# Agar-Agar: A Procedurally Generated Game

**Allen Zheng (az5782)**

CS378H: Computer Graphics
University of Texas
Austin, TX

# 1 Features

The goal of this assignment is to create a procedural-generated game reminiscent of *agar.io* with in-game mechanics honoring classic Graphics techniques learned in this class. Since the generation is simpler than that of 3D-Minecraft, focus is also placed on *what* is generated along with the *how*.

## 1.1 Perspective

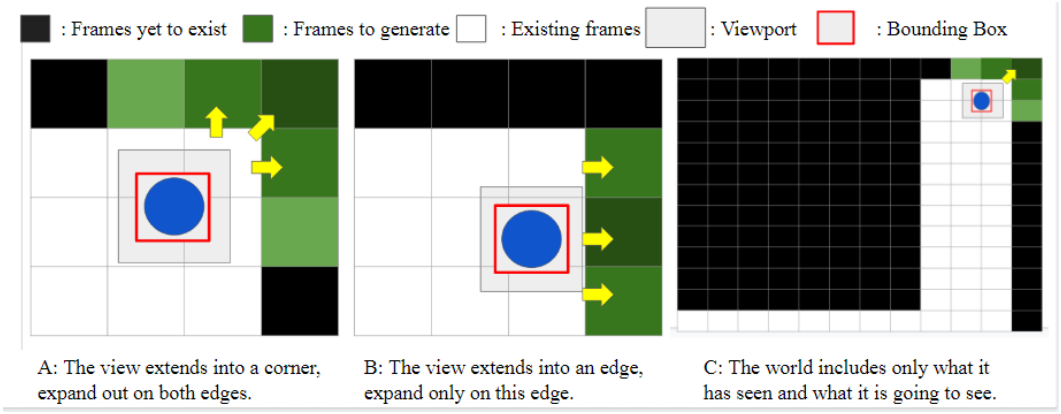
Perspective is bird's eye view, and stays fixed on the center of the screen, where the blobs' center of mass resides. The perspective scales relative to the player's size. While a player may appear to get bigger, on the Viewport it occupies the same pixels as the world *around* it gets smaller.

## 1.2 Movement

Movement follows the mouse ray, and the perspective moves along with the player's blobs' center of mass. The player can split itself with binary fission on an axis orthogonal to that of the mouse. Individual blob components want to converge toward the mouse, so its velocity must be relative to the *secant* ($\frac{1}{cos}$) betwixt the player's velocity. Linear Interpolation allows for "smooth" movement.

## 1.3 World Generation

The world's grid is represented by a `Map` (amortized $O(1)$ access) of individual grid components (called `Frame`). Nearby grid frames are automatically whenever the implementation detects that the viewport is on an edge, but not the entire dimensions of the world.




The above figure shows the two cases (A, B) of expansion as the viewport (and player) moves near world corners and edges. The number of `Frames` increases at a constant rate (linearly), as opposed to expanding the dimensions of a matrix (which grows in $n^2$). On C, we see that roughly two-thirds of the space is not yet needed that would've otherwise been generated.

## 1.4 `Frame` Components

Each `Frame` creates and stores its objects internally upon creation. Instead of the entire world having to update itself, we need to check the only `Frames` within the Bounding Box of the player's blobs. Similarly, only the frames within the Viewport need to be shown. Frames may consist of the following, whose properties depend on their `Biome` (chosen upon generation):

- Food, which the player can eat (upon collision) to grow larger.
- Hazards, which interact with the player's blobs upon collision. Breakers will automatically split it among an axis of collision, mirrors will reflect it for a short period equal to the angle of incidence, and black holes will induce a gravitational pull ($F_g = \frac{Gm_1m_2}{r^2}$) on the blob.

## 1.5 NPCs

The point of the game is to conquer other players to become the largest blob. This offline version supports other CPU-players that move and act randomly. Along with splitting, players can also move faster and reflect off other hostile blobs at a cost. Unlike world objects, NPCs are not frame-local; they are automatically generated on a global scope, and their updates are performed together with the actual player's. Non-players may be constrained within the world's limits or extend the world themselves (which should be the case when the game moves online to multiplayer).

## 2 Solution Design

### 2.1 Setup and References

The coding environment I used was the `p5.js` library. This appeals to me because the setup is straightforward – there's not too many obstacles before I can get something to show up on screen, which allows me to focus on my back-end logic more. From this library, I mostly use drawing functions and the occasional `Vector` operation. To get started, I watched Daniel Schiffman's (The Coding Train) videos on `p5.js` tutorials and a very basic *agar.io*. The video covers basic movement of one blob through a static global environment. I set up this coding environment after watching that video, so initially our code would've looked similar (see README and `test/test1`. I've added a lot more features as described in Section 1; all of the other logic comes from me.

### 2.2 `app.js`

This file is the primary handler of the display in the HTML window, through drawing each frame and calling other files to update their structures between frames. It also registers user commands and does all necessary initialization of data structures and images. The `show()` and `update()` are overloaded functions on many subsequent files that ultimately trace their calls back to this one.

### 2.3 `window.js`

This file contains the `Window` and `Frame` classes. The former describes an instance of the game's world, which includes its players and grid `Map`, the latter describes local attributes of the grid's individual monomer, like its biome, food particles, and course hazards. This separation allows the locality described in Section 1.4 to be exploited rather than using all global updates.

### 2.4 `biomes.js`

This contains the data for the `Biome` of a `Frame`, which includes its background texture, food density, and hazard properties. All `Biome` objects are preloaded in a `Biomes` object at the beginning of the program.

### 2.5 `blob.js` and `player.js`

These files describe `Blob` and `Player` classes, which are separate entities due to the nature of *agar.io*: a player comprises of its individual `component` blobs. Interactions with the world are mostly seen through `Blobs`, but the `Player` facilitates their actions along with their game state. The NPCs (`NonPlayer` and `NonPlayerBlob`) and food particles (`FoodBlob`) also inherit from these classes.

### 2.6 `hazards.js`

This file contains the abstract `Hazard`, which is the parent class of `Breaker`, `Mirror`, `BlackHole`. Each of these receive their attributes when their `Frame` container is constructed with a describing `Biome`, and they each override a `interact(blob)` function.

### 2.7 `fields.js`

This file consists of useful global variables and enums. The game contains many configurable flags and settings, such as the automatic spawning of NPCs or a mass factor to increase the force of a `BlackHole`.

### 2.8 Known Bugs and Issues

1. Angle measurements can lead to imprecise behavior near critical regions of some angles, such as close to 0 in $sec(x)$ and $cot(x)$. As a result, special exceptions are made if the angle is within a certain epsilon in a way more consistent with the game mechanics, but this could be handled more elegantly.
2. The game, due to its randomized spawning, may also suffer from "spawn-killing": a player spawning within radius of another player or black hole. There are naive ways to fix this but will cause the performance to suffer (continuously respawning, or evaluating all the hazards before). This issue and the former issue combined will sometimes cause a player to spawn within a mirror, which causes it to infinitely reflect its center.
3. Merge time occasionally doesn't register until another `split()` is performed, and it also includes time that the game is paused. Merging itself could be done more smoothly; at times it feels like a jump.