AI for Tactical Grenade Handling

William van der Sterren
- CGF-AI
  sterren@iae.nl
  December, 2000

keywords: AI, game, NPC, tactics, grenade, terrain reasoning, CGF, bot, CQB

1 Introduction

This paper discusses how to construct AI able to use grenades tactically in 3D tactical shooter games.

Today's AI implementations (in games such as Half-Life [Half-LifeSDK], Hidden & Dangerous, and Rainbow Six) do use grenades[a], but these AI implementations typically do not:

- intentionally bounce grenades around the corner
- intentionally lob grenades through windows or doors to a spot near their target
- use grenades to flush out threats from a hard to reach position
- understand when it makes sense and when it doesn't make sense to use grenades

Fig. 1 [a] Hand grenade bounced into a room, close enough to take out the threat (green lines are used to display the grenade trajectory); [b] AI throwing a hand grenade; [c] AI analysis results for the value of attacking the room to the left using a hand grenade.

These AI capabilities are essential to provide the player with smart AI team members of 'special forces' quality, to provide additional capable (opposing) autonomous forces, and to create a human-like multi-player combat experience as in, for example, Counter-Strike sessions.

This paper:

- discusses problems and solutions for generating and storing grenade trajectories
- illustrates how AI can better understand the use of grenades
- provides data on CPU load and memory consumption

The paper is organized as follows. Section 2 illustrates the intended grenade handling capabilities. Section 3 states the aim and requirements for the AI. Section 4 lists the pre-conditions for the AI and assumptions made about the game context.

Section 5 discusses generation of grenade trajectories: how to encode, pre-compute and select a useful and realistic grenade trajectory.

The AI 'grenade reasoning' in section 6. After sketching the complexity of using grenades tactically, specific attention is given to estimating the value of using a grenade, and selecting good grenade attack positions.

Section 7 lists experiments and the corresponding results. Further work is discussed in section 8, and conclusions are presented in section 9.

2 The New Grenade Handling Capabilities Illustrated

Imagine the following scenario:

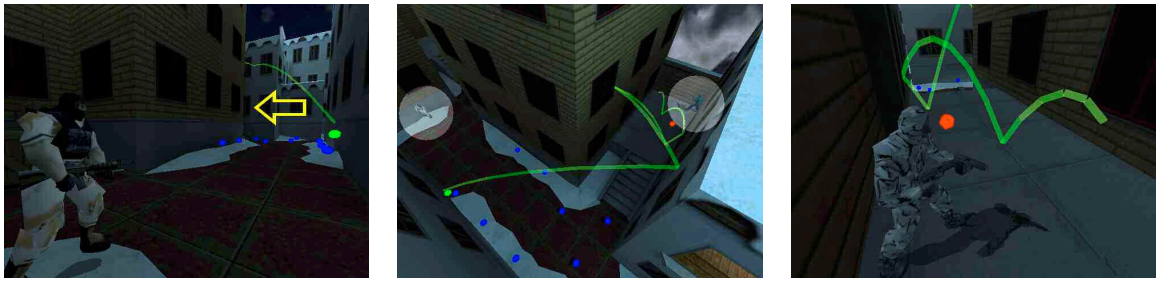
The NPC actor saw the enemy sniper move back, around the corner, into cover. 'That position may provide cover, but looks like a dead-end alley as well', the NPC thought. 'There's no way to evade a grenade from there but to move into my line of fire'.

Being the only one carrying hand grenades, it was his responsibility to check whether he could take out that sniper with a grenade.

He made a quick check. 'Darn, still too far from that corner to get a grenade near enough. However, moving up 5 meters allows me to do the job.'

The NPC took a grenade, moved up, figured out how to bounce a grenade around the corner, lobbed the grenade, and switched to his rifle...

Fig. 2 **[a] Initial situation: AI actor knows threat location, but isn't able to attack from this position. [b] The AI computations suggest: move up (to the green ball) to lob a grenade without getting into the threat's line of fire. [c] The AI suggested trajectory, from the target's point of view.**



The scenario illustrates the needs to:

- judge whether or not to (attempt to) use a grenade;
- check whether throwing a grenade is feasible;
- locating a good attack position to launch grenades from; and
- determine a good trajectory for the grenade to go around the corner (or through a window, etc.)

Fig. 3 [a] An attacker tossing a hand grenade down stairs (via a wall) towards a threat; [b] The hand grenade; [c] A hand grenade trajectory into a restroom.

3 Aim and Requirements

The aim, of course, is to build an AI realizing the scenario and needs sketched above. However, there is more to developing AI than just the functionality.

The resulting AI should fit within the game. There are requirements (and budgets) for the CPU utilization (average and peak) and memory consumption of this grenade handling AI.

CPU and memory budgets do not come out of thin air. Instead, they are derived from the (worst-case) targeted game situations and AI reasoning style. Of course, these budgets can be disputed, but they provide more guidance than no budgets.

3.1 CPU budget (on 'standard hardware'[b])

CGF aims to support combat between multiple fire teams (or "squads"), involving up to 32 AI actors. During the first seconds of the initial engagement, almost all of these 32 AI actors will try to engage each other (after which both the number of active AI actors and the grenade ammo available rapidly decrease).

The AI actor thinks at 10Hz. Every time, he considers his situation, and figures out the best weapon for the job. That means that he tests whether he potentially can use grenades to attack his target: does a trajectory from his position to the target exist?

Grenades are mainly used against known targets that move behind cover. Assume that every AI actor sees a target move behind cover twice per second. Every time, the AI will check whether using grenades is appropriate. If using a grenade is appropriate (the worst case), the AI actor will want to pick the best attack position to lob a grenade from.

Finally, handling a grenade (from pulling the pin, throwing, and readying a next grenade) takes about 3 seconds. During the first 0.5 second, the AI will adjust its aim for predicted movement of the threat.

Thus, the additional CPU load per second for AI grenade handling is:

Fig. 4 CPU budget for AI grenade handling.

32 x 10	=	320	checks for the existence of a grenade trajectory
32 x 2	=	64	checks for grenades being appropriate
32 x 2	=	64	checks for the best grenade attack position
32 x 0.5 / 3	~	6	aiming attempts

A CPU budget for this additional feature might be 0.5 fps, for a game rendering at 25 fps, thus 0.02 s = 20 ms per second.

Note that one of the computations, aiming a grenade, is a given. To aim the grenade, a trajectory is emulated and tested for obstacles. This takes about 1/8000 s each, thus consuming 0.8 ms for 6 attempts. That leaves 19.2 ms.

3.2 Memory budget

The memory-consumption for grenade specific look-up tables should not exceed 0.5 Mbytes per (significantly different) grenade type.

Often, hand grenades, flash bangs, and smoke grenades have similar physical characteristics and thus can share a trajectory. This typically is not the case for hand grenades and rifle launched grenades.

Fig. 5 [a] CGF-AI's waypoints (also on the drainpipe leading up to the roof); [b] CGF's AI (without grenades) in action; [c] a M203 grenade launcher being reloaded.

4 Pre-Conditions and Assumptions

AI isn't designed in a vacuum. Instead, we have a lot of assumptions about the game physics, the terrain representation available, and the AI terrain reasoning capabilities:

- the game physics for grenade bouncing are predictable to a large extent, optionally with some small randomized effects
- the grenade's launch velocity and direction are predictable to a large extent, optionally with some small randomized effects
- the AI is capable of predicting threat positions when they've lost sight of them (without this prediction capability, there wouldn't be much need to bounce grenades around the corner)
- the terrain is thought of by the AI as a grid of (small) cells with more or less uniform surface dimensions (I happen to use waypoints to define cells, but mesh based approaches should work equally well [Snook])
- the AI understands about areas such as rooms, halls, tunnels, and can efficiently determine entrances and exits from these areas

These AI pre-conditions may seem demanding. However, there actually is neither use nor hope for the AI to tactically employ grenades if these physics and terrain representation are not available.

In the case of the CGF AI and Action Quake2[1], these pre-conditions were satisfied.

5 Approach: Obtaining Trajectories

Being able to bounce, to follow curves, and to fly for up to 3 seconds, a grenade may travel unexpected ways. This makes it hard (and likely CPU intensive) to judge whether and how a grenade can reach a location.

Whereas a human can look at its environment and imagine instantly how to toss a grenade around a corner, through a hole or window, this is not completely the case for an AI actor.

Experiments confirmed this: generating a (feasible) trajectory takes in the order of 0.05 to 0.1 seconds (see Experiments). Such a CPU load exceeds the budget available to the AI. It effectively prohibits even a single AI actor to frequently think about using grenades for anything but a straight curve.

In other words, the AI cannot use the full spectrum of grenade trajectories without some help.

[1] Action Quake2 is a user-modification of the Id Software game Quake II. QUAKE and Quake II are registered trademarks of Id Software Inc.

The approach presented here trades computing time for memory, and uses an off-line computed look-up table to support AI grenade reasoning.

The main questions in designing the look-up table are:

- what to store?
- which queries to support?

Below, the contents of the look-up table are discussed. The tables themselves have been organized around grenade target positions, mirroring the interests of the attacking AI: from where to attack a given target position?

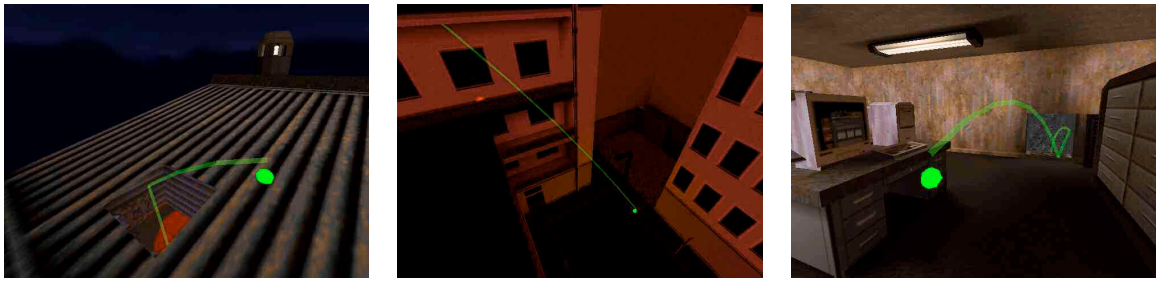
5.1 Encoding the Trajectories

Computing every possible trajectory in advance, for every possible pair of coordinates (QuakeII has ~1 inch resolution) would lead to an enormous (and redundant) amount of trajectories. If trajectories are to be stored in a limited amount of memory, solely a subset of the trajectories should be selected. But which subset?

Other (non-grenade) queries in the CGF AI almost all have been implemented in terms of cells / waypoints (see [Snook] or [Reece] for a discussion on terrain representations for AI). Those waypoints represent solely the terrain accessible to the human and AI actors. In the CGF AI, a dense grid of waypoints is used as the backbone of the terrain representation for navigation, (travel) distance, areas/sectors, tactical fitness, and other purposes.

The new grenade-related queries again would refer to (attacker and target) locations in terms of waypoints. So, it seemed obvious to deal solely with trajectories between waypoints. But are these trajectories sufficiently representative for the trajectories from or to the close surroundings of these waypoints?

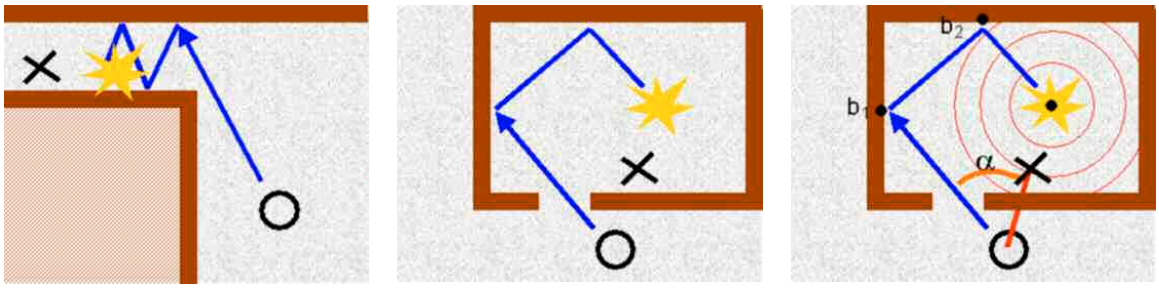
Fig. 6 **[a] A hand grenade trajectory through a hole in the roof; [b] A rifle grenade trajectory hitting the ceiling above the threat's balcony position; [c] A hand grenade trajectory into an air vent tunnel.**



The waypoint based trajectories are representative if these trajectories are to a large degree robust against deviations in the source (attacker) and destination (target) positions. If the attacker is not positioned exactly in the center of the cell, or on the waypoint, the pre-computed trajectory information should still lead to a grenade delivered close enough to the target position in most cases. (Some error, of course, is allowed for realism).

From studying a few grenade trajectories, and their relation to cells/waypoints, it is not that hard to decide what information to record.

Fig. 7 **[a] A trajectory bouncing into an alley, near the threat ('x'). [b] A trajectory bouncing into a room, engaging a nearby threat. [c] The key aspects of a trajectory annotated: aiming angles, bounce spot, damage radius.**

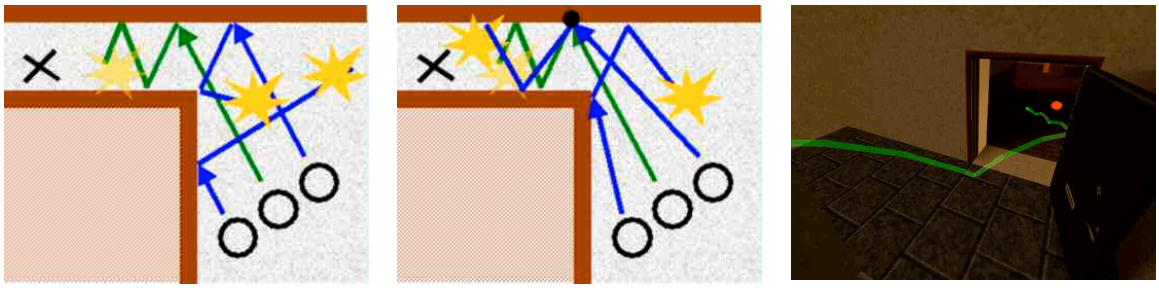


The images above present top-views of two different (bouncing) trajectories arriving near a threat (the 'x'), originating from the source (the 'o'). The image to the left shows a trajectory entering an alley, whereas the other images show a trajectory entering a room.

Due to the area damage effect, the grenade need not be delivered exactly on the threat's position. That means, in most cases, the pre-computed trajectory is robust against threats being at a small distance from the cell's center or waypoint.

A bigger problem is a deviation between the cell's origin and the attacking actor's position. This is illustrated below:

Fig. 8 **The effects of a (lateral) deviation when using the same launch direction as a pre-computed trajectory. [b] The effects of a (lateral) deviation when using the same bounce/aim spot. [c] Tossing the grenade through a door, using a bounce spot on the floor. Deviations in the attacker's position are unlikely to prevent getting the grenade through the door, if the bounce spot is aimed for.**



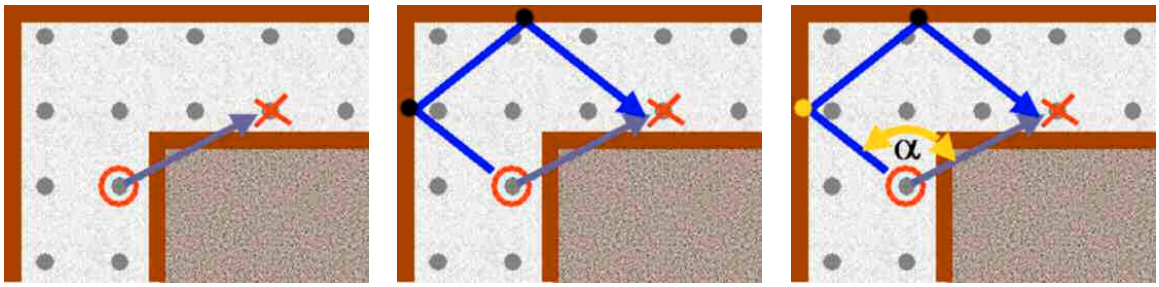
The image to the left shows the effects of a (lateral) deviation in position on a pre-computed launch direction (angles). The image in the center shows the effects of a (lateral) deviation in position when using a pre-computed (first) bounce spot.

In general, using the pre-computed bounce spot was found to be more robust than using pre-computed direction.

Note that robustness can be improved by selecting the more robust trajectories when exploring possible solutions. Note also that some error in using trajectories can be tolerated since it makes the AI less perfect.

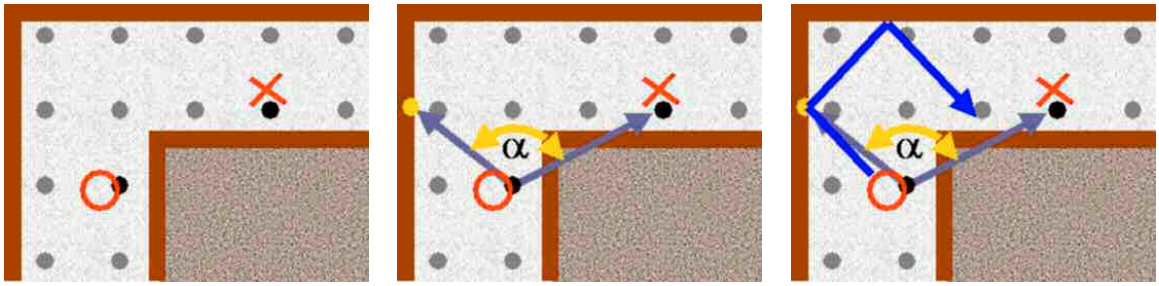
Basically, CGF records trajectories by storing, for a <attacker waypoint, target waypoint> pair, the direction to the spot to aim for (typically, the first bounce spot for the grenade). The whole process of encoding and decoding a trajectory is illustrated in the figures below:

Fig. 9 **Trajectory encoding: [a] First, feasible trajectories are determined for pairs of waypoints from waypoint (gray dots). [b] A feasible trajectory is found, including several bounces. [c] Aiming for this trajectory is done via the left-side wall (yellow dot). This aiming is encoded as the difference in angles between the straight line to the target, and the aim direction.**



The direction to the aim spot from a attacker waypoint to a target is expressed using yaw and pitch angles. More precisely, the direction is expressed in yaw and pitch angles relative to the line from attacker to target, making use of the property that most trajectories don't require throwing the grenade away from the target, thereby saving a few precious bits.

Fig. 10 **Trajectory decoding: [a] First, the nearest waypoints (black dots) for attacker and target locations are determined. [b] Then, the (bounce) spot to aim for is reconstructed, using the angle difference between the straight-line direction to the target waypoint and the trajectory. [c] The resulting trajectory goes from the attacker via the reconstructed aim/bounce spot to a location near the target. Note that (due to the attacker and target not being exactly on the waypoints), the resulting trajectory deviates from the pre-computed one (but still is useful).**



As the experiment results illustrate, storing a single trajectory (attacker waypoint, target waypoint, aiming direction, launch velocity, and a few other flags) consumes about 5.5 to 6.0 bytes (including overhead), or 5 bytes without overhead. A size of 4 bytes is achievable when making a few additional assumptions about the number of waypoints.

5.2 Approach: Generating Trajectories

Though it is clear how to record trajectories, they also need to be generated. This section first discusses the generation of rifle launched grenade trajectories before going into the more complex (bouncing) hand grenade trajectories.

5.2.1 Computing Rifle Launched Grenade Trajectories

As an aside: first a less complex trajectory generation problem: rifle launched HE grenade trajectories. Because a HE rifle grenade explodes upon impact, it cannot be bounced. A rifle grenade is launched with a fixed velocity, and is solely be aimed by adjusting the aiming angles. For a source and target combination, far fewer trajectories are possible than with hand grenades.

Fig. 11 [a] The target: a sniper positioned in the upper room, overlooking distant parts of the city; [b] Overview of the attacker and the room being attacked; the attacker cannot hit the target (the sniper) but can hit the ceiling overhead; [c] The rifle grenade launcher equipped attacker, seeing the grenade impact on the ceiling above the target.

For a rifle grenade, finding a trajectory just involved a two-pass approach:

1. for each attacker - target waypoint pair:
   - chose useful positions of impact: at the target's feet, on a close enough ceiling above the threat, or on a close enough wall behind the threat
   - test whether a trajectory to these positions is (mathematically) feasible; this simply involved some approximation and solving a number of 2nd order equations
   - test whether the trajectory was physically possible in the world geometry;
2. for each attacker - target waypoint pair without a direct trajectory:
   - look up all waypoints near target waypoint
   - look for trajectories from attacker to waypoint close to target
   - pick the closest one (with line of fire), and test if within damage radius

This process is more or less analytical: if there is a solution, there is a straightforward way of finding it.

In about 169 seconds, on the test hardware, 673 x 673 potential rifle grenade trajectories can be checked (that is: about 2700 trajectories per second on average, with a worst-case that is near 360 trajectories / second (if all trajectories are valid and require all checks). If trajectories can be generated this fast, pre-computing them may not be required.

5.2.2 Computing Hand Grenade Trajectories

Whereas hand grenades (that bounce, and explode on time) have a lot in common with these rifle grenades, finding valid trajectories for hand grenades is very different from finding trajectories for rifle grenades.

The two main differences:

- trajectories that bounce cannot be 'solved' analytically for a given start location and end location;
- evaluation of bouncing trajectories (by the game engine) is more expensive

To generate useful hand grenade trajectories, a 'genetic algorithm' was developed (in a number of iterations):

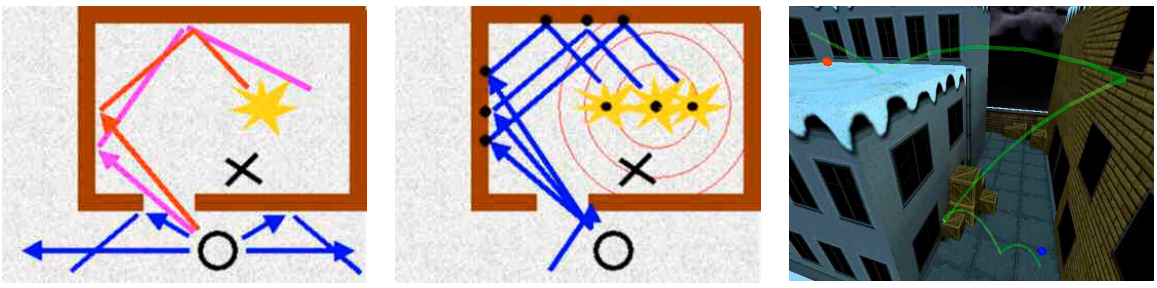
- create a number of 'initial attempt' trajectories by throwing in a 180 degrees yaw range / 180 degrees pitch range with respect to the target;
- repeat:
  - evaluate each of the attempts, rank them, and keep the better ones
  - expand on the best attempts by creating a number of new attempts within a small yaw and pitch range
- pick the best attempt, and determine whether it is 'good' enough

The real work of course is in defining the evaluation function, and in getting the algorithm to be sufficiently fast while generating the important trajectories. The evaluation function of a trajectory is explained below.

Getting the algorithm faster primarily proved to be an issue in caching the most expensive computations: ray tracing a trajectory. Once the algorithm was re-designed to find all targets for a single source waypoint, efficient caching of trajectories from that source, for all angles and velocities, became possible. This caching reduced the computation time for a complex 1000+ waypoint level from a couple of days to the odd 8 hours.

The majority of these long hours is spent on ray tracing the BSP.

Fig. 12 [a] Multiple attempts (blue, magenta) at a good trajectory, of which the magenta is chosen and improved to the red trajectory; [b] Testing a trajectory's robustness by taking into account slight deviations of the launch angles; [c] A basketball bounce pass style trajectory.



5.3 Judging the Quality of a Trajectory

Essential in getting good results from a genetic algorithm is the evaluation function. The evaluation function should distinguish the useful attempts and useless attempts, and select the most promising ones for further 'breeding'.

The evaluation function should favor those trajectories that:

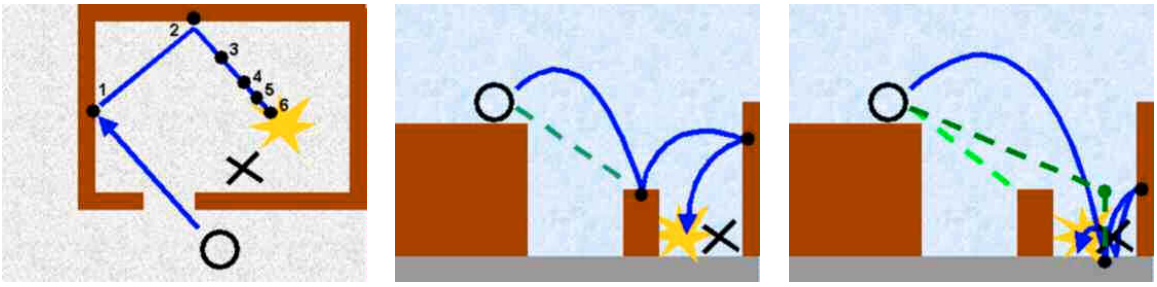
- damage the target
- do not damage the attacker
- are robust against deviations in aiming position and aiming angles
- 'feel human-like and realistic'

(The 'feel human-like' requirement was initially ignored. Upon seeing plenty of 'pinball' or 'billiards' style trajectories being generated, the evaluation function was extended).

The resulting evaluation function judged the following properties of a trajectory:

- *damage to the target*
  The grenade arriving within the damage radius of the target, and having a clear line of fire;
- *distance to the target*
  The distance to the target (ignoring damage radius and line of fire); this property is required for the genetic algorithm to work towards valid solutions;
- *damage to the attacker*
  The grenade arriving within the damage radius of the attacker and having a clear line of fire;
- *distance to first bounce*
  The distance from the attacker to the grenade's first bounce. Since it is this bounce point that is stored and use for aiming, the more distant the bounce point, the smaller the effect of deviations in attacker position and aiming errors.
  Distant bounces also feel more realistic (the algorithm generated too many basketball bounce pass style trajectories otherwise).
- *(almost) visible first bounce location*
  The location of the first bounce location should be visible by the attacker, or the bounce location should be slightly below a location visible by the attacker. This property was added for realism: humans are bad at aiming grenades for spots they cannot see.
  A special case is made for grenades lobbed through higher positioned apertures (typically, windows) or lower positioned apertures (holes). In that case, the attacker does not worry about the bounce spot, but about the grenade making it through the window or into the hole. The location above the bounce spot then typically is visible.

Fig. 13 **[a] The bounce complexity illustrated: for a trajectory the bounces 1, 2, and 3 hit varying planes, but bounces 4, 5, 6 and are consecutive bounces on the same plane that bounce 3 hit. The more planes involved, the less realistic the trajectory. [b] Grenade trajectories with a first bounce spot visible to the attacker are more realistic in most cases. However, this is not one of them, since it would be better to aim for the pit. See: [c] Same situation, but with a better trajectory making its first bounce in the pit. Though this first bounce location cannot be seen by the attacker, it can be assumed since the attacker can see a position slightly overhead the bounce spot.**



- *number of bounces (on different planes)*
  The number of bounces in the trajectory, counting consecutive bounces on the same plane as one. Planes are compared for being similar using the dot product of the normal vectors (> 0.9x).
  Again, a property to favor more realistic trajectories: humans have troubles with many bounces across different planes.

- *launch angles*
  Preferably, the launch angles are within a not too wide cone towards the target, for two reasons:
  - launch angles outside that cone may not fit in the format used for storing trajectories.
  - throwing a grenade in a direction 'away from the target' to get the grenade to the target does not feel realistic.

- *launch velocity*
  Higher launch velocities are favored because in the Action Quake2 game, the launch velocity deviations are constant, and these deviation thus are relatively for higher velocities.
  This property favors robustness.

All these properties are evaluated for the trajectory and for derived trajectories with slight deviations and in the launch angles and launch position. The resulting score for a trajectory is the 'pessimistic average' of the all these trajectories, that is, the average of the weaker individual trajectories. The score then reflects the robustness of the trajectory, and also enforces some risk avoidance.

6 Using grenades: why, when and where

Grenades are not employed 'any time, any place'. Typically, grenades are scarce: both player and AI actors are limited to only a few. So the AI need better understand why, where, and when it makes sense to lob or launch a grenade.

This section discusses AI grenade reasoning. First, an overview is given of the kind of reasoning required. Then, two algorithms are presented in more detail: one algorithm for the AI to estimate the value of throwing an grenade at a target, and a second one to pick the better positions to attack from.

6.1 Grenade Related AI Reasoning

To the AI, grenades are complicated stuff, partially because there is a lot involved. For example:

- grenades can damage threats who are out-of-sight
- assessing if a threat can be reached by a grenade is hard in the presence of obstacles
- grenade explosions may also damage non-hostiles near the target
- badly thrown grenades may bounce back to the attacker or other non-hostiles
- once the pin has been pulled, the grenade should be disposed off safely, even if the threat moves out of reach
- grenade trajectories may be blocked by nearby friendly troops, or dynamic structures (doors, vehicles) in the game world
- grenades can be cooked off, to prevent enemies from returning the grenade
- grenades are more effective against threats in confined areas
- game actors typically receive less damage from a grenade when behind cover and when prone
- ...

Many of these aspects can be dealt with by the AI using a:

- trajectory look up table
- test for obstacles on the trajectory
- proximity and cover tests for non-hostiles
- grenade evading behavior
- detailed AI weapon model

A trajectory look up table provides quick answers to whether a threat can be reached. The look up table may also rule out the trajectories that come with a large risk of the grenade bouncing back

To some extent, environment dynamics (such as doors) may be accounted for in the look up table, but typically, the look up table does not contain alternative solutions when the default trajectory is blocked.

Testing the trajectory for obstacles is a must for the attacker AI. The attacker's team mates or other non-hostiles might block the trajectory. Doors, vehicles, or other dynamic parts of the game world also might get in the way.

If the trajectory is feasible but is also blocked, the AI may search for alternative trajectories (expensive), alternate attack positions, or try to resolve the obstacle (open the door).

A trajectory that is feasible and clear of obstacles may still endanger friendly troops and non-hostiles close to the target location. Tests for these actors being near the target location, and without direct cover from the grenades blast need to be performed.

If friendly actors are in danger of getting hurt, the attacker may temporarily give up on using grenades, or inform the friendly troops of an incoming grenade ("frag out!") and assume they will evade the blast.

The grenade blast typically is evaded by moving away from the expected grenade explosion, preferably into to cover. To reduce the risks, ducking or going prone is advised [MCWP 3-35.3].

A detailed weapon model, describing the details of the hand grenade or rifle grenade (cooking off time, arming time, damage radius, etc.) can support the AI in taking the subtle details of the grenade into account.

As an alternative, the AI may cheat and ignore the hand grenade's pin and cooking off time at all, and use a random 'remaining time' to explosion.

Fig. 14 [a] Moving up to a good position to toss a grenade through the window (see also [c]); [b] A rifle attached grenade launcher; [c] The value of throwing a grenade into the room (see situation as in [a]).

More complex algorithms are necessary to enable the AI to perform advanced reasoning, such as:

- what's the value of throwing a grenade from here to that position?
- what is the best position to move to from here to attack that position?

These algorithms are discussed below.

6.2 The Tactical Value of Using a Grenade

To tackle the question of 'when to use a grenade', input was gathered in on-line games such as Rainbow Six: Rogue Spear and Counter-Strike (for Half-Life), and from military manuals. This quickly resulted in a number of simple heuristics.

Additional heuristics surfaced when debugging the 'unrealistic' feel of the initial AI grenade handling.

Here are the results: Employing a grenade to attack a position P_{target} from position P_{source} is more useful if:

1. a rifle cannot do the job (that is, there is no direct line-of-fire from P_{source} to P_{target}
2. the threat will have troubles evading the blast, because the target area is small and/or restricts movement (for example, tunnels, small rooftops)
3. exits from the target area can be covered with rifle fire from P_{source}
4. it is hard or takes a lot of time to reach P_{target} from P_{source}
5. the target area is known for being frequently visited, being used for ambushes or sniping
6. it takes a lot of time from P_{source} to establish a direct line-of-fire to P_{target}
7. the position P_{target} is a tactically stronger position than P_{source} (compare a dominating roof top position to a position in the middle of an empty square)
8. multiple threats are supposed to be near P_{target}

Fig. 15 [a] Value of a grenade thrown into a restroom; [b] Value of a grenade thrown from a roof into an alley (the roof being incorrectly classified as street); [c] Value of a grenade thrown downstairs in a large hall.

Given these heuristics, the overall value of using a grenade against a position simply is the weighted sum of the value of the individual heuristics. The AI actor then tests this value against a threshold, typically derived from the amount of grenades available, the rules-of-engagement, etc.

Below, the implementation of the heuristics 2, 3, 4, 5, 6, and 7 is discussed.

The threat's capability to evade the grenade's blast is largely determined by the target area. If that area is a large open street, evading is easy. If the area is a small room, evading is hard. If the area restricts movement (ladders that need to be climbed, doors that need to be operated, nearby cliff or rooftop edges), evading is harder.

The CGF AI regards the terrain as a number of areas (or regions) connected by portals[2]. Each area is classified (as a room, hall, street, alley, rooftop), and has a size. The heuristic 2 is implemented as a function of area type and size.

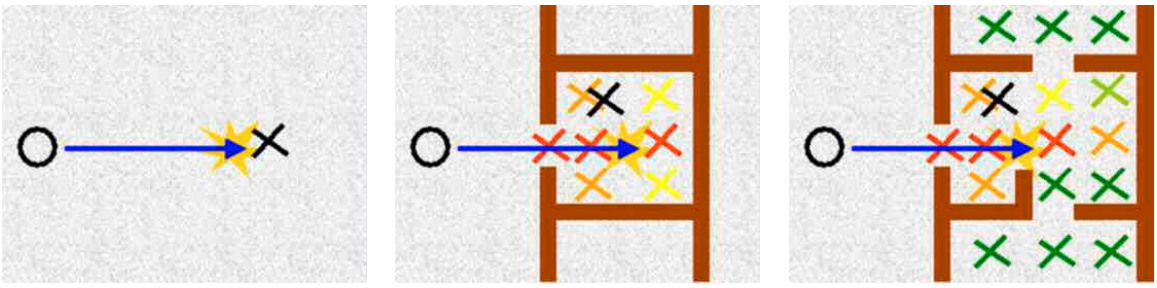
[2] The level designer typically creates such an 'areas and portals' annotation of the game world. However, the original and limited QuakeII AI never had a need to know about areas such as rooms, halls, streets, etc. and portal such as windows, doors, ladders. As a result, that info is missing from the QuakeII maps used by the CGF AI.
To make up for that, an algorithm was developed that using the waypoint grid and world geometry automatically constructs and classifies areas and portals. It does a decent job

Grenades are often used to flush out a threat from a position. This flushing out is solely effective if the attacker can fire into the exits from the threat's position.

Heuristic 3 is implemented by retrieving the all exits for the target area, and checking whether the attacker has a line of fire to these exits. The score is 1 if all exits are covered, 0.5 if one out of two exits is covered, and 0 otherwise.

A special case occurs when the attacker and threat are in the same area (such as a large hall or street). In that case, a more detailed check on paths leading to the exit(s) should be performed.

Fig. 16 [a] No need to use grenades if there is no cover for both attacker and threat; [b] Optimal use of grenades if the threat ('x') cannot evade the blast without entering the attacker's line of fire; [c] Maybe use grenades because the threat has limited opportunities to evade the blast.



If it will take a long time for the attacker to get at the target position, in general, it will be tough to force direct fire contact with the target. Such a 'long travel time' situation typically occurs when the threat is on a rooftop, whereas the attacker is at street level. In such a case, the threat has plenty of options to avoid contact and disappear. A grenade then offers the sole option to attack immediately, and becomes more useful.

The threat's situation is even stronger if the threat is able to quickly reach the attacker, whereas the attacker cannot reach the threat easily.

This heuristic 4 is implemented by determining the time needed for the attacker to get to the target position, and the time needed for the threat to get to the attacker position.

Heuristic 5 is similar, and its value is determined by the number and quality of nearby lines of fire to the target position.

Grenades are probably the safest way to distract or attack threats that are sniping or in ambush. Thus, in these cases the value of using a grenade is larger.

In games such as Rainbow Six: Rogue Spear, grenades are also employed to pre-empt a rush-assaults by the opposing force. More generally speaking, grenades have higher chance of taking out threats at locations that are frequently visited.

The CGF AI captures and interprets player and AI activity during the game, and relates that 'combat experience' to the terrain. That way, the AI learns where snipers might be located, where ambushes are more likely, where most traffic occurs, where most damage is done, etc. It is a reinforcement learning mechanism[c], enabling a better tactical understanding of the terrain that is adaptive as well.

Exactly this 'location specific combat experience', stored per waypoint and updated after each mission, is inspected to implement heuristic 5.

Another case when the threat should not be ignored, even if there is no direct line of fire, occurs when the threat occupies a very strong tactical position. For example, the threat occupies a tower from which all terrain can be observed and attacked, whereas the attacker is on the street.

Fig. 17 [a] Value of throwing a grenade into an alley; [b] Value of throwing a grenade on a high, hard to access, roof; [c] Value of tossing a grenade from a low roof into a room.

The CGF AI performs an off-line analysis to determine the tactical value of each location. This is done by taking into account the location's intrinsic properties (freedom of movement, amount of light available), the location's potential interactions with other locations (does it overlook many locations, does it provide nearby cover when engaged from other locations), and the location specific 'combat experience' (described above). This results in a tactical ranking for each location. Using these rankings, the locations can be compared for their tactical value.

A simple inspection of the tactical values for the attacker and target locations suffices to implement heuristic 7.

Note that this algorithm is a more-or-less straightforward translation of tactical concepts into simple evaluation functions, though might take some 'black art' and trial-and-error to come up with the right tactical concepts.

A disadvantage is that the original tactical concepts are not explicit in the resulting algorithm, unlike, for example, the rule-based approach taken by the SOAR-agent framework [Laird].

The main advantage, however, is the performance of the algorithm: it takes the AI less than 0.2 ms 2 to determining the value of using grenades against a target position. This enables multiple AI characters to frequently consider using grenades.

This kind of performance is mainly due to an underlying 'terrain reasoning database' optimized to support algorithms like these. The terrain reasoning database describes the terrain in terms of lines of sight/fire/concealment/cover, distances and travel times, areas and portals, key locations, location specific combat experience, etc.

6.3 Finding Good Grenade Attack Positions

When the AI actor spots a threat, and considers using grenades appropriate, it is important to find a good position to lob a grenade from. Such a position, if it exists, can be found by locating all positions from which a grenade can be lobbed to the target position, and ranking them for various characteristics.

Finally, the highest ranking position that also meets a few other criteria (for example, not occupied by a team member, and not in the line of fire of other threats).

Fig. 18 Example function definition for retrieving grenade attack positions.

```
unsigned int GetNearbyIndirectFireAttackPositionToTarget
               ( nodeid_t               aSourceSpot,
                 nodeid_t               aTargetSpot,
                 time_t                 aMaxTravelTime,
                 weaponid_t             aWeaponId,
                 annotatednodeid_t*     theResult /* out */
               );
       // returns the number of valid nodes (waypoints), and (annotated
       // and sorted by decreasing score) all valid nodes n in
       //     GetNodesWithinXDistanceOfNodeAnnotated
       //       (aSourceNode, aMaxTravelTime)
       // for which holds
       //     CanFireAtTargetUsingIndirectFireWeapon
       //       (n, aTargetSpot, aWeaponId)
       // and evaluates for these nodes n a score based on
       //   + lack of line-of-sight/fire to n from aTargetSpot
       //   + travel time to n from aSourceSpot
       //   + tactical quality of position n
       //   + cover in the general direction n - aTargetSpot
       //   + a path, concealed from aTargetSpot, to n
```

Again, we need to find a number of tactical properties defining good grenade attack positions, given a source position P_{source} and a target position P_{target}.

The following heuristics are used: an attack position P_{attack} is more appropriate if

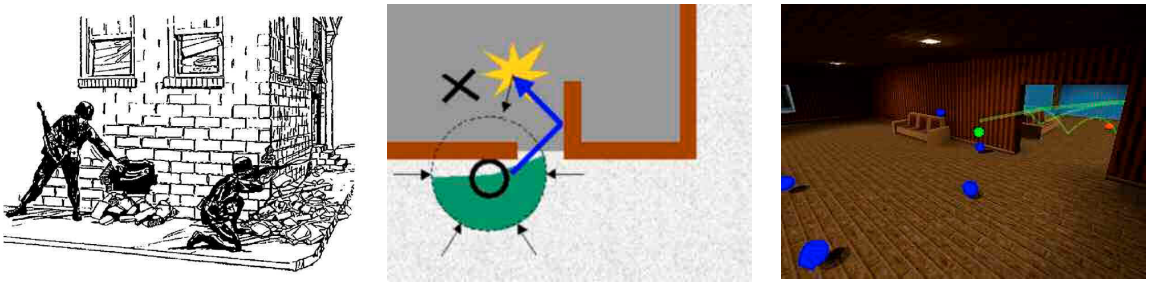
1. it takes little time to get there (from P_{source});
2. it offers concealment and cover from P_{target};
3. it is a better tactical position;
4. it offers concealment and cover from the general direction to P_{target} (to offer protection from the blast, even if the grenade doesn't land precisely on target)
5. the attack position can be reached from P_{source} without being seen from P_{target} and its direct surroundings.

The implementation of the heuristics 1, 2 and 3 have counterparts in the 'estimate value of using grenade' algorithm, and need no further discussion.

Heuristic 4 has been introduced to correct the AI for not hugging walls when throwing grenades (as is implicitly suggested in the USMC MOUT manual).

The heuristic is implemented as a function of amount of observation the position has from the general direction of the target. This tends to be zero or very small when obstacles are present shielding the attacker from the blast (as is illustrated below).

Fig. 19 [a] Hugging the wall (offering protection from the blast) when throwing a grenade (from USMC MCWP 3-35.3, fig 27a); [b] The wall hugging and related observation angles; [c] The algorithm's result: positions to toss a grenade into a room - the green position is preferred because of the wall hugging possible.



Heuristic 5 checks for the visibility of movement on the path from the source location to the grenade attack position, as observed from the target location and its immediate surroundings.

Basically, this heuristic is implemented by determining for each path (expressed in waypoints) to an attack position the number of waypoints in overseen by the target location and its surroundings.

The heuristic is implemented as a function of amount of observation the position has from the general direction of the target. This tends to be zero or very small when obstacles are present shielding the attacker from the blast (as is illustrated below). The implementation of this algorithm also needs than 0.3 ms on the test hardware. Such a performance enables the multiple AI actors determine attack positions frequently.

7 Experiments

The concepts, design and implementation mentioned above have been included in CGF. Below, you'll find some hard data on the implementation:

- Action Quake2 hand grenades
- rifle grenades (a CGF addition)

The hand grenades are thrown with one of three (player selectable) launch velocity of (about) 10 m/s, 18 m/s, or 23 m/s respectively, and about 0.5 m/s random deviation in the launching speed and direction. The hand grenades explode 2.0 seconds after being thrown (the throw animation itself takes almost 1 second). The hand grenades have a damage radius of 8.5m, provided a free line of sight is available to the target.

The hand grenades can be bounced.

The rifle grenades are modeled after the commonly known 40mm HE rifle grenade (as used in M203 grenade launchers): they explode on impact, but only after an 'arming delay' of 0.3 seconds to prevent the attacker to be hurt by the grenade impacting nearby. Again, the damage radius is about 8.5m, provide a free line of sight is available.

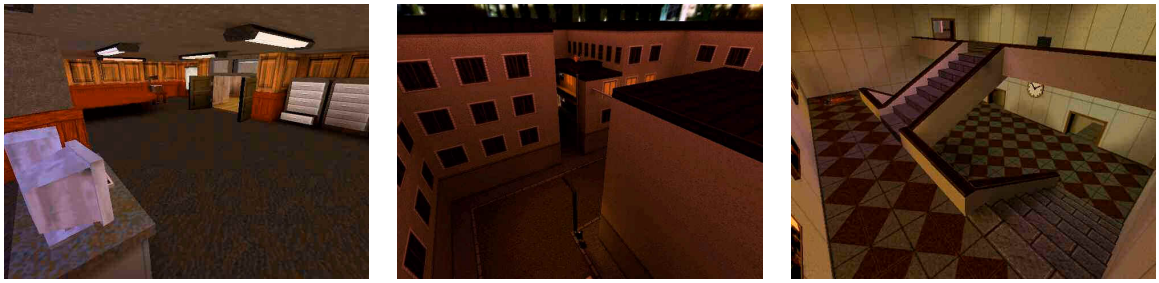
However, the muzzle velocity of the rifle grenade in CGF was chosen to be significantly lower (35 m/s) than the 'official' muzzle velocity (246 ft/s ~ 74 m/s). The muzzle velocity was reduced to maintain the balance between the game's weapons: at a lower muzzle velocity, the reach of the rifle grenade is shorter, and aiming the grenade is more difficult.

Below you'll find the key figures for recorded grenade and rifle trajectories for a number of maps:

Fig. 20 [a] The small, simple, urban outdoor Actcity2 map; [b] The medium size, city block outdoor/indoor Riotx map; [c] The large size, outdoor/indoor lighthouse map P1_Lightbeam.

- 'actcity2' is a small map, consisting of an urban setting with three squares connected by open roads, a few open buildings, and two closed buildings. It features mostly outdoor terrain.
- 'riotx' is a medium size map, representing a small city block with street areas, indoor areas and accessible flat roof tops.
- 'p1_lightbeam' is a medium size map, featuring a lighthouse, large accessible roof tops, squares and indoor hallways, rooms and tunnels.

Fig. 21 [a] Another look at 'p1_lightbeam', this time indoors; [b] The large size, open city block City map; [c] The large size indoor school building map Teacher.



- 'city' is a large size map, featuring wide open city blocks, balconies, rooftops, and a small number of rooms.
- 'teacher' is a large size map, featuring a large, 3 level school building with large hallways, a small theater, a sports hall, and a air vent tunnel network.

The table below lists the experiment results. An explanation of the table entries follows.

Fig. 22 Trajectory data and experiment results per map.

map / property	actcity2	riotx	p1_lightbeam	city	teacher
#waypoints (N)	473	673	811	951	1012
# doors	~ 6	~ 5	~ 10	~ 4	~ 33
hand grenades					
#trajectories	36781	58217	80068	84345	42870
avg #trajectory p. target	78	87	99	87	42
max # trajectory p. target	213	247	326	231	131
memory used	203 KB	318 KB	432 KB	460 KB	260 KB
avg trajectory size	5.7 B	5.6 B	5.5 B	5.6 B	6.2 B
memory / #waypoint	441 B	485 B	546 B	495 B	264 B
pre-computing					
processing time	3h:27m	6h:27m	7h:54m	6h:52m	8h:40m
time / (N*(N-1))	0.056 s	0.051 s	0.043 s	0.027s	0.030s
time / # trajectories	0.338 s	0.383 s	0.355 s	0.293s	0.628s
benchmarks					
"can throw?"	2.22e6 / s	2.12e6 / s	2.04e6 / s	2.08e6 s	2.32e6 s
"is appropriate?"	70300 / s	63800 / s	56300 / s	67700 / s	55800 / s
"attack position?"	6600 / s	6200 / s	4300 / s	5300 / s	3900 / s
rifle grenades					
#trajectories	49133	60596	127875	124131	49691
avg # trajectory p. target	104	90	158	131	49
max # trajectory p. target	245	227	360	320	149
memory used	263 KB	330 KB	665 KB	654 KB	294 KB
avg trajectory size	5.5 B	5.6 B	5.3 B	5.4 B	6.1 B
pre-computing					
processing time	93 s	169 s	353 s	269 s	277 s
comparison					
generating trajectories in-game	8.8 / s	8.5 / s	8.9 / s	6.7 / s	8.1 / s

Number of trajectories (#trajectories)
The trajectories have been computed with the map in its initial state. This means that most the doors present in the maps (few in any map but teacher) are closed (the default setting). This is the prime reason the number of trajectories recorded for teacher is this low. See 'Further Work' for a discussion about doors and trajectories.

Average trajectory size (avg trajectory size)
The average trajectory size consists of 5 bytes per trajectory plus the overhead of the lookup table data structures, amortized over all trajectories.

Memory per waypoint (mem / #way pnt)
The results suggest a memory consumption linear in the number of waypoints, thus linear in the accessible game world surface. Open structures and differences in height increase the number of trajectories, whereas water areas decrease the number of trajectories (grenades cannot be thrown properly in water).

Trajectory lookup benchmark ("can throw?")
The experiment consisted of 1 million 'does a feasible trajectory exist from waypoint N1 to waypoint N2?' queries, for random N1 and N2. The benchmark likely benefits from consistently hitting the L2 CPU cache, since the look up table fits in that cache[2].
Nevertheless, it illustrates that trajectory feasibility can be checked at virtually no cost, thus enabling the AI to test large numbers of waypoints for enabling an 'indirect fire' attack on a specific target.

Grenade use value benchmark ("is appropriate?")
The experiment consisted of 1 million assessments of the value of a grenade thrown from waypoint N1 to waypoint N2, for random pairs N1, N2 that have a valid trajectory (this being the normal case when used by the AI).
In the query, the target (N2) area is inspected (also for being an area used for ambushes, sniping, or frequent traffic), exits from this area are tested for being in N1's line of fire, and the alternative attacks from N1's position are considered. Finally, the presence of multiple threats (16 random threat positions are assumed) is tested. All these aspects are combined into a single rating.
All of this typically is done within 0.1 ms, for the largest Action Quake2 maps, enabling frequent use of this (high-level) query.

Grenade attack positions benchmark ("attack position?")
The experiment consisted of 1 million queries for a best 'grenade attack position' within 2 seconds travel time from waypoint N1 to attack a specific target at waypoint N2, for random pairs N1, N2 that have a valid trajectory (this being the worst case for the algorithm).
The query returns positions which are ranked for their distance from waypoint N1, for their concealment/cover from the attacked position N2, for having cover from the likely blast position, for their overall tactical rating, and for the amount of concealment along the path.
The performance measured is sufficient to enable the AI to check for grenade attack positions when appropriate.

Generating trajectories in-game
This benchmark shows the performance of in-game generation of trajectories, as opposed to pre-computing them. The experiment involved the generation of 1000 trajectories from waypoint N1 to waypoint N2, for random pairs N1, N2 that have a valid trajectory (this being the normal case!).
With slight variations in performance most likely caused by the differences in geometry complexity of the maps, the performance seems independent of the map size, and is about 0.1 s per trajectory.
The "time / (N*(N-1))" for pre-computing trajectories is less, mainly because during pre-computations, all targets Nx for source N1 are considered sequentially, thus enabling the caching of trajectory attempts. Such a situation does not likely occur in-game.
The "time / #trajectory" for pre-computing trajectories is higher, due to the extra efforts during pre-computing to obtain the most robust trajectory from a waypoint to another waypoint.

8 Further Work

Some additional work is needed to integrate AI grenade handling fully into the game, though large changes to the current design and implementation are not expected.

Specifically, the following issues need work:

- Doors
  Currently, trajectories are generated for the map as-is, thus without taking into account doors being able to open and close.

  Taking this into account requires a two-phase approach to generating trajectories: a first phase, with all doors closed, and a second phase in which all doors are open. In the trajectory record stored, a few flags already have been reserved to describe if a door is 'involved' in the trajectory, and whether the doors needs to be open or closed.

- Team level grenade handling
  For the team level AI to efficiently employ grenades, additional queries need to be developed to:
  - decide on the best candidate team member for a grenade attack; and
  - decide at team-level whether using a grenade makes sense

  In addition, proper team level responses to incoming grenades need to be added.

9 Conclusions

Bouncing grenades around the corner and tossing them through windows and holes clearly is within reach of game AI.

Moreover, AI that understands how to tactically use grenades can be implemented efficiently, even if it involves reasoning such as ‘does it make sense to use them in this situation?’, ‘what is the best attack position to launch grenades from?’

This paper first explores the feasibility of pre-computing grenade trajectories, and then illustrates the concepts behind tactical yet efficient reasoning about using grenades.

9.1 Pre-computing Trajectories

Generating hand grenade trajectories at run-time is expensive. Consuming about 0.05 s to 0.1 s of CPU time for a ‘human like’ bouncing trajectory, this is an order of magnitude too slow to enable the AI to frequently consider using a hand grenade.

Pre-computing a look up table of hand grenade trajectories leaves the AI more CPU time to actually think about the proper and tactical use of the grenade. These pre-computed trajectories, constituting a small subset of all trajectories possible between game world locations, need to representative: if there exists a feasible, safe, and relevant trajectory in the game world, the AI should be able to reconstruct this (or a similar) trajectory from the look-up table.

A dense grid of waypoints, covering all terrain accessible to the game actors, doubles as a backbone to compute and record representative trajectories. And this re-use of the waypoints facilitates tactical AI reasoning about attack positions and the grenade's effect on the threat.

The individual trajectory is best recorded as the direction to the first bounce spot. Using this bounce spot to aim the grenade is reasonably robust against deviations in the attackers exact position relative to the nearest waypoint. The direction towards the bounce spot can be recorded in polar coordinates, to reduce the amount of memory needed.

Given two locations in a game world, it is hard to analytically determine a feasible grenade trajectory. Therefor, a 'genetic algorithm' was developed to 'grow and select' trajectories. A fitness function determined the quality of the trajectory in terms of reaching the target, being robust, and looking 'human created'. The latter property proved to toughest to achieve.

Generating trajectories for a map is very time consuming, taking 4 to 9 hours (about as much time as it takes to perform radiosity lighting for that map). However, the problem lends itself well to parallel processing, and should not cause large problems for a game developer.

9.2 Memory Consumption

The look-up table of trajectories mainly consists of trajectories, and a few Kbytes of overhead. To store an individual trajectory some 4 or 5 bytes are need.

For a single type of grenade (such as the hand grenade), the amount of memory required varies from 200KB for a small game world to almost 500KB for the larger Quake2 game worlds. The amount of memory required is a more or less linear function of the accessible game world surface.

The most effective way to reduce the memory consumption is to filter out more trajectories. For example, trajectories that never would make sense, either tactically, or within the game (rules). A level design tool can easily be extended to annotate certain areas of the game world as being inappropriate for using grenades.

The trajectory look-up tables are organized per target waypoint, thereby enabling the AI to quickly retrieve feasible attack positions.

9.3 CPU Utilization

The highest CPU utilization seems to occur at the Teacher and P1_Lightbeam maps.

An estimate of the CPU utilization for a worst-case 16-on-16 engagement, with every AI actor involved, plenty of grenades, and always a means to attack using hand grenades, is:

Fig. 23 CPU utilization for worst-case 16-on-16 engagement, using benchmark results

budget			AI activity for 32 actors
20.0 ms			total AI grenade handling activity
measured			**based on benchmark results**
0.1 ms		320	checks for the existence of a grenade trajectory
1.2 ms		64	checks for grenades being appropriate
16.4 ms		64	checks for the best grenade attack position
0.8 ms	+	6	aiming attempts
18.4 ms			estimated CPU load for 32 actors

This estimate (18.4 ms CPU time per second) is within the CPU utilization budget (20 ms per second) set for this part of the AI.

9.4 Tactical grenade-related AI

High-level yet efficient tactical reasoning is possible once the look-up table for grenade trajectories is in place (together with a waypoint oriented 'database' describing the terrain).

Considering whether

- a grenade thrown into an area would likely flush out threats from that area into the line of fire;
- the threat would likely evade the blast;
- there are other ways to attack the threat with direct fire

altogether takes the AI less than 0.1 millisecond of CPU time on 'standard' hardware[b].

The AI has available the following basic grenade related queries to (pro-actively) employ grenades:

- a simple check if from a specific waypoint, a target position can be engaged using a grenade;
- an assessment of the value of attacking a target position with a grenade from a specific position;
- the generation of tactically sound launch positions nearby a specific position, to attack a target position with a grenade

These queries, combined with the existing CGF tactical queries dealing with tactical movement planning, threat lines of fire, etc., enable an AI to better emulate the behavior of a trained soldier, challenging opponent, or reliable and realistic team member.

9.5 Limitations and complications

Whereas pre-computed 'terrain reasoning information' may enable more advanced game AI, it also introduces complications.

Pre-computed terrain information assumes a fairly static game world. Dynamic parts of the world, such as doors, vehicles, destructible terrain, or temporary smoke clouds are tough to "catch" in look-up tables.

If the number of these exceptions is not that large, they can be dealt with by means of special entries in the look up table, or with temporary patches of the look-up table.

If the game requires a strongly dynamic world (typically also consuming more rendering CPU time), the AI design needs to be either less ambitious, or requires more CPU and memory.

The size terrain reasoning databases typically scales with the size of the terrain. For games aiming to provide a vast terrain in combination with slow moving individual combatants and long distance combat, such as NovaLogic's Delta-Force series, the claim on memory might be too heavy. More work is required in developing data representations that efficiently capture positions and associated (long-range) visibility, cover and concealment.

10 References

- [Snook]
  Snook, Greg, Simplified 3D Movement and Pathfinding Using Navigation Meshes in *Game Programming Gems*, M. Deloura (ed.), Charles River Media, 2000
  Good overview on terrain representations for AI in 3D game worlds, and their use for navigation purposes

- [Reece]
  Reece, D., Krauss, M., Dumanoir, P., Tactical Movement Planning for Individual Combatants, in *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*, 2000
  One of the few papers dealing with tactical path finding. Describes the relation between various terrain representations and tactical path finding. The approach by Reece, Krauss, and Dumanior has similarities with the tactical path finding implemented for CGF, though their focus is more on supporting large outdoor and indoor standardized terrain databases, whereas CGF is geared towards smaller 'worlds', and low CPU utilization

- [Half-LifeSDK]
  Half-Life SDK 2.1, Valve Software, 2000
  Half-Life (AI by Steve Bond) has the enemy NPC use grenades in a simple way: check if the trajectory straight to last observed position of the player is feasible. No intended bouncing. No explicit clue about the grenade's area damage effect. No reasoning about the value of using grenades. No notion of moving elsewhere to throw a grenade. The AI does check for nearby squad members that might get hurt.
  The NPC cheat by having available an unlimited amount of hand grenades.

- [MCWP 3-35.3]
  MCWP 3-35.3: Military Operations on Urbanized Terrain (MOUT) , US Marine Corps Doctrine Division, 1998
  Appendix A, addressing fundamental combat skills, emphasizes grenades being thrown via windows, or through holes, something most (combat simulator or tactical shooter) game AI's are incapable of.

- [Laird]
  Laird, J.E., It knows what you're going to do: Adding anticipation to a Quakebot in *AAAI 2001 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*, March 2000, AAAI Technical Report SS-00-02
  Quake2 bot, external to the game, based on the SOAR-agent framework. Creates detailed model of opponents, predicts opponent behavior and uses that in it tactics, thereby outpacing commercial game AI. Implemented in 750 rules. A single AI character seems to consume 10% of a 400Mhz PentiumIII (about the total budget for a few dozen CGF bots). The AI primarily is designed for one-on-one combat. The anticipation illustrated likely loses value in a more noisy, frantic deathmatch among 20 agents. The QuakeBot's generic, domain-independent, approach to prediction is a nice contrast to the domain-specific performance oriented implementation of tactical concepts presented in this paper.

---

[a] The simple solution to grenade handling is to have the AI randomly 'just' throw grenades (in a straight line) to the last visible location of a threat. Optionally, have the AI cheat and use the correct velocity to deliver the grenade 'on target', even though the player is limited to a fixed launch velocity.
This solution has been implemented in many tactical shooters, since 1998

[b] PentiumIII@500/100Mhz (Katmai, 512KB slow L2 cache), 128MB SDRAM, NVidia TNT2 Ultra graphics, Windows2000. C++/C code compiled with MSVC6, release mode (optimized).

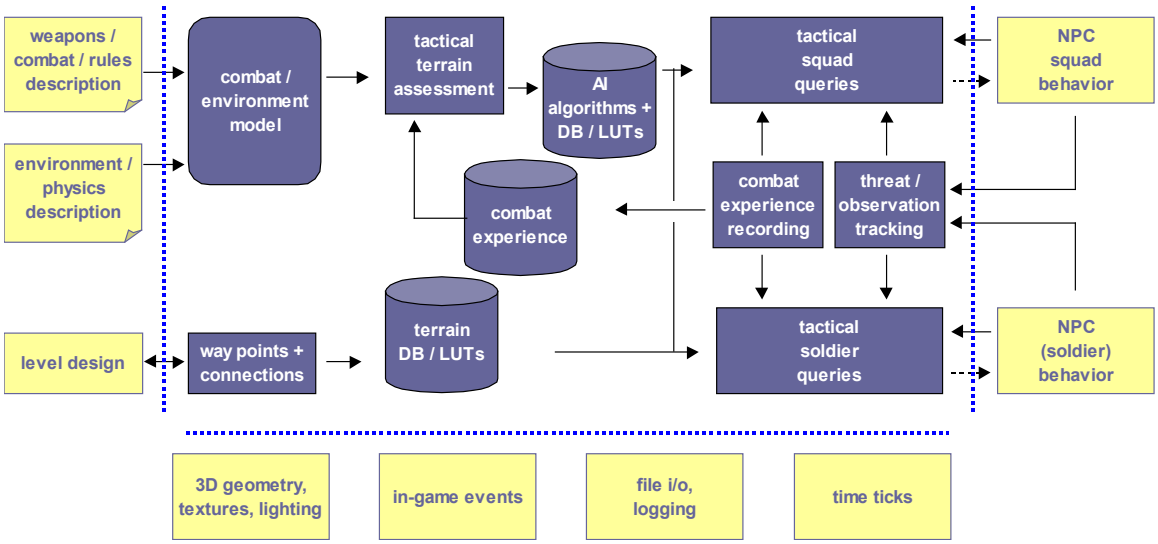
[c] Key in getting the AI to adapt to human tactics (instead of homing in to the flaws of the AI behavior) is to value human behavior more than AI behavior. It is not that easy to get this right.
You can download the CGF 0.81 (December '99) release to test it.
Recently, at the 2000 Game Developer Conference Game AI round table, AI mechanisms that recorded experience and knowledge "in" their environment were dubbed 'smart terrain'

A About CGF-AI

The grenade handling AI presented in this paper is developed within the CGF project. CGF is a project developing AI and AI subsystems for autonomous fire-team and squad level tactics. The AI is prototyped using modifications of 3D 'open' games, thereby also benefiting from the 25,000+ gamers that played CGF.

The CGF releases in 1999 are unique in providing autonomous yet scriptable teams, bounding overwatch team maneuvers, combat experience recording and adaptive tactics, smart environment scanning, terrain understanding, emergent combat patterns, and tactical movement planning.

Fig. 24 The AI focus on combat, terrain, tactics, soldiers, and teams illustrated. Most of the terrain database content is computed off-line, enabling the AI in-game to perform higher level reasoning more efficiently.



In 2000, the terrain reasoning subsystem was overhauled for better performance and to handle larger terrain. In addition, terrain classification, ambush planning were added.

The CGF AI itself (without game logic) is some 140 KLOC of C++ and C, using plenty of heuristics, FSM's, fuzzy logic, reinforcement learning and (off-line only) genetic algorithms.

B About Action Quake2

Action Quake2 (AQ2) is a user multi-player modification of the Id Software game Quake II. AQ2 aims to bring 'realistic action movie style' combat. It provides more lethal ('single shot kills') and realistic weapons (different weapon modes, sniper scopes, laser sights, clip based reloading), and bandaging. These new rules transformed Quake II into a tactical 'think on your feet' shooter game.

Action Quake2 evolved to become one of the top 3 most popular variants of Quake II, supported by over several hundreds of user-created maps.

The CGF AI, mentioned in this paper, is an independent effort to add AI to Action Quake 2.