GIUSEPPE TURINI

CS-485 ADVANCED GAME DEVELOPMENT

NAVIGATION AND PATHFINDING

WORLD OF THIEVES

HIGHLIGHTS

Introduction to Navigation and Pathfinding

The A* Pathfinding Algorithm

- A Simple Example of the A* Algorithm in 2D
- Pseudocode and Time Complexity of the A* Algorithm

A* Pathfinding in Unity

- Main Scripts and Settings for the A* Implementation in Unity
- Implementation of the A* Algorithm and Data Structures in C# and Unity

Path Following and Waypoints Obstacle Avoidance

- Simple Pathfollowing
- Automatic Waypoints
- Simple Obstacle Avoidance

Navigation Meshes in Unity

- NavMesh Configuration and Baking
- NavMesh Agents and Static and Dynamic Obstacles

NAVIGATION AND PATHFINDING

Sometimes a character needs to travel automatically from its location to a destination.
In some cases, this will be a simple movement in a straight line or along a preset path.
However, there are games where the route to the target is not so direct.
In these games, some control logic is needed to allow the character to find the route.

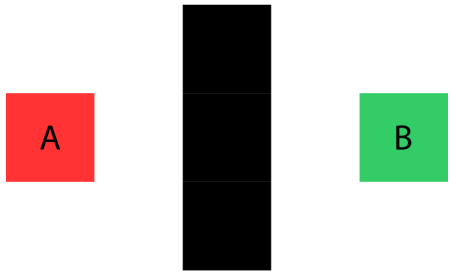
The process of choosing character movements in games is known as *navigation*.

Navigation techniques may involve an element of trial and error.
However, it is preferred to make the character plan its path intelligently (best route).

The AI task performed to establish the optimal way is known as *pathfinding*.

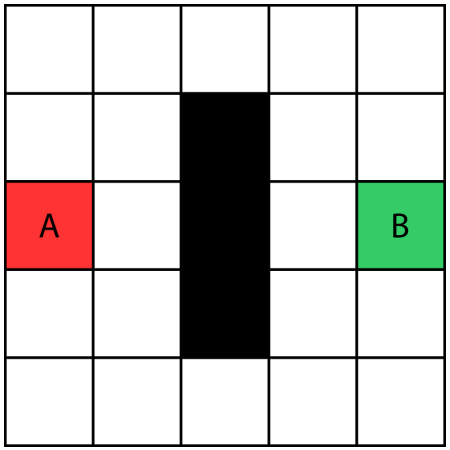
THE A* PATHFINDING ALGORITHM 1

A* (A Star) is apathfinding algorithm widely used in games.
It is popular mainly because of its performance and accuracy.



A* Simple 2D Example - Part 1

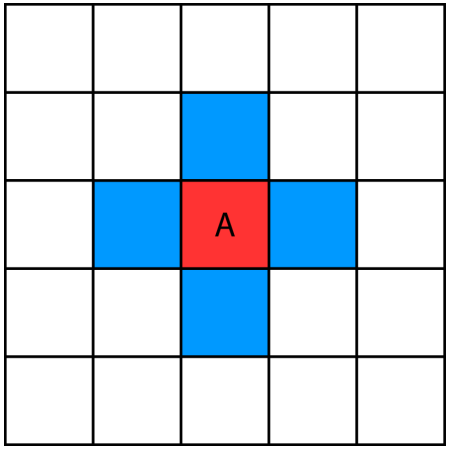
We want our unit to move from point A to B.
There is an obstacle in the way.
So we cannot move from A to B in a straight way.
We need to find a way from A to B around the obstacle.



We need to describe the map and the obstacles.
Discretize the map splitting it into 2D tiles (or 3D cells).
2D tiles can be quads or hexagons or triangles etc.
Now our map is a 5x5 2D grid.

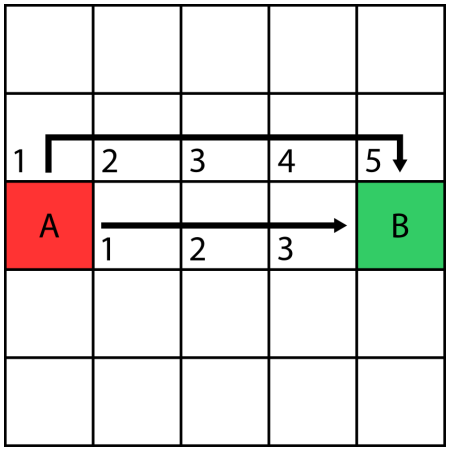
THE A* PATHFINDING ALGORITHM 2

A* Simple 2D Example - Part 2



Search for the best path to reach the target.
Calculate the move-score of each tile adjacent to the start.
Choose the tile with the lowest move-score.
4 adjacent tiles (if we don't consider diagonal movements).
We need to values to compute the move-score of a tile:

G: cost from start to the tile (lower left).
H: cost from the tile to the target (lower right).
F: final move-score equal to G+H (top left).



In this example we use the Manhattan Length.
The distance is the number of tiles to travel.
So G increases by 1 everytime we move toward the target.

THE A* PATHFINDING ALGORITHM 3

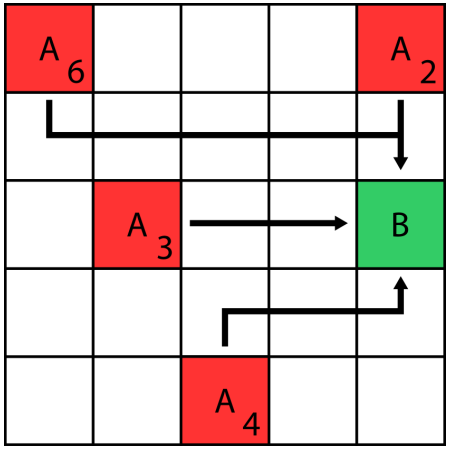
A* Simple 2D Example - Part 3

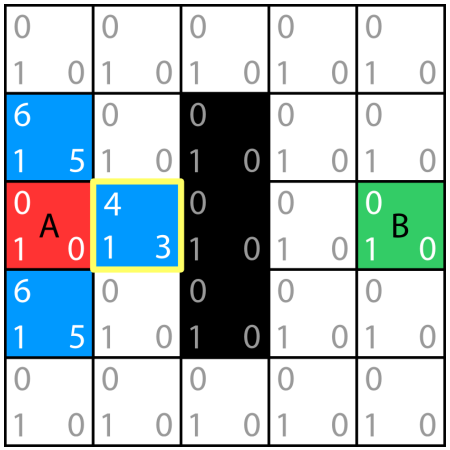
We can give different costs to different tiles.

H is computed in the same way.

Understanding G and H we can start looking for the path.
At first choose the starting tile.
Then determine the valid adjacent tiles.
Then calculate G and H and F for each valid adjacent tile.

Then select the tile with the lowest F (or move-score).
Choose this tile as our next move.
Store the previous tile as the parent of the selected tile.



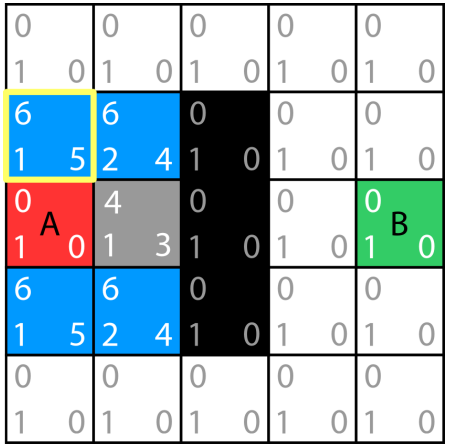


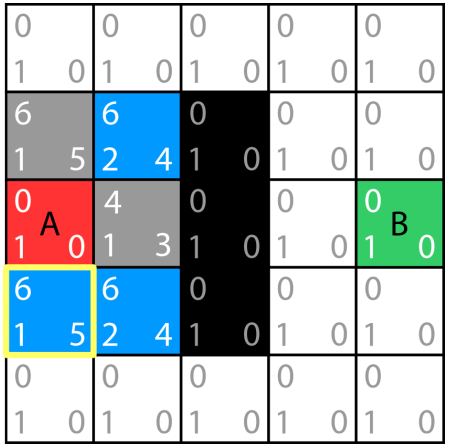
THE A* PATHFINDING ALGORITHM

A* Simple 2D Example - Part 4

Repeat the computation of G and H for valid adjacent tiles.

Repeat the entire process until we reach the target tile.

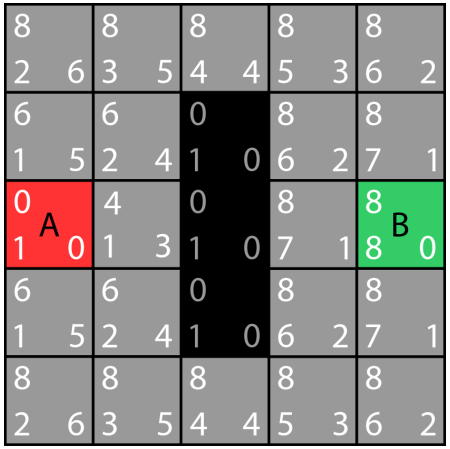


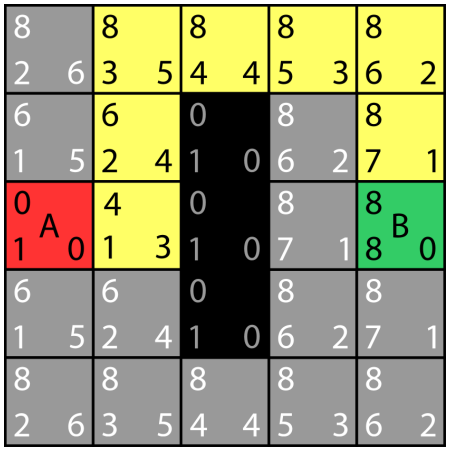


THE A* PATHFINDING ALGORITHM 5

A* Simple 2D Example - Part 5

Once we reach the target we can reconstruct the path.
Trace back from the target tile using the parents.





THE A* PATHFINDING ALGORITHM 6

A* Pseudocode - Part 1

```
function AStar(start, goal) {
    closedSet = emptySet; // Set of nodes already evaluated.
    openSet = {start}; // Set of tentative nodes to be evaluated.
    cameFrom = emptyMap; // Map of navigated nodes.
    gScore[start] = 0; // Cost from start along best known path.
    // H: Heuristic cost estimated from start to goal.
    fScore[start] = gScore[start] + hScore(start, goal);
    while( openSet != emptySet ) {
        current = node in openSet with lowest fScore;
        if( current == goal ) { return reconstructPath(cameFrom, goal); }
        remove current from openSet;
        add current to closedSet;
        ...
```

THE A* PATHFINDING ALGORITHM 7

A* Pseudocode - Part 2

```
        ...
        foreach( neighbor in neighborNodes( current ) ) {
            if( neighbor in closedSet ) { continue; }
            tentativeGScore = gScore[current] + dist(current, neighbor);
            if( ( neighbor not in openSet ) ||
                ( tentativeGScore < gScore[neighbor] ) ) {
                cameFrom[neighbor] = current;
                gScore[neighbor] = tentativeGScore;
                fScore[neighbor] = gScore[neighbor] + hScore(neighbor, goal);
                if( neighbor not in openSet ) { add neighbor to openset; }
            }
        }
    }
    return failure;
}
```

THE A* PATHFINDING ALGORITHM 8

A* Pseudocode - Part 3

```
function reconstructPath(cameFrom, currentNode) {
    if( currentNode in cameFrom ) {
        p = reconstructPath(cameFrom, cameFrom[currentNode]);
        return ( p + currentNode ); }
    else { return currentNode; } }
```

A* Time Complexity

The time complexity of A* depends on the heuristic H.

In the worst case, the num of nodes visited is exponential in the shortest path length.

However, this is polynomial when:

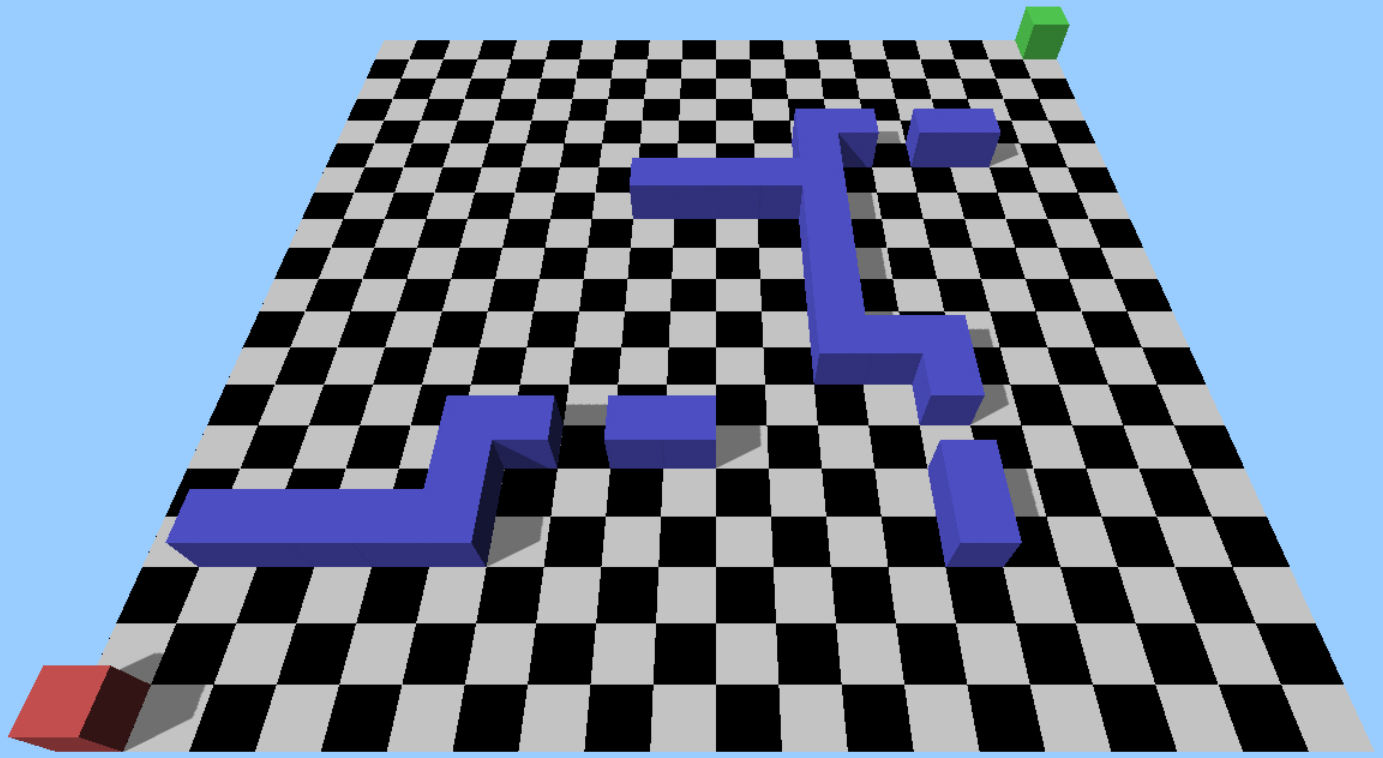
the search space is a tree,

there is a single goal state, and

the heuristic func H meets this condition: $|H(x) - H^*(x)| = O(\log H^*(x))$

where H* is the optimal heuristic (i.e. the exact cost to get from x to the goal).

A* PATHFINDING IN UNITY 1



A* PATHFINDING IN UNITY 2

Implementation of the A* Pathfinding Algorithm in Unity - Part 1

2 scripts for the implementation of the 2D grid required to run the A* algorithm.

- The AStarGridManager.cs script as the manager of the 2D grid.
- The Node.cs script as the class describing a 2D grid node.

2 scripts for the implementation of the A* pathfinding algorithm and data structures.

- The AStar.cs script as the implementation of the A* pathfinding algorithm.
- The PriorityQueue.cs as the data structure to properly select nodes.

1 script to test the implementation.

- The AStarTest.cs script as the tester of the A* pathfinding algorithm.

3 tags to mark: the "Start" object, the "End" object, and the "Obstacle" objects.

A* PATHFINDING IN UNITY 3

Implementation of the A* Pathfinding Algorithm in Unity - Part 2
See the Node.cs script.
See the IComparable interface: msdn.microsoft.com/en-us/library/system.icomparable

```
public class Node : IComparable {
    public float nodeTotalCost; // Total cost so far for the node.
    public float estimatedCost; // Estimated cost from the node to the goal.
    public bool bObstacle; // Does the node is an obstacle or not.
    public Node parent; // Parent of the node in the linked list.
    public Vector3 position; // Position of the node.

    public Node() {
        estimatedCost = 0.0f; nodeTotalCost = 1.0f;
        bObstacle = false; parent = null; }

    public Node( Vector3 pos ) {
        estimatedCost = 0.0f; nodeTotalCost = 1.0f;
        bObstacle = false; parent = null; position = pos; }
```

A* PATHFINDING IN UNITY 4

Implementation of the A* Pathfinding Algorithm in Unity - Part 3

```
    ...

    // Make the node to be noted as an obstacle.
    public void MarkAsObstacle() { bObstacle = true; }

    // The CompareTo method affects the Sort method.
    // This comparison uses the estimated total cost between two nodes.
    public int CompareTo( object obj ) {
        Node node = (Node) obj;
        if( this.estimatedCost < node.estimatedCost ) { return -1; }
        if( this.estimatedCost > node.estimatedCost ) { return 1; }
        return 0; }
}
```

A* PATHFINDING IN UNITY 5

Implementation of the A* Pathfinding Algorithm in Unity - Part 4

See the AStarGridManager.cs script.

```
// Grid manager class handles all the grid properties.
public class AStarGridManager : MonoBehaviour {
    public int numOfRows;
    public int numOfColumns;
    public float gridCellSize;
    public bool showGrid = true;
    public bool showObstacleBlocks = true;

    public Node[,] nodes { get; set; }

    // Origin of the grid manager.
    public Vector3 origin;
    public Vector3 Origin { get { return origin; } }

    private GameObject[] obstacleList;
```

A* PATHFINDING IN UNITY 6

Implementation of the A* Pathfinding Algorithm in Unity - Part 5

```
...

// staticInstance is used to cache the instance found in the scene.
private static AStarGridManager staticInstance = null;

// Static instance property to find the manager object in the scene.
public static AStarGridManager instance {
    get {
        if( staticInstance == null ) {
            staticInstance = FindObjectOfType(typeof(AStarGridManager))
                                as AStarGridManager;
            if( staticInstance == null ) {
                Debug.Log( "Could not locate a AStarGridManager." ); } }
        return staticInstance; } }

// Destroy the static instance when the game is stopped in the editor.
void OnApplicationQuit() { staticInstance = null; }
```

A* PATHFINDING IN UNITY 7

Implementation of the A* Pathfinding Algorithm in Unity - Part 6

```
...

// Initialize the grid manager.
void Awake() {
    obstacleList = GameObject.FindGameObjectsWithTag( "Obstacle" );
    CalculateObstacles(); }

// Calculate which cells in the grids are mark as obstacles.
void CalculateObstacles() {
    // Initialize the nodes of the grid.
    nodes = new Node[ numOfColumns, numOfRows ]; int index = 0;
    for( int i = 0; i < numOfColumns; i++ ) {
        for( int j = 0; j < numOfRows; j++ ) {
            Vector3 cellPos = GetGridCellCenter( index );
            Node node = new Node( cellPos ); nodes[ i, j ] = node;
            index++; } }
```

A* PATHFINDING IN UNITY 8

Implementation of the A* Pathfinding Algorithm in Unity - Part 7

```
...

    // Run through the obstacle list and set the obstacle position.
    if( ( obstacleList != null ) && ( obstacleList.Length > 0 ) ) {
        foreach( GameObject data in obstacleList ) {
            int indexCell = GetGridIndex( data.transform.position );
            int col = GetColumn( indexCell );
            int row = GetRow( indexCell );
            // Also make the node as blocked status.
            nodes[ row, col ].MarkAsObstacle(); } } }

// Returns the position of the grid cell in world coordinates.
public Vector3 GetGridCellCenter( int index ) {
    Vector3 cellPosition = GetGridCellPosition( index );
    cellPosition.x += gridCellSize / 2.0f;
    cellPosition.z += gridCellSize / 2.0f;
    return cellPosition; }
```

A* PATHFINDING IN UNITY 9

Implementation of the A* Pathfinding Algorithm in Unity - Part 8

```
...

// Returns the position of the grid cell in a given index.
public Vector3 GetGridCellPosition( int index ) {
    int row = GetRow( index ); int col = GetColumn( index );
    float xPosInGrid = col * gridCellSize;
    float zPosInGrid = row * gridCellSize;
    return Origin + new Vector3( xPosInGrid, 0.0f, zPosInGrid ); }

// Get the grid cell index in the A* grid with the position given.
public int GetGridIndex( Vector3 pos ) {
    if( !IsInBounds( pos ) ) { return -1; }
    pos -= Origin;
    int col = (int)( pos.x / gridCellSize );
    int row = (int)( pos.z / gridCellSize );
    return ( row * numOfColumns + col ); }
```

A* PATHFINDING IN UNITY 10

Implementation of the A* Pathfinding Algorithm in Unity - Part 9

```
...

// Get the row number of the grid cell in a given index.
public int GetRow( int index ) {
    int row = index / numOfColumns; return row; }

// Get the column number of the grid cell in a given index.
public int GetColumn( int index ) {
    int col = index % numOfColumns; return col; }

// Check whether the current position is inside the grid or not.
public bool IsInBounds( Vector3 pos ) {
    float width = numOfColumns * gridCellSize;
    float height = numOfRows* gridCellSize;
    return ((pos.x >= Origin.x) && (pos.x <= (Origin.x + width)) &&
            (pos.x <= (Origin.z + height)) && (pos.z >= Origin.z)); }
```

A* PATHFINDING IN UNITY 11

Implementation of the A* Pathfinding Algorithm in Unity - Part 10

```
...

// Get the neighbour nodes in 4 different directions.
public void GetNeighbours( Node node, ArrayList neighbours ) {
    Vector3 neighbourPos = node.position;
    int neighbourIndex = GetGridIndex( neighbourPos );
    int row = GetRow( neighbourIndex );
    int column = GetColumn( neighbourIndex );
    int leftNodeRow = row - 1; int leftNodeColumn = column; // Bottom.
    AssignNeighbour( leftNodeRow, leftNodeColumn, neighbours );
    leftNodeRow = row + 1; leftNodeColumn = column; // Top.
    AssignNeighbour( leftNodeRow, leftNodeColumn, neighbours );
    leftNodeRow = row; leftNodeColumn = column + 1; // Right.
    AssignNeighbour( leftNodeRow, leftNodeColumn, neighbours );
    leftNodeRow = row; leftNodeColumn = column - 1; // Left.
    AssignNeighbour( leftNodeRow, leftNodeColumn, neighbours ); }
```

A* PATHFINDING IN UNITY 12

Implementation of the A* Pathfinding Algorithm in Unity - Part 11

```
    ...

    // Check if the neighbour is not an obstacle, then assign the neighbour.
    void AssignNeighbour( int row, int column, ArrayList neighbours ) {
        if( ( row != -1 ) && ( column != -1 ) &&
            ( row < numOfRows ) && ( column < numOfColumns ) ) {
            Node nodeToAdd = nodes[ row, column ];
            if( !nodeToAdd.bObstacle ) { neighbours.Add( nodeToAdd ); } } }
}
```

A* PATHFINDING IN UNITY 13

Implementation of the A* Pathfinding Algorithm in Unity - Part 12

See the AStar.cs script.

```
public class AStar {
    public static PriorityQueue closedList, openList;

    // Calculate the final path in the path finding.
    private static ArrayList CalculatePath( Node node ) {
        ArrayList list = new ArrayList();
        while( node != null ) { list.Add( node ); node = node.parent; }
        list.Reverse();
        return list; }

    // Calculate the estimated heuristic cost to the goal.
    private static float HeuristicEstCost( Node currN, Node goalN ){
        Vector3 vecCost = currN.position - goalN.position;
        return vecCost.magnitude; }
```

A* PATHFINDING IN UNITY 14

Implementation of the A* Pathfinding Algorithm in Unity - Part 13

```
...

// Find the path between start and goal nodes using the A* algorithm.
public static ArrayList FindPath( Node start, Node goal ) {
    openList = new PriorityQueue();
    openList.Push( start );
    start.nodeTotalCost = 0.0f;
    start.estimatedCost = HeuristicEstCost( start, goal );
    closedList = new PriorityQueue();
    Node node = null;
    while( openList.Length != 0 ) {
        node = openList.First();
        if( node.position == goal.position ) {
            return CalculatePath( node ); }
        ArrayList neighbours = new ArrayList();
        AStarGridManager.instance.GetNeighbours( node, neighbours );
```

A* PATHFINDING IN UNITY 15

Implementation of the A* Pathfinding Algorithm in Unity - Part 14

```
...

        // Get the neighbours.
        for( int i = 0; i < neighbours.Count; i++ ) {
            Node neighb = (Node) neighbours[i];
            if( !closedList.Contains( neighb ) ) {
                // Cost from current node to this neighbour node.
                float cost = HeuristicEstCost( node, neighb );
                // Total cost so far from start to this neighbour node.
                float totalCost = node.nodeTotalCost + cost;
                // Estimated cost for neighbour node to the goal.
                float neighbNEstCost = HeuristicEstCost( neighb, goal );
                // Assign neighbour node properties.
                neighb.nodeTotalCost = totalCost;
                neighb.parent = node;
                neighb.estimatedCost = totalCost + neighbNEstCost;
```

A* PATHFINDING IN UNITY 16

Implementation of the A* Pathfinding Algorithm in Unity - Part 15

```
...

                // Add the neighbour node to the list,
                // if it is not already in the list.
                if( !openList.Contains( neighbourNode ) ) {
                    openList.Push( neighbourNode ); } } }
        closedList.Push( node );
        openList.Remove( node ); }
    // If finished looping and cannot find the goal then return null.
    if( node.position != goal.position ) {
        Debug.LogError( "Goal not found!" );
        return null; }
    // Calculate the path based on the final node.
    return CalculatePath( node ); } }
```

PATH FOLLOWING 1

Implementation of a Simple Path Following Approach in Unity - Part 1

See the SimplePath.cs script.

```
public class SimplePath: MonoBehaviour {
    public bool isDebug = true;
    public float Radius = 2.0f;
    public Vector3[] pts;

    public float Length { get { return pts.Length; } }

    public Vector3 GetPoint( int index ) { return pts[ index ]; }

    void OnDrawGizmos() {
        if( isDebug ) {
            for( int i = 0; i < pts.Length; i++ ) {
                if( ( i + 1 ) < pts.Length ) {
                    Debug.DrawLine( pts[i], pts[i+1], Color.red ); } } } }
}
```

PATH FOLLOWING 2

Implementation of a Simple Path Following Approach in Unity - Part 2

See the PathFollowing.cs script.

```
public class PathFollowing : MonoBehaviour {
    public SimplePath path;
    public float speed = 20.0f;
    public float mass = 5.0f;
    public bool isLooping = true;
    private float curSpeed; // Actual speed of the unit.
    private int curPathIndex;
    private float pathLength;
    private Vector3 targetPoint;
    private Vector3 velocity;

    void Start() {
        pathLength = path.Length;
        curPathIndex = 0;
        velocity = transform.forward; }
```

PATH FOLLOWING 3

Implementation of a Simple Path Following Approach in Unity - Part 3

```
void Update() {
    curSpeed = speed * Time.deltaTime; // Unify the speed.
    targetPoint = path.GetPoint( curPathIndex );
    // If reach the radius within the path, move to next path point.
    if( Vector3.Distance( transform.position, targetPoint ) <
        path.Radius ) {
        // Don't move the unit if the path is completed.
        if( curPathIndex < ( pathLength - 1 ) ) { curPathIndex++; }
        else if( isLooping ) { curPathIndex = 0; }
        else { return; } }
    if( curPathIndex >= pathLength ) { return; } // Check end point.
    // Calculate the next Velocity towards the path.
    if( ( curPathIndex >= ( pathLength - 1 ) ) && !isLooping ) {
        velocity += Steer( targetPoint, true ); }
    else { velocity += Steer( targetPoint ); }
    transform.position += velocity;
    transform.rotation = Quaternion.LookRotation( velocity ); }
```

PATH FOLLOWING 4

Implementation of a Simple Path Following Approach in Unity - Part 4

```
    // Steering algorithm to steer the vector towards the target.
    public Vector3 Steer( Vector3 target, bool bFinalPoint = false ) {
        // Calculate the direction from current position to target point.
        Vector3 desiredVelocity = target - transform.position;
        float dist = desiredVelocity.magnitude;
        // Normalise the desired Velocity.
        desiredVelocity.Normalize();
        // Calculate the velocity according to the speed.
        if( bFinalPoint && ( dist < 10.0f ) ) {
            desiredVelocity *= curSpeed * ( dist / 10.0f ); }
        else { desiredVelocity *= curSpeed; }
        // Calculate the force Vector.
        Vector3 steeringForce = desiredVelocity - velocity;
        Vector3 acceleration = steeringForce / mass;
        return acceleration; }
}
```

AUTOMATIC WAYPOINTS 1

Implementation of Path Following using Automatic Waypoints in Unity - Part 1

Several waypoints disseminated on your game level.
Each waypoint is a gameobject including the AutomaticWaypoint.cs script.
Each waypoint can be connected to other waypoint using its Connected list.
The Connected list can be updated automatically using the context menu:
    Component Context Menu + Update Waypoints.
    The Connected list will be updated using a line-of-sight criterion.

In the demo a unit patrols the game level following a path (see RobotAI.cs script).
The path has been described by waypoints.
The unit follows the path continuously picking the next waypoint.
    The unit switches waypoint when it is close to the current waypoint.
    The next waypoint is chosen using a minimum-angle criterion.

AUTOMATIC WAYPOINTS 2

Implementation of Path Following using Automatic Waypoints in Unity - Part 2
See the AutomaticWaypoint.cs script.

```
public class AutomaticWaypoint : MonoBehaviour {
    static public AutomaticWaypoint[] waypoints; // List of all waypoints.
    public AutomaticWaypoint[] connected; // List of connected waypoints.
    static public float kLineOfSightCapsuleRadius = 0.1f;

    // Return the closest waypoint to the input position.
    static public AutomaticWaypoint FindClosest( Vector3 pos ) {
        // The closer two vectors, the larger the dot product will be.
        AutomaticWaypoint closest = waypoints[0]; float closestD = 10000.0f;
        if( waypoints.Length == 0 ) { return null; }
        foreach( AutomaticWaypoint w in waypoints ) {
            if( w != null ) {
                float dist = Vector3.Distance( cur.transform.position, pos );
                if( dist < closestD ) { closestD = dist; closest = w; } } }
        return closest; }
```

AUTOMATIC WAYPOINTS 3

Implementation of Path Following using Automatic Waypoints in Unity - Part 3

```
[ ContextMenu( "Update Waypoints" ) ]
void UpdateWaypoints() { RebuildWaypointList(); }

// Search for all waypoints and recalculate connections.
void Awake() { RebuildWaypointList(); }

// Draw the waypoint pickable gizmo.
void OnDrawGizmos() { Gizmos.DrawIcon( transform.position, "WP.tif" ); }
```

AUTOMATIC WAYPOINTS 4

Implementation of Path Following using Automatic Waypoints in Unity - Part 4

```
// Draw the waypoint lines only when you select one of the waypoints.
void OnDrawGizmosSelected() {
    if( waypoints == null ) { return; }
    if( waypoints.Length == 0 ) { RebuildWaypointList(); }
    foreach( AutomaticWaypoint awp in connected ) {
        if( awp != null ) {
            if( Physics.Linecast(   transform.position,
                                    awp.transform.position ) ) {
                Gizmos.color = Color.red;
                Gizmos.DrawLine(    transform.position,
                                    awp.transform.position ); }
            else {
                Gizmos.color = Color.green;
                Gizmos.DrawLine(    transform.position,
                                    awp.transform.position ); } } } }
```

AUTOMATIC WAYPOINTS 5

Implementation of Path Following using Automatic Waypoints in Unity - Part 5

```
// Search for all waypoints and recalculate connections.
void RebuildWaypointList() {
    // * : Please note that FindObjectsOfType() is very slow.
    waypoints = FindObjectsOfType( typeof( AutomaticWaypoint ) )
                    as AutomaticWaypoint[];
    foreach( AutomaticWaypoint awp in waypoints ) {
        awp.RecalculateConnectedWaypoints(); } }
```

AUTOMATIC WAYPOINTS 6

Implementation of Path Following using Automatic Waypoints in Unity - Part 6

```
    // Recalculate waypoint connections.
    void RecalculateConnectedWaypoints() {
        List<AutomaticWaypoint> awpList = new List<AutomaticWaypoint>();
        foreach( AutomaticWaypoint other in waypoints ) {
            if( other == this ) { continue; } // Does not connect to itself.
            // Make sure we have a line of sight.
            if( !( Physics.CheckCapsule(    transform.position,
                                            other.transform.position,
                                            kLineOfSightCapsuleRadius ) ) ) {
                connectionList.Add( other ); } }
        connected = new AutomaticWaypoint[ connectionList.Count ];
        // List<T>.CopyTo() : Copies the entire list to a compatible array.
        connectionList.CopyTo( connected ); }

}
```

AUTOMATIC WAYPOINTS 7

Implementation of Path Following using Automatic Waypoints in Unity - Part 7
See the RobotAI.cs script.
The Patrol() coroutine is launched during the initialization of the script.
The coroutine runs forever, and it is paused/resumed accordingly to the unit actions.

All the animations are managed by the RobotAIAnimation.cs script.
The current animation is chosen accordingly to the current speed of the unit.
See the SendMessage command and the SetSpeed function.

Note: the following are only the relevant parts of this script!

```
void Start() {
    if( ( target == null ) && GameObject.FindWithTag( "Player" ) ) {
        target = GameObject.FindWithTag("Player").transform; }
    StartCoroutine( Patrol() ); }
```

AUTOMATIC WAYPOINTS 8

Implementation of Path Following using Automatic Waypoints in Unity - Part 8

```
// Coroutine to implement the patroling.
IEnumerator Patrol() {
    AutomaticWaypoint curWaypoint =
            AutomaticWaypoint.FindClosest( transform.position );
    while( true ) {
    Vector3 waypointPosition = curWaypoint.transform.position;
    // Are we close to a waypoint? -> Pick the next one!
    if(     Vector3.Distance( waypointPosition, transform.position ) <
            pickNextWaypointDistance ) {
        curWaypoint = PickNextWaypoint( curWaypoint ); }
    // Attack the player and wait until: player dead, or player hidden.
    if( CanSeeTarget() ) {
        yield return StartCoroutine( AttackPlayer() ); }
    // Move towards our target.
    MoveTowards( waypointPosition );
    yield return 0; } }
```

AUTOMATIC WAYPOINTS 9

Implementation of Path Following using Automatic Waypoints in Unity - Part 9

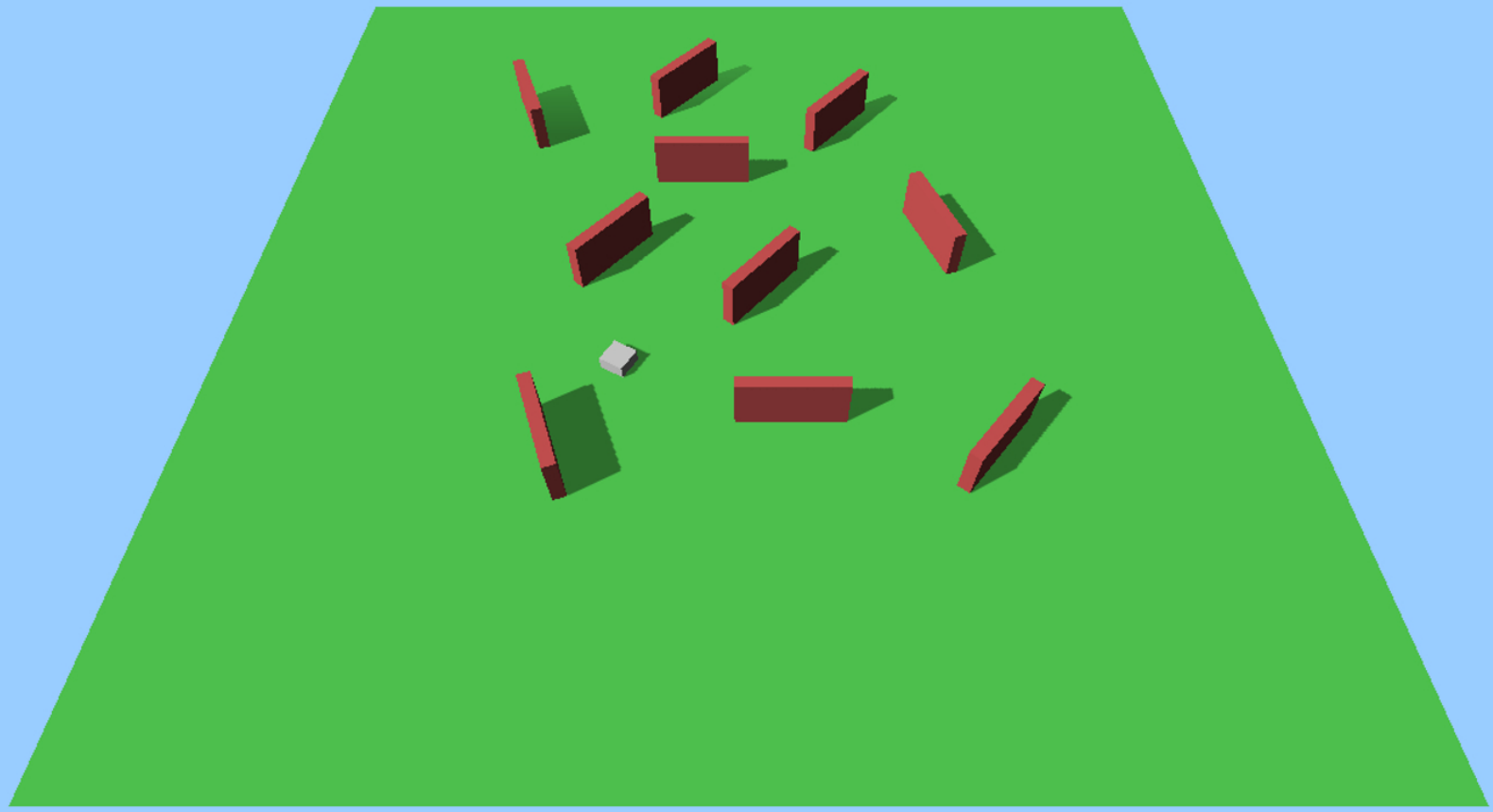
```
// Find the waypoint involving a minimal turn for the unit.
AutomaticWaypoint PickNextWaypoint( AutomaticWaypoint currWaypoint ) {
    // The direction in which we are walking.
    Vector3 forward = transform.TransformDirection( Vector3.forward );
    // The closer two vectors, the larger the dot product will be.
    AutomaticWaypoint best = currentWaypoint; float bestDot = -10.0f;
    foreach( AutomaticWaypoint cur in currentWaypoint.connected ) {
        Vector3 direction = Vector3.Normalize(
                    cur.transform.position - transform.position );
        float dot = Vector3.Dot( direction, forward );
        if( ( dot > bestDot ) && ( cur != currentWaypoint ) ) {
            bestDot = dot; best = cur; } }
    return best; }
```

AUTOMATIC WAYPOINTS 10

Implementation of Path Following using Automatic Waypoints in Unity - Part 10

```
// Move the robot towards input pos.
void MoveTowards( Vector3 pos ) {
    Vector3 dir = pos - transform.position; dir.y = 0.0f;
    // Check direction magnitude, if it is too low set unit speed to 0.
    if( dir.magnitude < 0.5f ) { SendMessage("SetSpeed",0.0f); return; }
    // Rotate towards the target.
    transform.rotation = Quaternion.Slerp( transform.rotation,
            Quaternion.LookRotation( dir ), rotSpeed * Time.deltaTime );
    transform.eulerAngles = new Vector3(0, transform.eulerAngles.y, 0);
    // Modify speed so we slow down when we are not facing the target.
    Vector3 forward = transform.TransformDirection( Vector3.forward );
    float speedModifier = Vector3.Dot( forward, dir.normalized );
    speedModifier = Mathf.Clamp01( speedModifier );
    dir = forward * speed * speedModifier;
    // Move the unit.
    GetComponent<CharacterController>().SimpleMove( dir );
    SendMessage( "SetSpeed", speed * speedModifier ); }
```

OBSTACLE AVOIDANCE 1



OBSTACLE AVOIDANCE 2

Simple Obstacle Avoidance Strategy - Part 1

Unit with the ObstacleAvoidance component attached.
Obstacles requiring a collider component and configured on the Obstacle layer.
See the ObstacleAvoidance.cs script.

```
public class ObstacleAvoidance : MonoBehaviour {
    public float speed = 20.0f;
    public float mass = 5.0f;
    public float force = 50.0f;
    public float minimumDistToAvoid = 20.0f;
    public int obstacleLayer = 8;
    private float curSpeed;
    private Vector3 targetPoint;

    void Start() { mass = 5.0f; targetPoint = Vector3.zero; }

    void OnGUI() { GUILayout.Label( "Click anywhere to move the unit." ); }
```

OBSTACLE AVOIDANCE 3

Simple Obstacle Avoidance Strategy - Part 2

```
void Update() {
    // Unit move by mouse click.
    RaycastHit hit;
    Ray ray = Camera.main.ScreenPointToRay( Input.mousePosition );
    if(Input.GetMouseButtonDown(0)&&Physics.Raycast(ray,out hit,100)) {
        targetPoint = hit.point; }
    // Directional vector to the target position.
    Vector3 dir = targetPoint - transform.position; dir.Normalize();
    AvoidObstacles( ref dir ); // Apply obstacle avoidance.
    // Don't move the unit when the target point is reached.
    if( Vector3.Distance( targetPoint, transform.position ) < 3.0f ) {
        return; }
    curSpeed = speed * Time.deltaTime; // Compute speed.
    Quaternion rot = Quaternion.LookRotation( dir ); // Rotate and move.
    transform.rotation = Quaternion.Slerp( transform.rotation, rot,
                                           5.0f *  Time.deltaTime );
    transform.position += transform.forward * curSpeed; }
```

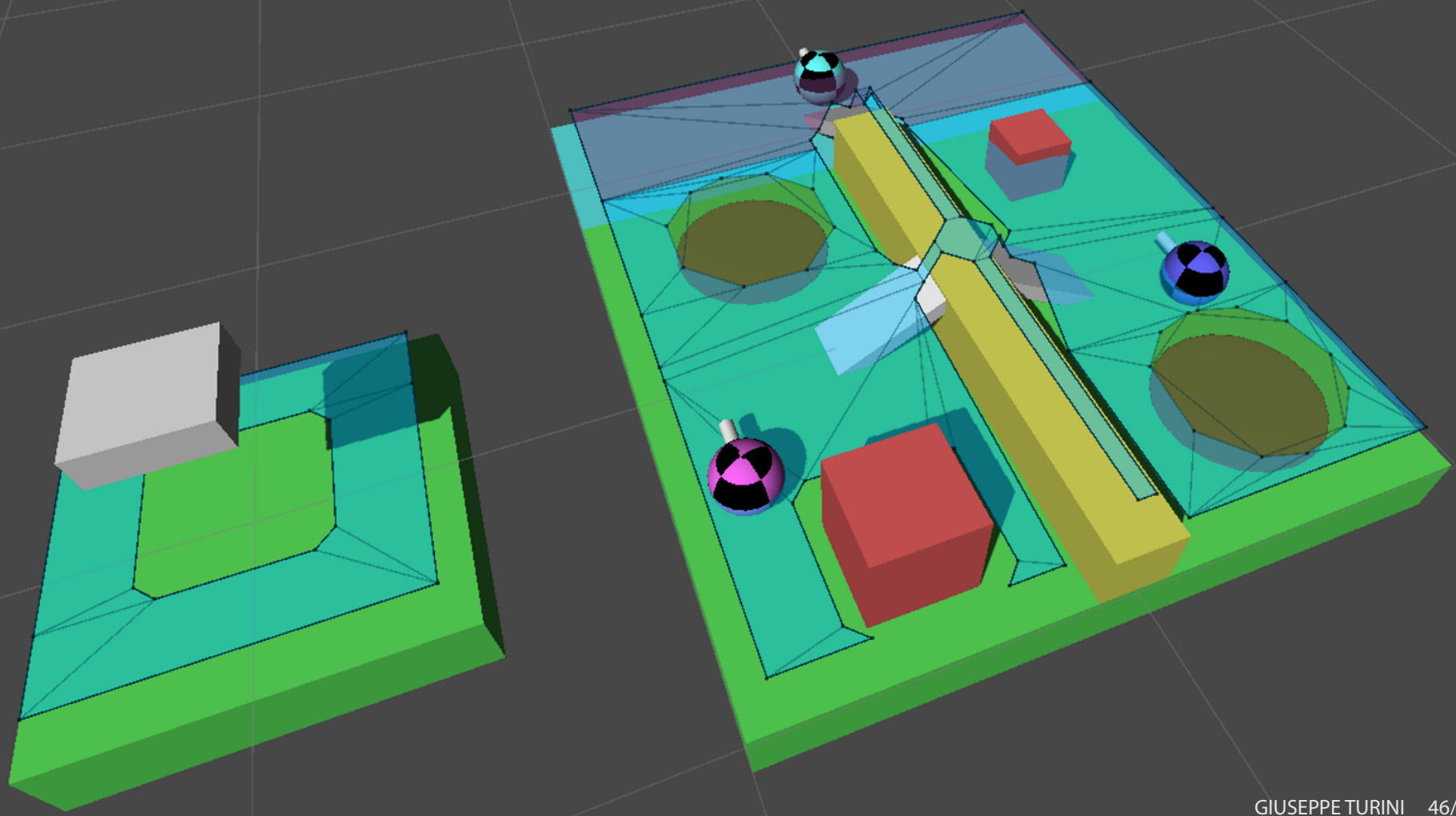
OBSTACLE AVOIDANCE 4

Simple Obstacle Avoidance Strategy - Part 3

```
    // Calculate the new directional vector to avoid the obstacle.
    public void AvoidObstacles( ref Vector3 dir ) {
        RaycastHit hit;
        // Only detect obstacle layer.
        int layerMask = 1 << obstacleLayer;
        // Check unit-obstacles hit using a minimum distance to avoid.
        if( Physics.Raycast(    transform.position, transform.forward,
                                out hit, minimumDistToAvoid, layerMask ) ) {
            // Get the hit point normal to calculate the new direction.
            Vector3 hitNormal = hit.normal;
            hitNormal.y = 0.0f; // Move constrained on the X-Z plane.
            // Get the new directional vector by using force and hit normal.
            dir = transform.forward + hitNormal * force; } }

}
```

NAVIGATION MESHES IN UNITY 1



NAVIGATION MESHES IN UNITY 2

Unity has a built-in pathfinding system based on *navigation meshes* (NavMeshes).

A pathfinding system needs to represent the game parts reachable by a character.
Additionally, it also needs to determine the best route between 2 points.
A special mesh (navigation mesh or NavMesh) is used in Unity to solve both problems.

The reachable area is defined by the polygons of the mesh.
The possible paths are represented by hops between adjacent mesh polygons.

In practice, usually the graphical mesh is too detailed and inefficient for navigation.
So, AI systems usually add to the *"walkable"* mesh a simpler, invisible mesh: a NavMesh.

Luckily, an efficient NavMesh can be generated automatically for a standard floor.
The process of computing a NavMesh automatically is known as *NavMesh baking*.

NAVIGATION MESHES IN UNITY 3

The Unity Editor Navigation Window - Object Section

You should mark objects to be included in the NavMesh as Navigation Static.
You can then set the Navigation Layer to Default if you want the area to be walkable.
Otherwise you can use other 2 default layers: the Not Walkable or the Jump layers.
Furthermore, you can also define additional custom navigation layers.

The Unity Editor Navigation Window - Bake Section

Selecting Bake gives you further options for NavMesh generation on a separate tab.

General settings:
- The Radius is the width of the narrowest gap of a walkable area.
- The Height refers to the min height of a walkable area.
- The Max Slope is the max steepness for a ramp to be walkable.
- The Step Height is the max height of a walkable step or bump.

Once these parameters are set, click the Bake button to generate the NavMesh.
The resulting NavMesh is shown in the Scene as an overlay on the level geometry.

NAVIGATION MESHES IN UNITY 4

Enabling a Character to Navigate

Once the NavMesh is set up, the characters need to be equipped to use it.
The NavMeshAgent component handles both pathfinding and navigation.
In your scripts, navigation is just the setup of the desired destination point.
The NavMeshAgent cannot be accessed directly, so you need to use a reference.

```
NavMeshAgent agent = GetComponent<NavMeshAgent>();
```

To set the character in motion, call the SetDestination.

```
agent.SetDestination( targetPoint );
```

The target point is specified as a point in world space.
The navigation system will find the *"closest point"* on the NavMesh automatically.

NAVIGATION MESHES IN UNITY 5

Navigation Layers and Costs - Part 1

To create the appearance of *"intelligent"* agents, the pathfinding system is designed to search for the *"optimal path"* from the starting point to the destination.

Example: The shortest route between 2 points might be covered in thick snow.
So it could be preferrable to walk along a cleared but longer path.
Example: At night, you might prefer to avoid walking down a dark alleyway.
You may prefer to take a longer but well-lit route for safety.

In both these situations each step along a path has a certain cost associated with it. The cost could represent the difficulty of the terrain, risk or many other factors you might bear in mind when planning a route. The optimal route is therefore the one with the lowest overall cost and this will not always be the shortest.

NAVIGATION MESHES IN UNITY 6

Navigation Layers and Costs - Part 2

Each NavMesh section can be set up with a cost value using Navmesh Layers.
In the Layers tab of the Navigation window, you can see a list of 32 available layers.
3 of these layers are *"built-in"* and reserved by Unity.
You can assign each of the User Layers with a name and a cost value.
And, in the Object tab you can chose a layer for each NavMesh object.

By choosing a layer for an object, you are also assigning the cost per unit of distance an agent travels over it. This enables the design of areas with different traversal properties (water, mud, etc.).

NAVIGATION MESHES IN UNITY 7

Navigation Layers and Costs - Part 3

You can set a layer to have a cost less than the default value of 1, but avoid the usage of negative costs to indicate advantages. The agent, trying to find the lowest-cost path, might stay a lot of time in an area of negative cost to reduce the cost of its current path overall. In these cases you should use a cost greater than one for the *"normal"* layer and a lower cost for the *"advantage"* layer.

The cost of a path is a very abstract concept and could involve many different factors:

Terrain Type:	For example to max the unit speed (mud, grass, asphalt etc.).
Risk:	For example to min the danger (forest with wild animals etc.).
Visibility:	For example to min the detection of the unit (light, dark etc.).
Exposure:	For example to keep the unit covered (close to walls etc.).
Damage:	For example to min the damage affecting the unit (lava, acid etc.).

NAVIGATION MESHES IN UNITY 8

The NavMesh Agent

A NavMesh Agent component can be added to a gameobject to allow the navigation.
It uses information stored in the NavMesh to calculate the optimal path.
Once the path is calculated, the agent moves the gameobject along the path.

The agent is defined by an upright configurable cylinder.
The cylinder moves with the gameobject but always remains upright.
The cylinder defines the region within which obstacles should not intrude.
The agent keeps the proper distance from obstacles accordingly to the cylinder size.
The path computed is a sequence of waypoints connected by straight lines.
The agent follows the path using acceleration and gradual turning.
Usually the agent won't be able to stop precisely on the target point.
An agent will automatically avoid fixed NavMesh Obstacles and other agents.

NAVIGATION MESHES IN UNITY 9

NavMesh Static and Dynamic Obstacles

Fixed (static) obstacles on a NavMesh can be set up as part of the baking process.

Dynamic (moving) obstacles can be added using the NavMesh Obstacle component.
Add it to any gameobject to configure a dynamic obstacle.
The cylinder representing the obstacle (collider) will move together with that object.