person finding a door-to-door route requests a new route id and

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

stores it in a state variable. A new route id is calculated by finding the lowest route id that is not in use by any person. Currently the score tables are not saved in the save file, so a person in a newly opened house may repeat some door-related things that happened just before saving. TBD: the score tables probably need to be saved.

11.4.3 Portals. Doors represent the only instance of an abstract portal concept. TBD: Stairways are expected to fit into this concept as well. A portal is currently defined as a two-tile object with a tile in each of two rooms. TBD: this will probably need to be modified to account for double doors or stairways with more than two tiles. TBD: Windows may also be portal objects so people can climb through them.

11.4.4 Portal tree. A portal has a special behavior tree, a portal tree, specified in the object definition (see also Object Definition, 3.2.1) which is expected to move a person from one room to the other or fail. When a person needs to route to something in another room, a door selection algorithm is run and the best door chosen. The basic door portal tree is typically something like the following:



TBD: this algorithm may need to be modified to consider all portals if the type of a portal is not always "kDoor". Then the portal tree of the door is pushed with interrupt onto the stack above the node containing the goto relative primitive. Note: this feature is the primary reason for the interrupt flag in the stack lines. TBD: The need for this may be eliminated if primitives are given their own stack lines.

11.4.4.1 Successful completion. When a portal tree completes successfully, the popping of the interrupted stack line should leave the person in the original goto relative primitive, which will then repeat the portal pushing process until the rooms of the start and goal coordinates match.

11.4.4.2 Failure. When a portal tree fails, it is expected to set the trap count data field of the person to a negative number, the negative of the object id of the portal object that failed. Then, when the stack line is popped and the goto relative primitive handler is called again, it can tell if a portal failure occurred and set the score of that object to zero. TBD: this protocol is a state-stingy hack that reuses a variable that previously had a clear purpose. Another state variable, something like "failed portal object" would make it much more clear what is happening. Possibly even a primitive to set the failed portal object.

Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.