# Project Milestone 10

**CS 4500 Software Development**

**Due:** Friday, April 23, 11:59pm

**Submission:**

1. Create a release on Github tagged `p10`, with executables and supporting files. See the individual descriptions for details.
2. Submit a ZIP on Handins with the relevant source code files and any additional files requested in the individual descriptions.

**Also see the bit about this milestone and final walks.**

---

This milestone offers a more than one route to credit. The total number of points available is 100, split in half between regular and extra credit: 50 is for regular credit, the remainder is extra credit. This means you can earn up to double the percentage allocated to this Milestone in extra credit. You can choose any combination of the below options.

| | | |
|---|---|---|
| 1 | Multi-game Server | 40 points |
| 2 | Hit Point & Combat System | 50 points |
| 3 | Remote Adversaries | 60 points |
| 4 | Procedural Level Generation | 100 points |

The total is capped at 100. For example, if you implement remote adversaries "perfectly", you'll get 60 out of 50 points, that is full regular credit + 20% of extra credit. Implementing features 2 and 3 fully yields 110 points, after capping 100, meaning full regular and full extra credit.

## Option 1: Multi-game Server

The SNARL server as implemented in Milestone 9 only supports a single game. Extend the server to support multiple SNARL games. Each game should start following the same rules as in Milestone 9 (at least one player, start when four players connect or timeout lapses). For simplicity, you can run games in sequence (a game starts after the previous one finished). If you implement a multi-threaded server, we will give extra points above the allocation specified at the beginning. In addition to the `(end-game)` message, the server should also send a running leaderboard of players from the games finished so far (the JSON format is up to you). The server should have a mechanism for the administrator to shut it down after any ongoing games finish. This can be via a GUI or by accepting an `exit` command on standard input in the console.

### Deliverables

Included in `p10-exec.zip`:

1. An updated version of the `snarlServer` executable, called `snarlServer1`.
2. An updated version of the `snarlClient` executable, called `snarlClient1`. The client should be extended with a rendering of the server's leaderboard at the end of a game.
3. A `README.md` explaining how to run a multi-game server, how it behaves and how to shut it down. Make reasonable decisions about command line arguments.

Included in the Handins zip:

1. A document, `Snarl/planning/multi-game.md`, describing the design and implementation of the multi-game server, including any necessary changes to the SNARL protocol.
2. Source code for updated and added components, and the above executables.

### Evaluation

Full credit can be earned for demonstrably satisfying the given requirements. If something seems underspecified, make a reasonable decision and explain it.

## Option 2: Hit Point & Combat System

Give players and adversaries hit points, allowing them to engage in combat. That means a player shouldn't be expelled immediately upon contact with an adversary, but should be able to sustain some damage before being expelled. The player may be be able to hit the adversary back. Make a reasonable choice on the damage delivered by a ghost and a zombie, with respect to the player's total hit points.

### Deliverables

Included in `p10-exec.zip`:

1. An updated version of the `snarlServer` executable, called `snarlServer2`.
2. An updated version of the `snarlClient` executable, called `snarlClient2`.
3. A `README.md` explaining how to run the above executables and a players' guide to the new combat/hit point system.

Included in the Handins zip:

1. A document, `Snarl/planning/combat.md`, describing the design and implementation of the combat system, including any necessary extensions to the SNARL protocol. You should include an explanation of how the system works and the rationale behind the hit-point management.
2. Source code for updated and added components, and the above executables.

### Evaluation

The evaluation is based on the quality of the implementation, not the properties of the system itself. Full credit can be earned for implementing a well-designed hit-point management system. If something seems underspecified, make a reasonable decision and explain it.

## Option 3: Remote Adversaries

For this option, extend the SNARL protocol to handle remote adversaries. Take care not to affect existing interaction between players and the server: existing player clients should be still able to play the game as before. This means you need to consider how adversaries connect to the server and distinguish themselves from player clients. Remote adversaries should be used by the game to fill the role of a ghost or zombie when needed. Provide implementations of a remote Zombie and a remote Ghost (these can be the same executable). You do not need to support an arbitrary number of remote adversary connections, but at least two should be able to participate in the game. If more adversaries are needed than connected, their roles should be filled by local instances.

### Deliverables

Included in `p10-exec.zip`:

1. An updated version of the `snarlServer` executable, called `snarlServer3`.
2. Executable(s) for adversary client(s).These should take the same `--address` and `--port` arguments as `snarlClient` in Milestone 9.
3. A `README.md` explaining how to run the above executables. In particular how to start the server, connect some number of adversaries and players.

Included in the Handins zip:

1. A document, `Snarl/planning/remote-adversary.md`, describing the implementation (added/changed components), specifying extensions to the SNARL protocol and explaining how backward compatibility is ensured.
2. Source code for the updated SNARL server, the remote adversary components, and the above executables.

### Evaluation

Full credit can be earned for demonstrably satisfying the given requirements. If something is underspecified, make a reasonable decision and explain it.

## Option 4: Procedural Level Generation

For this option, implement a procedural level generator. To get the full 100 points (that is, regular + extra credit), implement a generator based on this Gamasutra article. The article describes a slightly modified version of a procedure used in *TinyKeep*. It is not the best written piece of text and does not contain enough detail to be a tutorial, so you will have to do some research on your own. It contains minimal amounts of code examples in Lua, which should be relatively understandable. Here's the gist, with Ferd's comments.

The procedure can be split up into the following steps:

1. Generate randomly sized rooms at random points within a circle of a given radius.

   Code for generating a random point inside a circle is provided and should be translatable to your language easily. Ignore the bit about grid alignment: our coordinates are already in terms of grid tiles, not pixels. Note, that the function as given centers the circle at $(0, 0)$. I use a relatively small radius.

   The dungeon described in the article uses "intermediate rooms" instead of hallways and so they generate many of them, then identify "main rooms". We have separate hallways and so only have a few main rooms.

   It's enough if you use a simple rectangular layout with walls around the perimeter of the room and no inner layout.

2. Separate rooms by moving them apart until they don't overlap.

   The article recommends using a physics engine instead of separation behavior. I recommend using separation behavior instead of a physics engine (unless you can get a physics library running on Khoury), which is relatively easy to implement. A description can be found here, but Google should provide a few additional results. I can share my separation computation function in Haskell if desired. Experiment with different overlap thresholds to see what gives best results in terms of space and distribution – we want to leave enough room for hallways.

   I recommend against simply evenly moving the rooms from the center of the circle – you might end up with a huge empty level. Separation behavior gives pretty nice results.

3. Generate a connected graph out of rooms, taking the topology into consideration.

   We now have a collection of disconnected rooms and need to figure out which rooms should be connected by hallways.

   This step creates a graph out of the level, by generating potential connections. This is probably the trickiest part in the article. They use Delaunay Triangulation. Implementing the algorithm on your own can be quite challenging and I strongly suggest finding an implementation online that you can adapt, or a library that provides the functionality. I used a library.

   Instead of using Delaunay, you could experiment with finding "good neighbors" on a room-by-room basis: for each room, search for a few rooms that are closest and don't have another room between them. If you come up with a good replacement for Delaunay for this step, let me know – I'm curious.

4. Reduce the number of connections in the graph.

   The graph generated in the step above is connected, which is what we want (we want to be able to reach all rooms), but it is *too* connected. The article suggests generating a *minimum spanning tree*, but I find using a simple breadth-first traversal to generate a spanning tree works just as well. With BFT the initial room will become a "hub" for the level. This step will ensure that all rooms are reachable with a minimum number of connections, which is perhaps too few. Therefore add a few of the removed connections back.

   Note that for a small number of rooms (e.g., 5) the difference between the graph resulting from step 3 and the spanning tree is small, so this step might be skipped for small levels.

5. Generate hallways between connected rooms.

   This is the last step. For each pair of connected rooms, generate a hallway. If the rooms are aligned, a hallway can be a simple line (no waypoints). If they are not, you'll need to introduce a waypoint or two. Feel free to get more creative and randomly introduce additional waypoints, but make sure hallways don't overlap.

   Again, ignore the parts that talk about non-main/hub rooms. We don't have those.

Let's be clear, getting full 100 points on this option might be challenging. If it helps with estimating the time needed, I implemented the first 4 steps above in under a day, though it's currently not the most readable code.

You can implement your own, perhaps simpler, approach to level generation, or use only certain parts of the above process, but the grade will be determined by comparison to what we outlined above and we will look at how your implementation addresses the same concerns (randomly sized and spaced, non-overlapping rooms; connectedness; non-overlapping hallways). Any implementation should satisfy the following additional requirements:

1. The generation should be parametrized by:

   a) the number of rooms and

   b) the minimum and maximum dimensions of rooms.

2. It should reliably generate valid levels with at least 20 rooms.

3. It should place 1 key and 1 exit in two random but different rooms, on random traversable non-door tiles. If only one room is generated, place both objects in the same room.

## Deliverables

Included in `p10-exec.zip`:

1. An updated version of the `snarlServer` executable, called `snarlServer4` with an additional argument, `--generate`, which, instead of using a local level file, will generate new levels as the game progresses.

2. A standalone executable `snarlGen`, with the following optional arguments:

- `--rooms INT`, where `INT` is a positive integer specifying the number of rooms in the level. Default: 5.
- `--min [ROWS, COLS]`, specifying the minimum possible dimensions in the generated level. Default: `[4, 4]`.
- `--max [ROWS, COLS]`, specifying the maximum dimensions of a room. Default: `[15, 15]`.
- `--json`, specifying that a JSON level representation should be printed to standard output. The JSON should follow the format for **`(level)`** specified in Milestone 4.
- `--render`, specifying that a preview of the level should be rendered to the screen.

When `--json` is used, the executable should be runnable purely in a terminal.

If none of `--render` or `--json` is given, behave as if the parameter was `--render`.

3. A `README.md` explaining how to run the above executables.

Included in the Handins zip:

1. A document, `Snarl/planning/generation.md` describing the implementation details of your level generation, deviations from the above procedure, and any issues you have run into.

2. Source code for the level generation component and the above executables.

#### Evaluation

For full 100 points (regular + extra), implement the above generation steps. An alternative approach to generation is acceptable and we will award at least 50 points to a working generator that reliably generates a valid random level with 1-20 rooms in under a minute on Khoury Linux (provided it satisfies the additional requirements and the deliverables are submitted).

## This Milestone and Final Walks

We will review this milestone during final walks, either the finished work or work-in-progress. This will give you an opportunity to demonstrate your implementation and explain the choices, and help us with getting you the grade sooner.

If you have scheduled a final walk before you are able to submit a full solution, create an additional "work-in-progress release" tagged `final-walk` on Github before 9am on the day of your walk. Once the work is submitted, we will review the submitted work in light of our discussion during the final walk and might reach out with any final clarifications via email.