# Better Tower Defense – Project Plan Semester 2

Student: Jacob Dufault (jacobdufault@gmail.com)

Faculty sponsor: Professor Bernhard

## Project Goals (same as semester 1)

This is a tower defense game that will focus on excellent multiplayer, though there will also be a well-developed campaign. There are a large number of innovations within the tower-defense genre, some new to games as a whole and some pulled from other game genres. Some of the major innovations are as follows:

One of major problems with tower defense games involves quickly overpowering content; the player gets bored as the game takes five or six turns to catch up with player's progress. A new dynamic difficulty system will be implemented where the engine automatically scales the difficulty of the content depending on how well the player does.

Another area of concern in traditional tower defense games concerns the resource system, where resources are gathered via killing enemies. This has issues in multiplayer (particularly where one player grows more and more powerful until they are the only one that can kill enemies). An economic system where the player manages an economy is proposed. This additionally gives the player more to do when killing waves.

Significant strategic depth is provided through the effect system, which allows for "effects", such as burning or freezing, to be applied to entities. The strategy arises because effects in dependent upon one another to determine their effect on the object, whereas the classic solution is that they are independent.

The entity representation within the engine be a novel implementation. High level benefits of this system include simple parallelization, removal of the update order problem, and removal of event dispatching.

## Technical Challenges (same as semester 1)

The tower defense genre was chosen for technical reasons; it should be reasonable to implement as compared to an RTS / FPS / RPG within the senior project timeframe. Compared to an RTS, tower defense games have a number of technical simplifications; path finding can be a simple A* implementation (though locomotion will remain the same), the number of units is relatively lower (a couple hundred at absolute most), and the AI is significantly simpler, among other simplifications. However, there are still some technical issues, such as the need for a lock-step networking model.

### Dynamic Difficulty (same as semester 1)

When playing a tower defense game, the player can create a very efficient setup of towers that simply outpaces the difficulty of the waves. Due to the static nature of how waves are defined, such as spawn 10 monsters, then 15, then 20, other tower-defense games do not respond well to this situation.

Instead, a "dynamic difficulty" system for waves can be implemented. The level will contain the classic setup for spawning monsters, but based on analyzing gameplay the game can adapt and dynamically increase difficulty. Some potential methods for doing analysis are presented as follows:

- average enemy health
- average enemy health variance

- average enemy lifespan
- average enemy clustering
- average income and wealth
- total number of towers

The dynamic difficulty system will not decrease difficulty, as this can be easily abused. For example, the user could have trouble with a mission. The user wants to win quickly and knows about the dynamic difficulty system; he/she wants to decrease difficulty, so they do poorly in the first few waves, which will cause the difficulty to go down – there will be no challenge to the user. This removes the required element of strategy from the game.

## Resource System (same as semester 1)

After having placed towers, the user is essentially forced to slowly watch the action unfold and make minor variations to their tower layout. In effect, this is an extremely boring part of tower defense games. The resource system is novel in that it gives the player something to do during this phase of gameplay.

Fundamentally, resources are used when the player builds towers and sends units at the enemy. However, how does the user acquire resources? In a classic tower defense game, the user acquires them via killing enemies. This has a number of problems, most important of which is a snowball effect in multiplayer when one teammate starts killing more units than the other players; he/she will then gain more resources, letting him get better towers, kill *even more* units, ….

In response, resources will be acquired via a simulated economy that the player manipulates. The player will have a “base” of sorts which they will build; for example, they will build a farming system to support a population and a set of houses; next, they will build a marketplace to foster trade, and finally a castle to increase worker effectiveness, just as an example. Depending on time constraints and implementation details, this system can even modify gameplay itself, for example, how frequently powers are allocated. While monsters are being killed the player will manage this system.

## Effect System (same as semester 1)

Effects are used to modify the state of entities within the world. For example, there can be a SlowEffect which reduces the speed of an Entity. However, this is not the novel portion of the effect system.

Towers interact with enemies only through effects. For example, a tower applies a “Burn” effect to an object. When the burn effect is active by itself, it reduces the health of the entity by a couple percent. However, the user has another tower placed which applied a frost effect which is meant to slow down the enemy. When this tower attacks, the frost effect is applied but an object cannot both be burning and frozen, so the effects cancel each other out. The end result is that the frost effect canceled out the burning effect – the enemy is unaffected by either tower.

To reiterate, the action that the effects execute on the target object are *dependent* upon the other effects that are active on the target object.

## Powers (same as semester 1)

One of the annoying mechanics with tower defense games are “leaks”; when you kill all but one unit in a wave. This problem is inherit within the game itself, but mechanics can be designed to make this an enjoyable (and not a frustrating) part of the game.

The user will have “powers” that they can invoke. For example, they can use some money to apply a burn power that applies a burn effect to a single enemy, or an oil power which causes oil to spill over a region of terrain; each enemy that passes through that region then has the oil effect applied to it.

These powers can then be used to deal with leaks without forcing the user to disrupt their tower building strategy.

### Entity System (novel implementation) (same as semester 1)

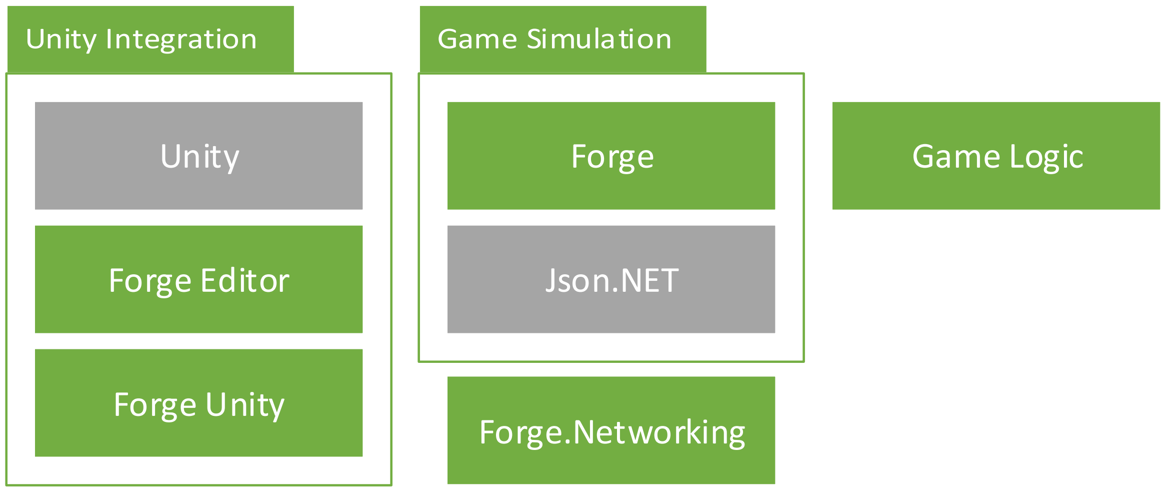
The gaming industry is currently in the middle of an engine design shift towards component-based entities versus the more traditional inheritance based entity system. The key difference here is that component-based entities are composition based while inheritance based entity systems are inheritance based.

However, the gaming industry has not gone far enough: there are many more innovations awaiting in this design. *Artemis* is a pioneering entity system which extends the entity-component system to an entity-component-system model; entities are still just collections of components, but components no longer contain any logic within them. Instead, systems now contain logic and operate upon sets of components which contain a certain required set of components.

*Artemis* provides a much more powerful model, but even it does not go far enough. *Artemis* does not solve the "update order" problem; that is, the world can end in a different state depending on the order that systems are updated in. This is a problem for concurrency – systems (which do almost all of the work in game (excluding rendering, which is already on another thread)) cannot be automatically parallelized to different CPU cores, or if it is parallelized, it has to be *extremely* carefully managed. For clarity, this is a problem for concurrency because of the lock-step networking model: simulations on all computers have to be *identical*.

An addition taken from functional programming is applied to *Artemis.* Systems are viewed as state transformation functions which map one state of the game to the next state; they do not directly modify the current game state (which means that the order in which they are applied does not matter). The two most recent game states are stored and systems can listen for differences between them. This completely removes the need for Events within the game; for example, a common event type is a "Position update"; instead, a system can just request to be notified when the state of the position component has changed and will then have access to both the previous and the current position, automatically with no additional code required from the position component. Further, objects which modify the position are completely oblivious to this dependency between the system and the position, increasing decoupling and decreasing required code maintenance.

## Design



**Unity**: The Unity game engine, by Unity Technologies

**Forge Editor:** A set of editor plugins for Unity that simplify editing Forge content

**Forge Unity:** Integration between the Forge entity framework and Unity

**Forge:** The Forge entity framework itself. Has lots of advanced features, most importantly deterministic game simulation. Automates multithreading.

**Serialization**: The serialization framework. Previously a propriety library, now Json.NET.

**Forge.Networking:** Networking code (such as a chat system and a turn system) built on top of Lidgren.Network.

**Game Logic:** Game simulation code. This depends only on Forge and is rendering engine agnostic.

## Progress Summary

The entity framework has gone through another round of polishing; many bugs were fixed. Serialization works extremely well now with a clean design. An editing framework was developed that allows more general component models than the one shipped in Unity. Networking was rewritten to be more modular.

## Milestones

By the end of the semester a working game will be implemented.

### 1 (February 19th)

- Resource system
    - Extends building placement logic
    - Each player needs their own resources
    - Define relationships among buildings and how resources are gathered
- Power system
    - Activate effects in a specific region on the map
- Add a number of tower types to test the flexibility of the effects system
    - Direct AOE Tower: Create an area effect every n turns that has an effect that deals direct damage.
    - DOT AOE Tower: Create an area effect every n turns that slowly damages enemies that are within the affected area.
    - Direct Tower: Emit a random effect every n turns
    - Multicast Tower: Emit a random effect every n turns to m units

### 2 (March 19th)

- Implement dynamic difficulty system
    - Will require analysis on how the player is currently performing
        - Average kills per minute
        - Average wealth
        - Average time spawned unit is alive
        - Number of towers
        - Total worth of base
- Implement different level types
    - Primarily impacts how units are spawned
    - Player versus player where each player controls spawning

- Create poster for senior design showcase

### 3 (April 16th)

- Implement more unit types
    - Interesting events on death (spawn more units)
    - Go invisible
    - Only targetable by certain types of towers
    - Specific resistances
    - Affect nearby towers
    - Affect nearby units
- Create demo of the game
- Create a user manual explaining core concepts of Forge and of the game

## Approval

“I have discussed with the team and approve this project plan. I will evaluate the progress and assign a grade for each of the three milestones.”

Signature and Date: ______________________________________________________________