Managing Pacing in Procedural Levels in Warframe

Daniel Brewer

1 Introduction

Warframe is a cooperative online action game where players take on the role of space ninjas with machine guns and perform missions in procedurally generated levels. Over time players gain experience and improve their characters, allowing them to take on more challenging and numerous foes.

Since the levels are procedural, players will never experience the exact same level layout. While this keeps the experience fresh, the changing size and layout means that traditional hand-crafted level-design tricks, such as trigger volumes and spawn scripts, cannot be relied upon to control the pacing. Instead, *Warframe* uses an AI Director to understand the structure of the procedural level, monitor the intensity of the action around the players, and track their progress (Brewer 2013). With this information, the system can then intelligently spawn enemies to provide an interesting and appropriate challenge for the players.

In this article, we will first describe the method used to generate the procedural levels in *Warframe*. We will then discuss the data structures used by the AI Director to understand the structure and flow of the generated level, how we measure the intensity of the action around the players, and how we can control the pacing at runtime. Finally, we will cover how designers can work with the AI Director to handle some of the special cases in the game.

2 Procedural Level Generation

Warframe levels are generated by connecting pre-made blocks. Each block is crafted by a designer with a number of external portals that act as sockets to connect the blocks together. Only compatible portals of the same size can be connected, as shown in Figure 1. For example, a 5x3 portal can only connect to another 5x3 portal and cannot be connected to a wide, 9x3 portal. Level blocks can be rotated and transformed as necessary to line up compatible portals.

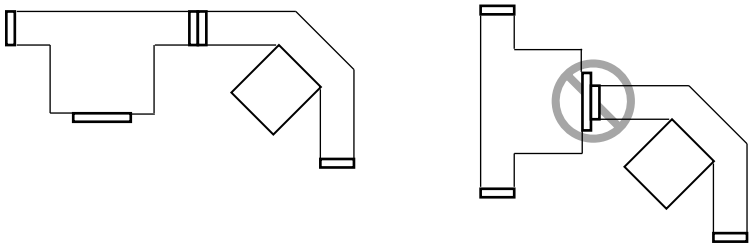


Figure 1 Blocks can only be connected via compatible portals. Left shows a valid connection between compatible portals of the same size. Right shows an invalid connection between portals of different sizes.

Each block is categorized by a block-type that specifies its appropriate use. These block-types include *Start*, *Connector*, *Intermediate*, *Objective*, and *Exit*. Levels always begin with a *Start* block and end with an *Exit* block. *Connector*s are smaller blocks that link the larger blocks together and space them apart. Finally, *Intermediate*s are large, more complex combat spaces. A segment is then a series of connected level blocks. Since most levels have only one objective, we typically have two segments, one between the *Start* and the *Objective*, and one between the *Objective* and the *Exit*. Segments can be a random length and are made up of a combination of *Connector* and *Intermediate* blocks. Using the first letters of each block type, an average level might be represented as the string, "SCICOCICE", or a longer one as "SCICICOCCICCE".

For each mission, designers configure a procedural level template that prescribes how the level should be generated at runtime. The template lists which blocks are available and their block-types, as well as specifies the length and composition of each segment, as well as the maximum branching depth. The layout generator will then use this template to create a sequence of block-types. For each block-type, the generator shuffles and selects a block at random, connecting it to the previously selected blocks to create the playable level. The generator needs to ensure this level is valid and does not overlap itself, as illustrated in Figure 2. To do so, a depth-first, recursive search is done, and if a block cannot be placed in a valid position, we unwind the previous step and try a different selection. The algorithm is described in pseudo code in Listing 1.

To reduce the chance that a procedural level template fails to produce any valid levels, we use an automated test to attempt to generate thousands of layouts. This allows us to detect if we fail to generate any complete layouts, or if an excessive number of rollbacks

are required to find a valid layout. When necessary to improve the success rate, level designers either modify the portal locations or shape of existing blocks, or they create new blocks that increase the variety of blocks available to generate the level. As a general rule, we recommend a pool size that allows the algorithm to select approximately 15-20% of the available blocks for any particular generated layout. As you might expect, more complex blocks with bends or loopbacks can cause layout problems as they further constrain the set of viable neighbors. Adding more blocks that simply have portals opposite each other can help to create distance between blocks to avoid overlaps. We also try to keep to only one or two portal types per procedural template as using too many different size portals can result in too few valid options when trying to add a block.

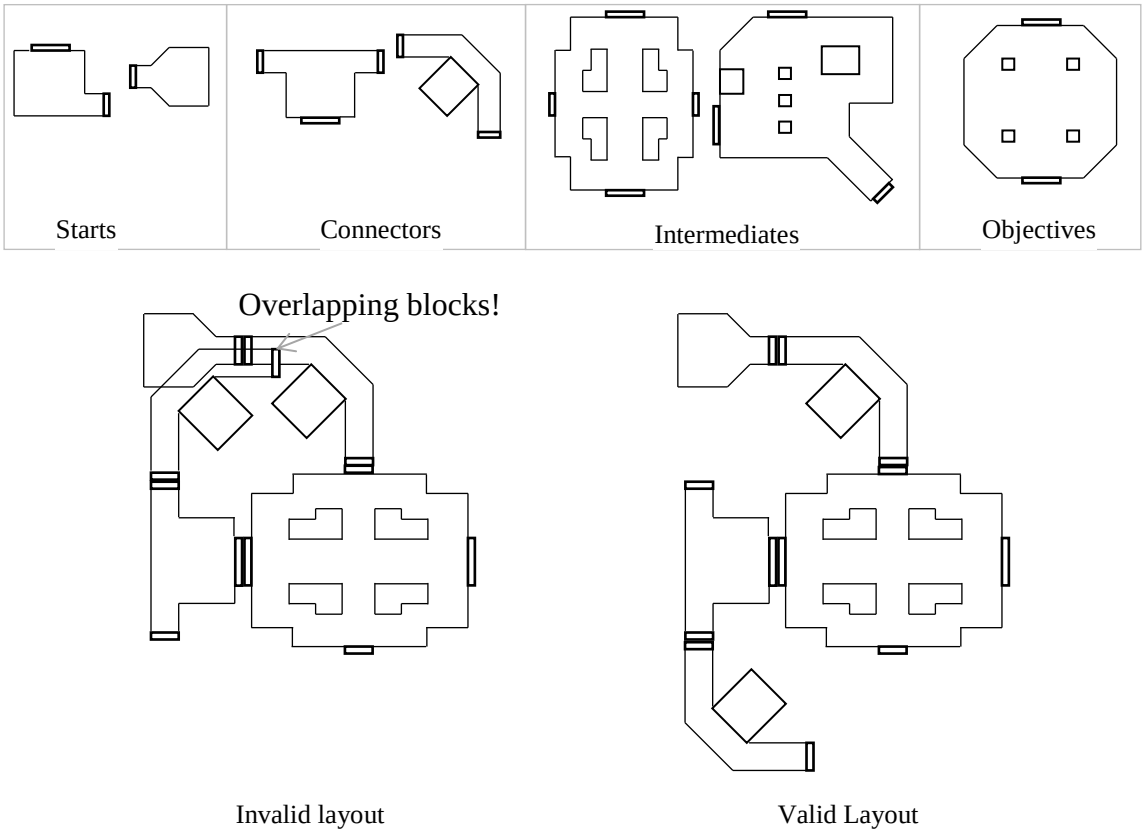


Figure 2 Blocks are categorized into types such as *Start*, *Connector*, *Intermediate* and *Objective*. When attempting to place a block in the level layout, no overlap with existing blocks is permitted.

By adjusting how the open-portals list is constructed, it is possible to control how many branches are generated. If the open-portals list is restricted to the last block placed, this will result in a linear level with no branching, similar to pushing the blocks onto the end of an array. Allowing the open-portals list to include unconnected portals from previously placed blocks, however, allows the new block to generate a branch off another, earlier block.

Once the layout is complete, there may still be portals that do not connect to anything. Caps are placed on these portals to seal the level and door fittings are placed on

portals that connect blocks together, covering up any seams.

Designers specify different templates for each mission and environment. An "Exterminate" mission on a spaceship, for example, will have its own layout template that will be different from a "Sabotage" mission on a planetary outpost. The layout template ensures that only the appropriate blocks for the mission are selected and prevents selecting an "Assassinate Objective" block for a "Sabotage" mission, or a spaceship block for an outdoor level.

Listing 1 Pseudo code for the algorithm to generate levels for Warframe.

```
Generate blockTypeQueue, e.g. SCICOCCE
if PlaceNextBlock(0, blockTypeQueue, placedBlocks):
  #we have a successful layout but now we need to cap off
  #any remaining open portals and add doors to the portals
  #that join placed blocks
  for each block in placedBlocks:
    for each portal on block:
      if portal is open:
        push (cap, portal) onto placedBlocks
      else:
        push (doorFitting, portal) onto placedBlocks
  return placedBlocks
return emptylevel

PlaceNextBlock(depth, blockTypeQueue, placedBlocks):
  if depth >= blockTypeQueue.size:
    #all blocks placed successfully
    return true
  currentBlockType = blockTypeQueue[depth]
  availableBlocks = get list of blocks of currentBlockType
  shuffle availableBlocks
  for each testBlock in availableBlocks:
    if TryPlaceBlock(testBlock, placedBlocks):
      return PlaceNextBlock(depth+1,
                            blockTypeQueue,
                            placedBlocks)
  #failed to place any valid block at this depth
  #we need to undo our last attempt and roll-back
  pop (lastBlock, lastPortal) off back of placedBlocks
  mark lastPortal as open
  return false

TryPlaceBlock(testBlock, placedBlocks):
  if placedBlocks is empty:
    push (testBlock, root) onto placedBlocks
    return true
```

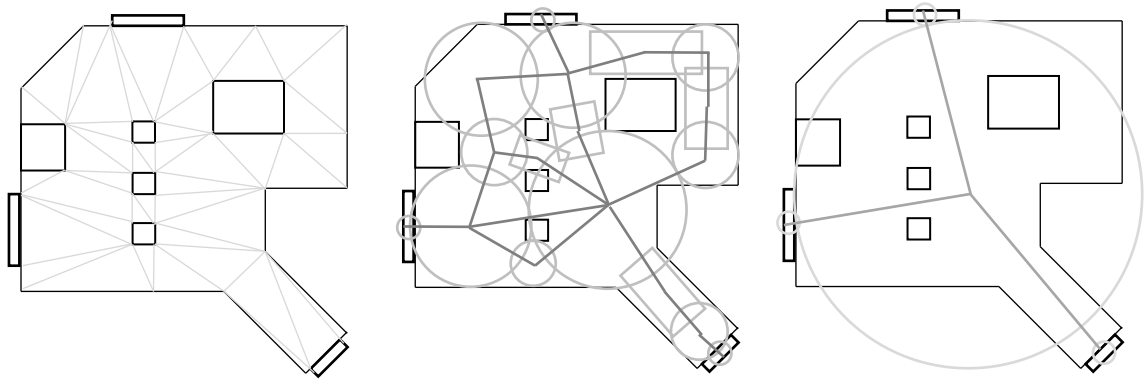
```
  for each block in placedBlocks:
    for each portal on block:
      if portal is open:
        push portal onto openPortalList
  shuffle openPortalList
  for each openPortal in openPortalList:
    if CanPlaceBlockOnPortal(testBlock,
                             openPortal,
                             placedBlocks):
      push (testBlock, openPortal) onto placedBlocks
      mark openPortal closed
      return true
  return false

CanPlaceBlockOnPortal(testBlock, openPortal, placedBlocks):
  for each testPortal on testBlock:
    if testPortal is compatible with openPortal:
      try place testBlock connecting testPortal to openPortal
      if testBlock does not overlap any placedBlocks:
        return true
  return false
```

3 Understanding Level Structure

For the AI Director to make pacing decisions at runtime, it needs a way to reason about the structure and flow of any generated level. We need to identify which way the players should go to reach their objective, if they are currently moving towards or away from this objective, and where enemies should spawn to ensure they have a meaningful impact on gameplay. To accomplish this, we use a tool called the Tactical Area Map or "TacMap".

The *TacMap* is essentially a rough corridor map. It is a graph structure that represents the structure and flow through the level, as shown in Figure 3, that can be used by the AI Director. Each node in the graph is referred to as an area. The connections between areas represent how one may move from area to area through the level. A *TacMap* is created offline for each level block. At runtime, after generating the procedural layout, all the individual *TacMap*s are then connected together into a single graph that represents the structure of the entire level. Doors and connections between blocks are marked up in the *TacMap* so we can identify chokepoints and make decisions based on whether the doors are locked. The *TacMap* is coarser than the navigation mesh used for path finding, but finer than the level blocks (see Figure 3) and provides a practical level of detail to make tactical decisions.



NavMesh with 44 nodes | *TacMap* with 17 nodes | Level Block with 3 portals

Figure 3. Comparing the granularity of the *TacMap* to the navigation mesh and level block.

We often need to search for entities in the level, such as spawn points, cover locations, alarm panels, et cetera. To do so, we add references to these entities to the areas in the *TacMap* graph. Critically, this allows the searches to follow the structure of the level and not just straight-line distance.

Various forms of influence maps are used to track the runtime properties that the AI Director can use to make pacing decisions. Rather than a traditional 2D grid, the influence maps are setup on the *TacMap* to allow influence to both accumulate in areas and spread to adjacent areas through the connections in the graph, which represents the actual flow through the level. As an added advantage, since the *TacMap* is in 3D and not restricted to a 2D plane, it can also handle verticality.

Depending on the game, there are a number of potentially useful influence maps you may want to consider. Below we discuss four of the more important maps, namely the distance map, player-influence map, active-area map, and the visibility map.

3.1 Distance Map

The distance map is generated by seeding the area containing the objective with a distance of zero and then performing a Dijkstra flood out from the object to fill the entire graph along each connection between areas of the *TacMap*. This provides a reasonable representation of the actual distance players are required to travel, which may not be a straight line, and only needs to be updated when the objectives change.

Once calculated, the distance map provides the Director with the players' progress through the level and their remaining distance to the objective. By noting which neighboring areas have lower distance scores, the Director knows the implied direction of travel toward the objective. We can then determine whether they are moving towards or away from this objective, by tracking the players' distance to the objective over time. The distance map can also be used to bias the enemy cover selection algorithm to encourage enemies to take up positions between players and the objective.

3.2 Player-Influence Map

The player-influence map is seeded by setting a high influence in areas containing players and then having the values decay linearly as they spread out from the players. This map is updated multiple times a second and we use both the current value and difference with the previous value. If the difference is positive, it means that the player influence is increasing, caused by players moving towards this area. Likewise, if the difference is negative, the player influence is decreasing as a result of players moving away from this area. This allows us to select spawn locations for enemies ahead of the players, even if the players are exploring and moving away from the objective. This means the enemies are far more likely to be encountered by the players and have an impact on gameplay. If we spawned enemies behind players as they moved through the level, there would be a high chance that those enemies would never be encountered and would simply be a waste of resources.

3.3 Active Area Map

Since levels can be sprawling, an active-area map allows us to maintain a bubble of activity around the players that moves with them as they progress. Enemies within the Active Areas are considered relevant to the gameplay, while enemies outside this bubble can be deactivated, destroyed or teleported to new positions to keep pressure on the players without wasting resources.

To calculate the Active Area, any areas within the same block as the players or within neighboring blocks are marked with an active score. The scores in all other areas decay over time until they reach zero, at which point the areas are considered inactive. This decay time provides a short delay before deactivating enemies in case the players are moving back and forth across block boundaries.

3.4 Visibility Map

When the *TacMap* is built offline, the approximate visibility between areas is calculated by casting many rays between areas. The size of the area determines the number of rays required as they are distributed evenly across each area. If all the rays are blocked, we can assume there is no line of sight between these Areas. If some of the rays are unhindered, we can assume that the areas are potentially visible to each other. These results are encoded in a lookup table.

At runtime, a visibility map is then calculated on the entire *TacMap* using this visibility approximation lookup table and based on the players' view cones. Areas that are visible to players are marked with a score, which decay over time. From the perspective of the Director, any area with a positive score might be visible to the players or may have recently been seen. Having the score decay over time provides some hysteresis, thus if players rapidly change direction this will prevent enemies from spawning behind the players' backs as they turn around.

Additionally, we initialize all visibility scores to -1 at the start. This indicates that they have not yet been seen by a player. This can allow the AI Director to continue to spawn enemies in a room out of sight from the players until players have visited that room and can also provide a metric for how much of the level has been seen and cleared by the players.

4 Measuring Intensity

In order to control pacing, we first need to measure it. However, pacing is difficult to quantify: How much action is going on around the players? How engaged are the players? Do we need to spawn more enemies or should we let off for a bit and allow the players to collect themselves before continuing? In addition, players in *Warframe* gain experience by completing missions and upgrading their characters and equipment over time. Thus, an encounter that may have challenged them before might be too simple when the players play the mission again later.

While we are always experimenting with more advanced intensity measurement calculations, often simplicity is best. To that end, every time the player takes damage, her intensity is raised. This amount is proportional to the normalized ratio of damage received to her total health and shields. Every time the player kills an enemy, her intensity value rises by an amount proportional to the distance from the player to that enemy. If a player is being targeted by an enemy, we maintain the intensity, otherwise it decays over time. This is a similar scoring metric to that used in Left 4 Dead (Booth 2009).

While not a perfect score, it reasonably estimates that players who are killing enemies or taking damage themselves are most likely in the thick of the action compared to players who are not receiving any damage and not killing any enemies.

5 Controlling Pacing

Now that we have a measure for the intensity of the action around the players, we can attempt to shape the play experience. If we graph the intensity over time, we want this graph to have a rhythmic rise and fall pattern similar to a roller-coaster. As the players explore and encounter enemies, a fire-fight ensues, causing a rise in intensity. After the players dispatch the enemies, the intensity dies down until the players encounter more enemies and the intensity starts rising again, as shown in Figure 4.

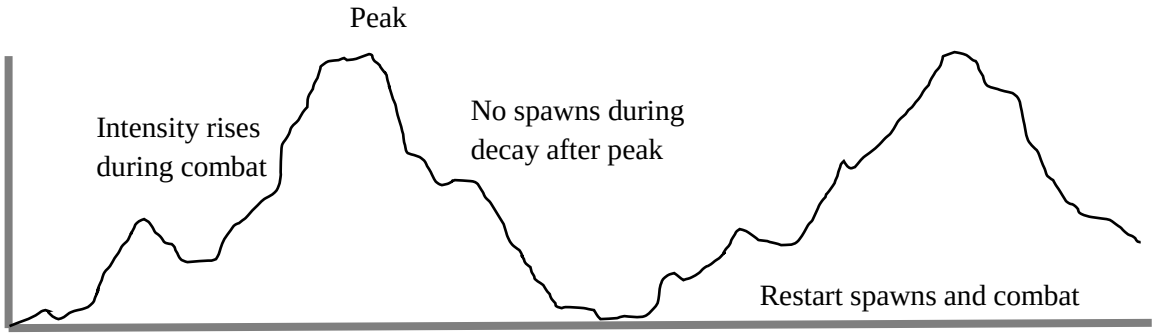


Figure 4: Intensity will rise and fall during gameplay. If intensity reaches 100, it has peaked and we hold off spawning more enemies until the intensity dies down, giving the players a

breather after intense combat.

Our primary tool for controlling the pacing in *Warframe* is spawning enemies. While intensity is low, the AI Director should spawn more enemies ahead of the players, and should continue to spawn enemies as intensity rises, until it reaches the maximum supported value or peak. Once intensity peaks, we want to maintain the peak for short time and then back off and stop spawning until the players have had a chance to dispatch all the enemies and recover from the fight. The intensity then decays down until it reaches zero and the spawning process begins again. This should drive the intensity back up again as the players encounter the next group of enemies.

Varying the number of active enemies around the players allows us to control the experience. As players are naturally attracted to conflict, we can thus subtly lead the players towards the objective by having more enemies spawn if players are heading towards the objective and fewer if they are moving away from the objective. If the players have not yet been discovered, we use a lower limit on active enemy count. This allows our space ninjas to perform some stealth gameplay and sneak through the few patrolling guards. Once the players have been spotted and the alarm sounded, we want increased enemy activity and therefore the active limit is increased.

Finally, the limit also depends on the type of block. *Connector* blocks tend to be smaller and so have lower limits. *Intermediate* blocks, representing larger combat areas, have higher limits and the *Objective* block will have the highest limit. Since our procedural layout generation typically produces alternating connector and intermediate blocks leading up to an objective or exit block, this helps modulate the enemy numbers and provides a satisfying rhythm.

6 Spawning Enemies

After deciding to spawn, the AI Director then needs to decide which enemy and where to create them. The mission will specify a list of available enemy types, their selection probability and an optional maximum simultaneous limit. Using a weighted random selection allows the AI Director to spawn many ordinary grunts and fewer specialist or rare units. Having a maximum simultaneous limit for certain enemy types ensures players will not stumble into a room full of extremely tough enemies and instead will encounter them in more reasonable numbers.

Once we've selected the type of enemy to spawn, we want to determine where that enemy will have the biggest impact on gameplay. There is no use spawning an enemy near the start of the level if the players have already completed the objective and are heading to the exit. We also don't want to spawn enemies in a room recently vacated by the players. To avoid breaking immersion, enemies should typically not spawn in plain sight of the players. In general, an enemy should be created ahead of the players and close to them, but out of sight.

Certain levels and enemies allow for special-case spawning. Examples are the robotic deployment containers that have a special deployment animation when the container opens and activates the robotic unit housed inside. In this case it is preferable to use a

special spawn point that is visible to the players, so they can witness the arrival event. The AI Director first tries to find a special spawn points for the enemy, but if none are available, it will fall back on selecting a normal, hidden spawn point.

The algorithm used by the AI Director to search through the *TacMap* for spawn points is a breadth-first search, described in Listing 2. This search starts at the areas containing the players and spreads out, keeping areas within the given constraints and discarding unsuitable ones. Next, we collect valid spawn points from each valid area. Since the search for valid areas expands out from the player, the list of valid spawn points will be approximately sorted by increasing distance from the players. We select a point with a Gaussian random number biased towards 0, which will prefer points closer to the players instead of ones further away.

Listing 2 Algorithm used to search the TacMap for spawn points for an enemy. The call to `GaussianRand` will generate a random number in the range from a Gaussian distribution with the mean at 0 and the standard deviation of 0.5.

```
Add players' areas to open list
While open list is not empty:
  Pop area off front of open list
  If area has not yet been visited:
    Mark area as visited
    If area is valid to keep:
      Push area onto keep list
    If area is valid to expand:
      For each neighbor of area in the TacMap:
        Push neighbor onto back of open list

For each area in keep list:
  For each spawnPoint in area:
    If spawnPoint is enabled and not in use:
      Push spawnPoint onto availableSpawns list

SelectedSpawnIdx = GaussianRand(0, availableSpawns.size)
Return availableSpawns[SelectedSpawnIdx]
```

7 Mission Specific Complications

The previous sections cover most situations handled by the AI Director. Some missions require some special case handling, however, which we describe below.

7.1 Extermination

Extermination missions require the players to progress through a linear, non-branching gauntlet and dispatch a specific number of enemies. The AI Director needs to not only ensure that the required number of enemies spawn, but that they all have spawned before the players have reached the end of the level, and that they are spread out through the level and not clumped at the beginning or at the end. Additionally, we do not want the concentration of enemies to be completely even, as this would be dull. Instead, there should ideally be

groups of enemies and some predetermined peaks of activity.

The first step is to determine the number of enemies required by the mission, keeping in mind a reasonable density for the intended experience. If we require too many enemies in a tiny level, we will need to spawn too many simultaneously in order to have them all created before reaching the exit. Alternatively too few in a large level would result in very sparse, uninteresting pacing. To achieve a reasonable density, we calculate the total number to spawn based on the maximum distance to the objective, as read from the distance map, ensuring a reasonable number of enemies for the size of the level.

A population graph is used to compare progress through the level with percentage of total population required to have spawned by this point. The slope of this graph represents the density of enemies encountered. The steeper the slope, the more enemies will be encountered, the flatter the slope, the fewer enemies. As shown in Figure 5, we use a piecewise linear graph with two peaks to ensure all the enemies spawn and that the players encounter a good variety of concentrations of enemies on their way through the level.

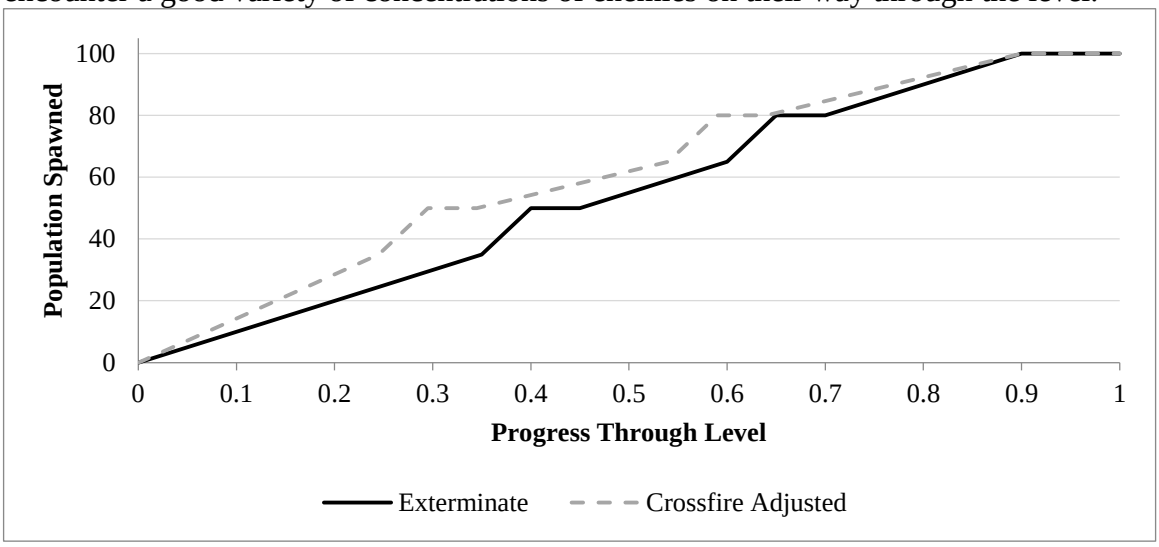


Figure 5: The population graph for exterminate missions is fixed and provides two peaks of activity to ensure all enemies spawn before reaching the end of the level. For crossfire missions, the graph is shifted to match the position of the combat front in the level.

7.2 Crossfire

Crossfire missions operate just like Exterminate missions, except the players ally with one faction to oppose a common enemy. An example is a boarding action between capital ships of rival factions. The players start on one ship, and fight their way across onto the other. One of the blocks in the level represents the main combat front where the enemies are clashing. It would be disappointing if the players were to arrive at these battle lines only to discover a lull in activity.

Instead, to overcome the vagaries of the procedural layout, we tweak the predetermined population graph based on the actual layout. First we look up the combat front in the level's distance map. We shift the second peak in our graph to this point to ensure it lines up with the features of the level. We then shift our first peak halfway between the start and the previously adjusted peak. This modified graph as shown in Figure 5, now

matches the level layout better, and ensures the action is where we want it to be.

7.3 Survival / Excavation

Survival missions require almost constant, escalating pressure as the players try to survive for as long as possible, while moving around the map to respond to new objectives.

Although the AI Director's default pacing is designed more for exploring a level, it works in conjunction with mission scripts by providing a number of functions that the scripts can use to query the environment and spawn enemies. To help the Director support a constant flood of enemies to harass the players, mission scripts can be customized to override the spawn point search filters. Thus, in survival missions, spawns can be selected closer to the players and all around them.

Enemies need to rush to attack the players, so having the flexibility to alter the spawn selection is useful. There is little point spawning an enemy in a position where they cannot navigate to the players, such as on the opposite side of locked doors.

The mission script also overrides the normal Director pacing decisions, instead monitoring the active enemy count directly and making explicit requests to the AI Director to spawn more enemies as required. In this mission, the search filter is provided by the script, though the AI Director still handles the selection of enemy type and performs the search for spawn points. The designers can escalate the intensity over time by setting the script to request simpler enemies at the beginning and introduce more challenging enemies as time progresses.

When it comes to selecting the next player objective, the AI Director provides functions to search through the *TacMap* to find objectives matching the requirements, such as distance from players and other active objectives.

8 Conclusion

As games embrace procedural generation, we can't rely on traditional designer scripted set-pieces. We need systemic and procedural pacing solutions. The AI Director in *Warframe* is one approach to this problem. It handles the dynamic pacing requirements of standard missions on procedural levels by monitoring the intensity of the action around the players, tracking their progress through the level, and reasoning about the flow and structure of the level at runtime in order to provide a compelling gameplay experience.

There will always be times when a fully automated system cannot handle specific design requirements, but in these situations most of the information required is maintained by the AI Director and the *TacMap*. This allows designers to script special case encounters in a systemic fashion, without having to worry about the specifics of the level layout, which can vary at runtime over repeat play-throughs of a mission and as players level up their characters.

9 References

Booth, M. 2009. From COUNTER-STRIKE to LEFT 4 DEAD: Creating replayable cooperative experiences. *GDC 2009*, San Francisco, https://www.gdcvault.com/play/1422/From-COUNTER-STRIKE-to-LEFT

Brewer, D., Cheng, A, Dumas, R., Laidacker, A. AI Postmortems: *Assassin's Creed III*, *XCOM: Enemy Unknown*, and *Warframe*. *GDC 2013*, San Francisco, https://www.gdcvault.com/play/1018058/AI-Postmortems-Assassin-s-Creed

10 Biography

Daniel Brewer graduated in 2000 with a BScEng in Electronic Engineering, focusing on Artificial Intelligence, Control Systems and Data Communications. Since then he has become a veteran game developer. As lead AI programmer at Digital Extremes, he has been instrumental in bringing the co-op, online, multiplayer action shooter, Warframe(2013) to life, as well as continuing to further develop, maintain and upgrade the game AI systems in the game. Other notable titles include Halo 4 multiplayer DLC packages(2012), Darkness II (2012), BioShock 2 multiplayer (2010) and Dark Sector (2008). Over the years, he has presented numerous talks about various facets of AI at the Game Developers Conference.