# 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*.