# Devlog: Windowsilk

Welcome to my devlog for the game **Windowsilk**.

A local multiplayer game for 1-X players, playing spiders moving across an increasingly chaotic web, catching bugs and avoiding falling off their web.

The game was made for the **Github Game Off (2021)** game jam, which had the theme **“BUG”.**

As is required by that jam, the full source code and assets for the game is available on GitHub: <TO DO: Link>

That’s also the reason why I won’t put any code in this devlog, and only explain the general process (with images and short descriptions of the algorithms).

## What’s the idea?

When the game jam theme was announced, I was finishing up my previous game: **Carving Pumpkins & Dwarfing Dumplings** <TO DO: Link>.

In that game, every player is a *shape* that can throw knives. When a knife hits your opponent, it slices through them (splitting them into two, realistically). If you’ve become too small, you’re dead and out of the round.

It has a **Halloween** theme, which caused me to write down some loose ideas for a “Bat”-related arena and a “Spider”-related arena.

When creating a mockup for the Spider arena, I got this idea: “What if players could *only move over the web itself?*”

With the follow-up idea: “And what if those knives could *cut* through strands of the web, dropping players to their death (if you aimed well)?”

The arena never made it into the game. (Too complex, too different from the rest, and I ran out of time.) But the idea was nice.

And when I saw this jam’s theme was BUG, I knew I had to make this right now.

The starting idea, which is the first thing I always write down when creating the project, was:

**Players are spiders. Level is a spider web. You can only move over the web, but you can *jump* to other locations to create new lines.** (As we all know, spiders shoot silk out of their butts. So jumping to a different line would create a new connection between the two.) **Your objective is to stay alive and/or collect a certain amount of bugs**.

The objective was still a bit vague. But I had enough of a main idea to already get started.

## Step 1: What’s a Spider Web?

What’s a spider web? It’s

* A collection of *lines*
* Which meet at certain *points*

I created two scenes: **Edge** and **Point.**

An Edge knows these things:

* Its extremes (start and end point, no particular order)
* The entities currently on it

It uses its extremes to draw a rectangle ( + physics body in the same shape) from one point to the other.

A Point knows these things:

* The edges connected to it
* The entities currently on it

It simply draws a circle with the same thickness as the lines. At least for now.

To create a web, I just need to create Points, and then tell it to create edges between two given points.

But that’s where the first issues already start: **what if lines overlap?** What if I create a new edge that crosses through an existing edge?

In those cases, we want to create a *new point* where the two edges overlap, and split the old edge in two. (This keeps the web consistent and easy to traverse: no overlapping edges, you can choose a different direction to travel at teach point.)

TO DO: Image => Devlog1

## Step 2: Weaving a Web

Notice how I drew an *arrow* at the end of the line. Why? Because I realized something: **we will never need to create a new line between two specific points.**

**Instead, we want to simply create a line from point A to *the first point it hits in a certain direction*.**

If you have any experience with game development, you’ll immediately think**: ray casts!**

I wrote a function that does the following:

* Given a starting point and direction …
* Cast a ray from the point, in that direction.
* This returns the first edge that it hits.
* Create a new point at that location + split the edge
* Create a new edge from the starting point to the new point

TO DO: Image => Devlog2

Using this, the player can jump in any direction, and it will accurately move the player *and* create a new edge from start to end, growing the spider web.

(In this case, the starting point is the *player position* and the direction is *the direction input by the player joystick/arrow keys*.)

The nice thing is we can *also* use this for creating the spider web in the first place. We need *some* structure to start with, *some* lines for the players to stand on when the game starts.

Well … I simply call this function for all corners (top-left corner to bottom-right, top-right corner to bottom-left) and we get a basic web, with all the correct points and edges.

**Remark:** I created four bodies as the “walls” of the level. These are positioned exactly at the edge of the screen. Any raycast that hits no edge, will hit one of these bounds and place a point there. Then the algorithm is simply the same.

**Remark:** if the point I want to create is *really close* to an existing point, I don’t create a new one. I just snap to the existing point and use that. Prevents creating a mess *and* saves us calculations. (In fact, if *both* points already exist and have an edge between them, I obviously can ignore this whole algorithm and just move the player without doing anything else.)

## Step 3: Walking over it

### The general idea

How do we make players walk over the web?

* They need to know where they are now.
* If it’s an edge, they can only move along it. (Either forward or backward.)
* If it’s a point, they should pick a new edge (from the point) to go next.

Lines are nice, because they are simple and predictable. You can only move up or down. There are easy algorithms to check if a point lies on a line segment.

So this is how it works:

* Start the player on any edge. (Just pick one from the starting web we created.)
* Anytime we move over it, check if we are now *out of bounds*. (We’ve exceeded the end points of the line. I’ll explain the “moving” part soon.)
* If so, we should be at one of the extremes of the line: either the start or end point. Find out which one it is.
* The next time we move, pick the best edge around this point. Set it as our current edge.
* Repeat.

Players constantly alternate between edges and points. (Even when you jump. Because, as you can see in the earlier image, you will create a new point and land there.)

On edges, you simply follow them. On points, you try to find the next best edge.

### Moving from point->edge

How does that work?

* Calculate the *vector* for each edge. (Subtract our current point from the end point of the edge.)
* Calculate the *dot product* between that vector and our *movement input* vector.
* The option with the highest dot product is chosen as the best edge, as it most closely aligns our movement input.

If you don’t know what a vector is: it’s an arrow with a certain *direction* and *size*. For example, if I want my character to move one unit to the right, I add the vector (1,0) each frame, which stands for (x = 1, y = 0).

The dot product (between two vectors) is a simple formula with this consequence:

* A value of 1 means the vectors are identical. (They point in the same direction.)
* A value of -1 means they are exactly opposite.
* A value of 0 means they are orthogonal. (Their direction is 90 degrees from each other.)

This means that a *higher* dot product, means two vectors are *more alike*. And that’s why we want the edge with the highest dot product.

*Remark:* the values I gave above are only true if both vectors are *normalized*, which means that their size is exactly 1. If not, the rule of “higher = more alike” is still true, the values are simply different.

TO DO: Image => Devlog3

### Moving along an edge

This is very similar to the idea above.

* Calculate the vector for the edge.
* Calculate the dot product between that *vector* and our *movement input vector*.
* If it’s above 0, it means we just move along the edge vector. (We were already oriented correctly and just go forward).
* If it’s below 0, it means we are the wrong way around and go backward. (Simply negate the edge vector by putting a minus sign in front of it.)

Hopefully you can also see *why* this is true. If the dot product is above 0, it means our movement input is more alike the edge vector than the reverse of it. So we move forward, instead of backward.

TO DO: Image => Devlog4

### Important remarks

**Remark #1:** computers have floating point imprecisions. I can place a point at position (10,0), and it’s actually placed at (9.99998,0).

This means that the algorithm for checking if a point is on a line segment *might, on rare occasions, fail*. Players might move off the web. In the current state, they would be locked out of this system and can’t play any further.

To mediate this problem, I use quite a large “margin” in the algorithm. You can be a bit *off* and still be considered *on the line*. It loses some visual precision, but ensures gameplay will work.

In the future I will need to code some more robust “fail-safe” that can, if needed, always snap players to the nearest point/edge. But that’s a worry for later.

**Remark #2:** when you enter a point I, purposely, *freeze player movement* for 100-300 milliseconds. (Still need to find the best value.)

Why? If I don’t do this, you need to pick your new direction *the exact moment you enter a point*. This is really hard to do for players. Also because you can’t clearly *see* when you enter a new point.

By freezing you for a fraction of a second, you have time to pick your new direction and press the keys (or move the joystick) to mirror that. It makes moving along this web *way more smooth and enjoyable*.

## Step 4: The actual game

I was surprised how easy it was to create this and how well it worked. It’s already *fun* to move around it, jump to new strings, and build the web as you go.

This created some doubt. I envisioned this as a *competitive party-like game* (where you battle against each other over control across the web). But it might be better as a *puzzle game* or as a *cooperative* *game*.

In these situations, I simply continue working on things I *am* sure I need.

These things are:

* Something to track how much silk you have. This controls how far you can jump (and if you can jump at all).
* A small jump animation/tween.
* Different “types” of silk. (Which would be a variable in the code, and a color/icon over the edges. These are the equivalent of “terrain types” from most games.)
* A system that places “stuff” on the level. It should be able to (purposefully) place it *on* the web or *off* it. Why? Player should always be able to grab something useful, but it shouldn’t be too easy – they need to jump for most things.
* Some conflict resolution: what if there’s already another spider where you want to go? What if you try to move off the map?

So I am going to make this and hope I find an answer to my question “what’s the objective and direction for this game?” in the meantime.

### Conflict resolution

Before jumping, I shoot a raycast that *only* finds other entities. (Not parts of the web or anything else.)

If something is already there, the jump is not allowed. (Alternatively, I might *push away* the thing that’s there. Depends on how powerful I want the jump to be.)

### Placing stuff

A few years ago, I figured out a nice system for placing stuff “roughly on top of a network”.

It, again, uses some nice properties of points and lines:

* Pick a random edge.
* Pick a point along the edge. (Position = Start + vector \* <some number between 0 and 1>)
* If we want the thing to be on the spider web, we’re done!
* If not, get the *orthogonal vector*. (Rotate the edge vector 90 degrees left or right.)
* And offset the position using that vector. (Position += Orthovector \* <some number between 0 and 1> \* MAX\_OFFSET)

This ensures the thing is placed *near* existing edges, but not exactly *on* them.

(We could go further than this. We could calculate the *polygons* the web creates and then place objects near the *center* of those polygons. I’ve done it before for other projects, but I think it’s too complicated to add now. Maybe at a later stage.)

TO DO: Image here =>

### Levels

At this point, I realized this “shoot line in direction” algorithm was *so powerful* I could actually use it for random level layouts!

The idea is simple:

* Start with a random point and shoot a line in a random direction.
* Pick a random point on a random edge, shoot a line in a random direction.
* Repeat the step above until satisfied. (A certain number of edges or points is reached.)

Of course, this can become quite chaotic. One part of the field might be filled with 20 lines, while others are completely empty.

To soften the issue, I’d do things like:

* Snap random directions to 8 or 16 fixed angles.
* Don’t pick random points *really close* to the extremes of an edge – pick them more near the center.
* Keep a count of the *number of points per quadrant*. If a quadrant becomes way fuller than others, simply don’t pick new points there.

TO DO: Image here =>

We’ll see how well this works. I might still add manual levels, if I feel they are better. But it’s okay for now.

### Owners & Silk Types

Reading this chapter hopefully shows how good it is to just *start making stuff* for a game. Implementing these things (which I needed anyway) gave me another idea:

**What if lines had an *owner*?**

Lines created by *you* get *your* color/icon. This means only you can travel over them. (Or your team mates, if this game will support teaming up.)

Why do I think this is interesting?

* If everyone can travel everywhere, there’s not really any strategy. The person closest to an object will get there first, end of story.
* If you’ve just paid 6 silk to jump all the way to the other end of the map … it’s a bit annoying if all other players can just use that line for free. It doesn’t feel fair. It promotes “waiting until others do the work”, which isn’t interesting behavior in a game.
* It just made sense to me. Your silk has your color, and it is yours.

Of course, I’ll need to balance this idea. Maybe:

* You *can* travel over other colors, but it costs 1 silk each time.
* The owner wears off after a while.
* Lines only become yours if *the jump is big enough*
* Lines only become yours *if a certain powerup is active*.

We’ll see. For now, I’m just coding the edges to support all this. (In a clean, modular way.)

## Step 5: Simplifying

After the “come up with as many ideas as possible”-stage comes the “simplify and streamline into an actual game”.

Writing down all the ideas, I realized there were *three different systems* …

* Items
* Entities (either predators that will eat you, other players, or bugs to eat)
* Silk types

… all of which did similar things.

I mean, I could add a silk type (“sticky”) that slows down your movement. But I could also add an item/powerup that does the same thing. What do I choose? Both? Neither? Is this much overlap a good thing or a bad thing?

My experience tells me this is too much and it should be simplified. With that mindset, I got some interesting ideas.

### Idea #1: Entities = items

**Forget items. The entities *are* the items.**

I mean, where do spiders get their silk (in real life)? Certainly not from powerups that suddenly pop up on their spider web :p No, they eat bugs, and their body converts it into silk.

Why not use this in the game as well? Bugs walk around over the web. If you bump into them, you eat them and get their power/resource/whatever.

Some bugs give points towards the objective. Others simply help you with good powerups. Others are predators that will give you negative stuff. *But all of them are entities*.

This simplifies immensely …

### Idea #2: Entities = leave trails

… but we can do even better.

Where do *silk types* come from? How do they appear?

* If I make them appear randomly, there’s no way for players to predict it or use it.
* If I only create silk types at the *start* (when I generate the initial spider web), they will soon be gone as the web expands.
* If I ask *players* to paint the silk, I’d need another button or system, which is the opposite of simplifying.

Here’s the answer I found most satisfying:

**What if *entities* left trails when they walk? When they exit an edge, it’s converted to the “silk type” that belongs to them.**

This way, the web will constantly shift as new types are added (or removed). Additionally, players can *predict* *how* the web will change. They see a specific bug walking somewhere and know: within a few seconds, the terrain will change there.

Look at that. Two simple ideas, two simple rules, and we’ve gone from three separate systems to *one system to rule them all*.

* Entities appear all over the field.
* They paint the web as they go.
* Eat them to get their powerup.

### An important remark

I “wasted” an evening on this step.

I couldn’t force myself to continue programming or start drawing icons for the items, because I had these questions that I wanted answered. I knew there were some gaping holes in the gameplay and that blindly executing the ideas wasn’t great.

(Additionally, I’ve created several games the past few months which had this exact same structure: terrain types, powerups, movement + jumping/throwing mechanic. I was kinda done with programming the same game over and over.)

So I watched a soccer match, then some YouTube videos, then did some research, then exercised – all the while asking myself these questions and writing down ideas from time to time.

To an outsider, it would look like I did absolutely nothing of value for 5 hours.

For me, these hours saved this project and made it *so much better* than the original idea.

The lesson here is that there’s great value in just relaxing, taking a break, thinking a bit on a problem before continuing. It might feel like wasting time. Others might look at you and think you wasted your day. But you haven’t. Creative work needs time, to solve the unavoidable issues creatively.

With that in mind, the next step will be obvious.

## Step 6: Creating entities

These “bugs” will be the lifeblood of the game. They should feel alive, like an intelligent creature, not just a powerup that happens to move up and down.

Here’s the plan. Each entity has a

* **Movement type**
* **Point value**
* **Silk type**
* **Specialty**

When you eat the bug, you gain its **point value**. It will probably be low numbers between 0-5 or 0-10. (Large or hostile bugs have large point values. But you need to do something special to get them.)

The **movement** determines how it moves. Some bugs might walk over the spider web. Others might fly freely, anywhere they like. Things like that.

The **silk type** is the trail it leaves behind. For many bugs, especially flying ones of course, this will be empty.

The **specialty** is, well, what makes this bug unique. The reason it’s in the game.

Whenever possible, the *silk type* and *specialty* will be something in the same vein. A bug that leaves the “move faster” terrain, will also make your spider faster when eaten. (It just makes sense. Helps remember what it does.)

### Movement

Because we already have the code for moving across a web, the basic idea is easy:

* Start the bug somewhere.
* Pick a random direction (forward or backward)
* When we reach a point,
  + Color the edge (we just exited) to our trail type.
  + Pick a completely random direction. (Remember: the algorithm will automatically snap it to the edge that resembles your input most.)
* Keep moving until dead.

We can do more interesting stuff, though. Like:

* A creature that never backtracks. (It never enters an edge it’s already been.)
* Or one that *only* backtracks.
* Hostile creatures might prefer edges with other players on them, or leading to points closer to another player.
* Friendly creatures might do the opposite and flee.

“Off-web” movement for flying bugs is even simpler:

* Start a timer
* Whenever the timer runs out, pick a new random direction and restart the timer.
* Always fly in your current direction.

By making the timer more random, this already feels quite natural. If we only *rotate* our current direction slightly, instead of picking a completely new one, bugs will fly a bit more smoothly.

### Fleeing & Chasing

These behaviors use many of the same principles we already used before (with vectors and stuff).

To flee from danger, we simply:

* Cast a Ray straight ahead. If it hits something that can eat us, move in the *opposite* direction of the ray.
* When we reach a point and must pick a new edge, only pick edges that have *no threat* on them.

To chase it, we do the reverse.

* When our Ray hits something, actually move *towards* it.
* When we must pick a new edge, purposely pick those with food on them.

This is how it works for web-based creatures. For flying creatures, we can keep the RayCast … but they don’t walk over points and edges, so what now?

These creatures get an extra collision circle (that’s about 3 times their size). When it detects something inside, it moves in the opposite direction (if fleeing) or towards it (if chasing).

Implementing this, however, raised some questions …

### About eating (and getting eaten)

At first, I *split* the Player and Entity scenes. My thoughts were:

* Players need *way more modules* than Entities, as they can receive input, jump across the web, keep track of points, etcetera.
* So it would be a waste of performance (and time) to support this for *all* Entities.

However, I soon realized that Players and Entities had *too much in common* to be separate objects. I was copy-pasting modules the whole time, until I had enough.

I reworked the code (and structure) so that *everything* in the game is an Entity. The Players simply receive a few extra modules to poll input and keep track of points.

When adding that “points” module, however, I ran into a silly issue. I already had a module named “points” that knew *how much points a bug was worth* (when eaten). So I had a name clash for two points scripts and things went bad.

Instead of simply renaming the module to “score” or something, I though this was another change to *simplify* and *streamline* the game, because I had two important questions:

* What determines if an entity can eat you (or you can eat them)?
* Where on earth do I show *two* values for all players: silk and points?

And the answer, as always, was to solve both in one go: we **don’t** show two values for players, by making your points the only thing that matters.

* Each entity has only one “points” number.
* By eating another entity, their value is *added* to your total.
* When jumping, you *pay* from your score.
* And now the most important one: **any entity can eat another *if they have more points*.**

So yes, players can be eaten by small bugs if their points are low. Players can eat other players, if they have enough points for that. Heck, computer-controlled entities can eat other entities on their path and grow based on that.

(A great way to show your point value, beyond showing the actual number, is to **grow** entities based on the value. So if you have 0 points, you’re really small. If you have 10 points, you’re this humongous spider.)

Doing this solved some of the last gameplay questions I had, created some great new possibilities, and allowed me to simplify my Entity scene.

(As there’s almost no difference between a Player and an Entity. Just two modules: “Input” is added, “AI” is removed. Yes, even *jumping* across the web is now something that some entities could do.)

### Trails

We already know when we leave an edge (and enter a point), so I can simply use that signal to paint the previous terrain. At least, that’s how it works for *web-moving bugs*.

For flying bugs, I eventually decided to let them paint as well in the following way: when they *cross an edge*, they paint it.

However, this was too powerful, as flying creatures can cross many edges in a single second. To combat this, a *timer* was added to the painter. Any time you paint, that functionality is disabled for the next ~3 seconds.

(This also applies to bugs moving on the web, as this behavior is desirable in general. You don’t want a bug flip-flopping between edges (perhaps because it’s fleeing) and painting five edges within half a second.)

TO DO: Image

I did have a problem in terms of *space* (or *legibility*). When I created icons for the silk types and placed them on the edges … they were just too small to read. I increased the edge thickness, but it’s not a full solution. I might look into …

* *Repeating* the icon across the whole edge
* *Flashing it* (bigger and brighter) once in a while, or only when it first appears
* Heavily simplifying the icons (to the point they’re a bit abstract) and explaining the general idea to players at the start.

### Specialties

It will take some time (and experimentation) to find the most fun properties I can give to bugs.

Some early testing, though, revealed the following:

* **Speed** matters a lot: both for catching bugs and outrunning enemies. If you can’t speed up or slow down, your fate has been sealed ten seconds before it happens. You know you can’t escape that predator following you. As such, there should be plenty of (strategic) ways to change this on the fly.
* In the same way, a **shield** and **non-hostile** bugs are nice. It would allow you to play more defensively, if you desire, and give yourself safety in the stress of the game.
  + (Non-hostile just means they will never eat anything. A shield will protect you against any attacks. There might even be a “*reverse Uno”-card* of sorts, where anyone trying to eat you is killed.)
* **Varied movement** and **web (creation) patterns** are great. A bug that flies *completely randomly* isn’t fun, because everything that happens is just random. A bug that only flies horizontally, or only clockwise, or pauses every once in a while is more interesting and strategical.

So, at this point, I just threw everything I had against the wall and then tested it. (Usually, that means the *code* and *functionality* comes first, and then I draw the actual sprites/colors/visual effects to go with it.)

After some frustrating hours rewriting my web code to be much cleaner, I got it all to work :p

(To give you an idea: players will create a new line in the web when they jump, and are limited to a maximum distance. Entities, on the other hand, do not create anything new and don’t have (the same) max distances. A jump is just a jump to them. This *also* means that some special powers, such as destroying other lines, cannot be used by computer entities.

The original “jump” module was only made for the players. It took quite a while to restructure it to support all the exceptions for other entities, *without* the code becoming a huge slow mess.)

### An issue with bugs

Bugs all have the same color scheme. That’s the issue :p

And yes, I know why. They’ve evolved to blend in with their environment (leading to all sorts of black, grey, beige, dark green colors) or scare off predators (with bright flashing red colors).

But in a game like this, players need to be able to *immediately* see what type of bug they’re dealing with. And for that, we need unique silhouettes and colors.

(And bugs are not that distinctive in the silhouette department as well, unfortunately.)

For each bug type, I searched for reference images that had at least *some* splash of color. Then I exaggerated this and just picked whatever colors were left for the other bugs. (Which led to blue-tinted beetles and purple-tinted crickets and that kind of stuff.)

*Remark:* players will of course have their own, bright, player colors that have nothing to do with the rest of the bugs.

TO DO: Image

## Step 7: Turning it into an actual game

At this point, I’d been working on the game for roughly a week. This meant that:

* All core functionality was working (mostly): creating a web, moving over it, eating other entities, etc. (There are still some nasty bugs and exceptions to handle, as always.)
* About 30 different bugs had been implemented. All of them did *something* unique to the web and the gameplay.
* I’ve been testing, experimenting, thinking about good objectives all that time.

The reason I implemented all those different bugs, without even knowing for certain what the game rules would be (and if it was even fun), is because I’ve learned you just can’t *predict* what will work.

If you just *build* it, you can test it and immediately see if it works.

### Observation 1: Special powers

That’s where the first observation comes in: **most of the special powers I invented are fine, but some are a bit “meh”** (they either don’t do much or are too cliché). The coming days, I want to come up with more *unique* bug types that would be more interesting.

For example: there are now a handful of bugs that destroy edges in different ways. One edge is destroyed as soon as *one* person leaves. Another is a timebomb that is destroyed after some time. Another is destroyed by cutting it using your “jump” input. All of these are *fine*, but I’m not sure if I need that many ways to do similar things. Perhaps more *creative* stuff can be done. Something unique to this idea of “move over a web” and “build the web while playing”

### Observation 2: Power Creep

The second observation is: **power creep.** As soon as you’ve eaten a few bugs and are at a good size (10+ points) … nothing really scares you anymore. Any extra points from there don’t change your game or playstyle. It’s just a boring “eat even *more* creatures until you win”.

But, ending the game as soon as someone gets 10 points isn’t viable either. It happens way too quickly. Players might get lucky at the start and reach 10 points in no time.

I’ve decided to solve it this way:

* Each team has a *home base*.
* This base shows your *total points*. Whenever you visit your home base, all your points are drained and added to the total.
* You can’t hold more than 9 points at a time.
* You need to reach quite a high *total number of points* to win.

Why do I think this will work?

* The low ceiling on points means you simply cannot grow too big and must regularly drain the points.
* Every time you drain, you become vulnerable again and the tension rises.
* Games can last a good amount of time (~5 minutes), even if someone becomes very (un)fortunate.
* Players don’t need to remember how many points they need to score: I can just display it on the home base (“23/50” collected)

But most importantly, the home base presents a huge strategic value. Namely, players will have to constantly ask themselves these questions:

* “Do I risk getting more points, or play it safe and go back to base now?”
* “Do I want to keep my high point total in case something good comes along, or am I content to drain it and start from scratch?”

I was planning to add this “home base” as a special mode. But testing the game … it just doesn’t work without it, and it’s too valuable *not* to include by default.

### Observation 3: How to handle death?

At first, I wanted to make death optional. A rule you could turn off. And when turned off, I would build simple ways to prevent deaths.

For example: if someone destroys the edge you’re walking on, you die, as you fall off the web. I could go around this by simply teleporting any entities to one of the endpoints of the line.

But, when it comes to actual gameplay, this has huge issues.

* What if those endpoints are also removed? How do I write a quick, safe algorithm for teleporting you to *something*?
* Even if I manage to do so, that *something* might be very far away (if a large edge is destroyed). That’s really disorienting and can lead to very unfair situations.
* And what if all entities are teleported to the same point? The big ones will immediately eat all the small ones, until one is left!

Needless to say, I chose to abandon this idea. *Dying is dying*.

If your edge is destroyed, you die. If you are eaten, you die. If some special power kills you, you are dead.

It makes the rules and gameplay *much more consistent* (and easy to program and balance).

The fact that you’re dead, however, doesn’t mean you need to *stay dead* :p

There’s nothing worse than playing a (local multiplayer) party game, dying in some stupid way during the first 30 seconds … and having to sit through 5 minutes of others having fun without you.

No, everybody should stay in the game until someone wins. **When you die, you simply respawn with a handicap.**

Here’s the idea:

* It takes a few seconds before you respawn. (This “delay” is both a breather so you can process what just happened and a penalty for dying.)
* You lose all your points.
* You go back to your home base.
* **The target objective is lowered**.

Let’s say you need to score 50 points to win. Every time someone dies, this numbers decreases. 49 points to win. 48 points to win. In other words, the more you die, the *easier* it becomes for others to win.

**Remark:** This also prevents the game from reaching a stalemate, where everyone just keeps dying without making progress. Even if all players are *that* incapable, the target will lower and lower until *someone wins*.

**Remark:** in case of a draw (multiple teams have the target objective once lowered), we use some other tiebreaker. Like: number of deaths, number of edges created, etc.

**Remark (added later):** Oh, something I forgot to mention is that the target objective is obviously lowered for *everyone but you*. If your own target is also lowered, endlessly committing suicide might even become a good strategy if you’re in the lead :p And we don’t want that.

### Observation 4: The scale of things

Until now, I’d been testing the game on a completely black background, as it allowed me to see the (white) silk of the web. Additionally, I just input a line thickness that seemed right at the start.

Now that I know the complexity of the game, the icons used, etcetera, I realized:

* Edges (and points) need an *outline* to really make them pop out against any background.
* We don’t want black backgrounds – makes the game look like something it isn’t (a dark, serious game about being an insect)
* Edges must be way *thicker*. This also means that points and jumps need a larger minimum distance (between them).

The downside is that we lose some space on the field, allowing for *fewer* points and lines as the game progresses, and thus fewer variation.

But the upside is that everything is way easier to see at a glance and the game pops out more (visually) – which are more important to me.