Tabelle 1

Überordnung	Funktion	Aufgabe	Parameter	Ausgabe	Verwendung	Wann wird sie ausgeführt?	Wie funktioniert sie?
Konstanten	`constSafeBorder`	extra Sicherheitsradius um Minen, in denen Wegpunkte als „nicht betretbar“ markiert werden	SKALAR		Beim Erstellen des Nodegrids	wird manuell definiert und ist während des Spiels konstant	
	`constGridRadius`	Abstand der Nodes voneinander	SKALAR				
	`constNavSecurity`	Welchen Korridor müssen die Wegpunkte frei von Kollisionen sein, damit sie durch simplifyPath gestrichen werden.	SKALAR		in „simplifyPath“		
	`constWayPointReachedRadius`	ab welchem Abstand soll ein Wegpunkt als „besucht“ markiert werden?	SKALAR		bei der Berechnung der Beschleunigung		
	`constMineProxPenality`	Strafpunkte, die Nodes hinzugefügt werden, die in der Nähe von Minen sind. Je dichter sie an einer Mine liegen, desto höhere Strafpunktzahl	SKALAR		Beim Erstellen des Nodegrids		
	`constCornerBreaking`	wie hoch ist der Faktor, mit dem, abhängig vom Winkel zum darauffolgenden Wegpunkt, berechnet wird, auf welche Geschwindigkeit an einem Wegpunkt abgebremst werden soll	SKALAR				
	`constCompetitionModeThreshold`	gibt an, wann der Gegner und wir dieselbe Tankstelle ansteuern und unser Spcaeball in den competition mode gegen soll					
Statische Variablen	`nodeGrid`	Die Variable, in der das NodeGrid gespeichert ist	CELLARRAY		zur Wegfindung	bei jedem neuen Erstellen des NodeGrids	
	`waypointList`	Eine Liste der als nächstes anzufliegenden Wegpunkte	CELLARRAY		Die Wegpunkte werden von „calculateBES()“ nacheinander abgeflogen	Bei jedem Definieren neuer Wegpunkte	
	`drawHandles`	Die Handles der Zeichenobjekte der Wegpunkte (zum debuggen)				beim Einzeichnen der Wegpunkte	
	`ArrayOfMines`	Eine Kopie des Minenarrays	CELLARRAY		Zum Feststellen des Verschwindens einer Mine	Am Anfang und immer, wenn eine Mine verschwindet	
	`StartNumberOfTank`	Die Anzahl der Tankstellen zu Beginn des Spiels	SKALAR		Zum Festlegen, ab wann ein Spieler genug Tanken hat, um zu gewinnen	Einmal am Anfang	
	`NumberOfTank`	Kopie der Anzahl der Tankstellen					
	`ignoreTanke`	Die Nummer der Tanke, die bei der nächsten Berechnung ignoriert werden soll, da der Gegner sie vor uns erreicht				Immer, wenn eine Tankstelle vom Gegner vor uns erreicht werden wird	
	`tankeCompetition`	TRUE wenn im competition mode	TRUE/FALSE				
Funktionen	`whatToDo()`	Entscheidung über Angriff, Flucht und Tanken	-	-		wird jede Iteration ausgeführt	• Haben wir mehr als die Hälfte aller vorhandenen Tanken getankt oder befinden wir uns in der Nähe des Gegners und haben mehr getankt, als dieser -> **attackEnemy()** • Hat der Gegner mehr als die Hälfte aller vorhandenen Tanken getankt oder befinden wir uns in der Nähe des Gegners und er hat mehr getankt, als wir **-> fleeEnemy()** • Ansonsten prüfen, ob die nächste Tankstelle noch vorhanden ist (**checkTankPath()**) und, sobald die Wegpunkte leer sind, den Weg zur besten Tankstelle finden (**createPathToNextTanke()**)
	`calculateBES()`	errechnet die Beschleunigungsrichtung für unseren SpaceBall	-	Beschleunigungsvektor		wird jede Iteration ausgeführt	• Sind keine Wegpunkte vorhanden, entgegen des Geschwindigkeitsvektors auf 0 komplett herunter bremsen • Werden wir am nächsten Wegpunkt zu schnell sein (**checkIfTooFast()**) -> Vollbremsung entgegen des Geschwindigkeitsvektors • ansonsten fahren wir in die Richtung der Summe des normierten Vektors zum ersten Wegpunkt der **waypointList** (**dir**) und des fünffachen normierten Vektors, der die Kombination des aktuellen Geschwindigkeitsvektors und des Zielvektors ist (**corr**) • Wenn unsere Distanz (**calcBreakDistance()**) zum Herunterbremsen auf eine am Wegpunkt benötigte Geschwindigkeit (**calcBreakingEndVel()**) größer ist, als die Distanz zu diesem, bremsen wir, aber wir korrigieren unseren Weg weiterhin • Sobald wir in die Nähe des ersten Wegpunktes unsere **waypointList** kommen (Abstand zu diesem ist kleiner, als der in **constWaypointReachedRadius** definierte), wird dieser gelöscht und die Wegpunkte neu eingezeichnet
	`checkIfTooFast()`	überprüft, ob wir an einem Wegpunkt zu schnell sein werden	-	TRUE / FALSE	Wird in **calculateBES()** verwendet		
	`emergencyBreaking()`	Überprüft, ob er eine Notbremsung durchführen muss		TRUE / FALSE			
	`calcBreakingEndVel()`	berechnet die minimale Geschwindigkeit, die wir am nächsten Wegpunkt haben sollten	-	Betrag einer Geschwindigkeit			
	`calcBreakDistance(vel, endvel)`	Errechnet die benötigte Entfernung, um von einer Geschwindigkeit auf eine andere abzubremsen	• *vel:* höhere Geschwindigkeit (SKALAR) • *endvel:* niedrigere Geschwindigkeit, auf die heruntergebremst werden soll (SKALAR)	Betrag einer Distanz			
	`setupNodeGrid()`	Die Welt wird in Nodes unterteilt, denen ein bestimmter Wert zugeordnet wird, je nachdem, wie gut es für uns wäre, dort entlangzufahren	-	-		Am Anfang und bei jedem Verschwinden einer Mine	
	`updateNodeGrid(pos, radius)`	Updated das Nodegrid um die Position mit dem Radius					
	`isWalkable(pos, radius)`	Überprüft, ob eine Position betreten werden darf, oder nicht	• *pos:* Position auf dem Spielfeld, die geprüft werden soll (VEKTOR) • *radius:* autom. gesetzt durch **secureSpaceballRadius** (SKALAR)	TRUE / FALSE			
	`customSetdiff(array1, array2)`	Gibt ein Differenzelement von zwei Arrays aus, die sich nur um ein Element unterscheiden. Dabei überprüft: array1(i).pos == array2(j).pos Sinnvoll für Minen oder Tankstellen	array1, array2 die beiden arrays	Differenzelement der Arrays			
	`findPath(startp, endp)`	Sucht den bestmöglichen Weg zwischen zwei Punkten	• *startp:* Anfangspunkt (VEKTOR) • *endp:* Endpunkt (VEKTOR)	CELLARRAY mit Wegpunkten als VEKTOR			
	`getValidNodePos(gridPos)`	Sucht einen begehbaren Wegpunkt in der Nähe der eingegebenen Position	• gridPos: eine Position auf dem Spielfeld als Grid-Koordinate (VEKTOR)	eine begehbare Position auf dem Spielfeld als Grid-Koordinate (VEKTOR)			Sollte ein Wegpunkt nicht begehbar sein (**isWalkable()**), sucht diese Funktion einen Wegpunkt in der Nähe, der begehbar ist
	`worldPosToGridPos(pos)`	Konvertiert Welt-Koordinaten zu Grid-Koordinaten	• *pos:* eine Position auf dem Spielfeld als Welt-Koordinate (VEKTOR)	*pos:* eine Position auf dem Spielfeld als Grid-Koordinate (VEKTOR)			Die Welt-Koordinate wird gerundet und das Ergebnis ist die Grid-Koordinate der dichtesten Node
	`clamp(value, min, max)`	„Schneidet“ eingegebene Werte an min/max Werten ab	• *value:* ein Wert (SKALAR II VEKTOR) • *min:* SKALAR • *max:* SKALAR	SKALAR II VEKTOR			Ist der eingegebene Wert kleiner, als der Min-Wert, so kommt als Ergebnis der Min-Wert heraus, genauso beim Max-Wert. Werte dazwischen werden so ausgegeben.
	`nodeFromGridCoords(pos)`	Gibt den Node anhand Grid-Koordinaten aus	• *pos:* Grid-Koordinaten (VEKTOR)	STRUCT (Node)			
	`equalsNode(a, b)`	Überprüft, ob zwei Nodes dieselben sind	• *a:* Node 1 (STRUCT) • *b:* Node 2 (STRUCT)	TRUE / FALSE			
	`equalsVec(a, b)`	Überprüft, ob zwei Vektoren dieselben sind	• *a:* VEKTOR • *b:* VEKTOR	TRUE / FALSE			
	`getNeighbourNodes(node)`	findet die Nachbarnodes des Parameter nodes	• node: Pathnode	Cell Array von Nodes	in findPath werden Nachbarnodes eines Nodes gebraucht, um den Pfad zu berechnen		
	`simplifyPath(path)`	vereinfacht einen Gegebenen Pfad	• *path:* CELLARRAY mit Wegpunkten als VEKTOR	waypoints : CELLARRAY mit Wegpunkten als Vektor			
	`containsHeapNode(nodes, pos)`	prüft, ob der Heap einen Node enthält	• nodes: Heap (Array von Nodes) • pos: Gridposition des Nodes	TRUE / FALSE			
	`insertHeapNode(heap, nodePos)`	fügt ein Node in einen Heap ans Ende an	• heap: Heap • pos: Gridposition des Nodes	neuer Heap mit eingefügtem Node			
	`removeHeapNode(heap, index)`	entfernt ein Node vom Heap	• heap: Heap • index: Index des Elements im Heap	neuer Heap mit entferntem Node			
	`sortHeapNodeDown(heap, index)`	sortiert ein Node automatisch entsprechend seinem Wert im Heap nach unten	• heap: Heap • index: Index des Elements im Heap	neuer Heap mit sortierten Nodes			
	`sortHeapNodeUp(heap, index)`	sortiert ein Node automatisch entsprechend seinem Wert im Heap nach oben	• heap: Heap • index: Index des Elements im Heap	neuer Heap mit sortierten Nodes			
	`swapHeapNodes(heap, index1, index2)`	Vertauscht zwei Nodes im Heap (gebraucht für die Sortierfunktionen)	• heap: Heap • index1: Index des 1. Elements im Heap • index2: Index des 2. Elements im Heap	neuer Heap mit getauschten Nodes			
	`vecNorm(vec)`	Normalisiert einen gegebenen Vektor (erstellt Einheitsvektor in dieselbe Richtung)	• *vec:* VEKTOR	VEKTOR			
	`appendToArray(array1, array2)`	Fügt Cellarray 2 hinten an Cellarray 1 an und gibt das zusammengefügte Array aus	• *array1:* CELLARRAY • *array2:* CELLARRAY	CELLARRAY			
	`createPathToNextTanke()`	Findet die nächste Tankstelle und fügt diese automatisch als nächsten Wegpunkt ein. Dabei holt sich die Funktion von createTankEvaluation eine Tankstellenbewertung und holt sich die beste Tankstelle heraus.					
	`createTankEvaluation(position)`	Erstellt eine Bewertung jeder Tankstelle - Wichtig um herauszufinden, welche Tankstelle als nächstes angeflogen werden soll. Es werden Entfernung zur eigenen Position, Winkel zu anderen Tankstellen und Entfernung zum Gegner berücksichtigt.	• *position:* Je geringer die Entfernung zu dieser Position, desto mehr Punkte für die Tanke				
	`getHeadedTanken()`	Gibt die momentan angesteuerten Tankstellen aus. Es sucht die Wegpunkte ab, und schaut ob sich in der Nähe der Wegpunkte Tankstellen befinden. Danach schreibt es die Tankstellen ID in einen Vektor, der am Ende ausgegeben wird.		tankenList: VEKTOR der Tankstellen in Tankstellen IDs			
	`checkTankPath()`	Überprüft, ob der Gegner eine Tanke erreichen wird, die wir auch ansteuern. Dabei wird die Zeit berechnet, die wir bzw. der Gegner braucht, die angesteuerte Tankstelle zu erreichen. Außerdem wird die Zeit beachtet, die der Spaceball braucht um seine Geschwindigkeit an die Tankstellen anzugleichen. Befinden sich zwei Spaceballs im Wettstreit um eine Tanke, wird der competitionmode aktiviert und wegpunkte hinter die Tankstelle gesetzt, um nicht vor der Tankstelle abzubremsen.					
	`attackEnemy()`	Regelt den Angriff: Falls der Gegner direkt angeflogen werden kann, wird der nächste Wegpunkt auf Abfangkurs gesetzt. Falls nicht, wird der Pfad zum Gegner über den Pathfinder berechnet.	-	-			
	`calcEnemyHitPosition()`	Berechnet die Position, auf die zugesteuert werden muss, um den Gegner abzufangen. Um bei großen Entfernungen nicht zu übersteuern, wird die Position direkt auf den Gegner gesetzt.					
	`getEnemyAccPos(enemypos)`	Diese Funktion verlängert den Punkt des Gegners nach hinten, damit mit Vollgas kollidiert wird und vorher nicht abgebremst wird.	• *enemypos:*				
	`fleeEnemy()`	Regelt die Verteidigung					
	`corridorColliding(startp, endp, radius)`	prüft, ob im gegebenen Korridor beschrieben mit Start, Endposition und Radius bzw. Breite, ob sich darin eine Mine befindet.	• *startp:* • *endp:* • *radius:*	True = Mine befindet sich darin False = Korridor ist frei			
	`lineColliding(startp, endp, radius)`	Prüft, ob die Strecke (Start bis Endpunkt) von einer Mine kleiner als radius entfernt ist.	• *startp:* • *endp:* • *radius*	True = Mine befindet sich darin False = Strecke			
	`distanceLinePoint(startp, endp, point)`	Berechnet die Entfernung eines Punktes von einer Strecke	• *startp:* • *endp:* • *point:*				
	`getPerpend(vec)`	Berechnet den Normalvektor des Eingangsvektors vec	• *vec:* VEKTOR				
	`projectVectorNorm(vec1, vec2)`	Projeziert vec1 auf vec2 und gibt die projezierte Länge aus (kann auch Negativ werden)	• *vec1:* VEKTOR • *vec2:* VEKTOR				
	`getTimeToAlignVelocity(vel1, vec)`	Berechnet die Zeit, die gebraucht wird um die Geschwindigkeit vel1 auf den Vektor ve auszurichten	• *vell:* • *vec:*				
	`getMaxVelocityToAlignInTime(vec1, vec2, time)`	Errechnet die maximale Geschwindigkeit die der Spaceball haben darf, um seinen Geschwindigkeitsvektor von vec1 zu vec2 in der gegebenen Zeit auszurichten.	SKALAR: Geschwindigkeit				
	`safeDeleteWaypoints()`	Löscht alle Wegpunkte und erstellt einen sicheren Bremswegpunkt, um nicht zu kollidieren					
	`debugDRAW()`	Zeichnet die Wegpunkte aus der „waypointList“ als kleine rote Punkte ein					