# Devlog: Rolling in the sheepe

Welcome to my **devlog** for the game “Sheepe”, otherwise known as “Rolling in the Sheepe”.

The idea was simple: everyone is a sheep in a **random shape**, you can only **roll yourself**, and the first to reach the finish wins.

It’s another one in my series of **One Week Games**, hence the extremely simple idea and limited scope.

So let’s get started!

## Task 1: Random shapes

I use **Godot Engine**, which makes this *very* easy.

* I place points in a circle
* Then I randomly move them inward/outward a bit
* Then I tell it to draw this list of points as a polygon.

<TO DO: Code example here> / Image

That’s it. Done.

(I’ll talk about some issues with this and modifications later, but for now this is fine.)

## Task 2: Turn shape into body

Again, Godot to the rescue:

* Create a “CollisionPolygon2D” node
* Hand the list of points we just created as its polygon.
* Make it a child of a “RigidBody2D” node.

Done.

## Task 3: Rolling Shapes

As we’ll be using physics, we don’t care about “rotating” the shape perse, we care about *adding angular forces* that cause it to rotate.

Hopefully, the friction with walls/floors will allow it to move forward and actually make this game possible.

* When the right key is down, add a POSITIVE angular force to the body
* When the left key is down, add a NEGATIVE angular force to the body

Going strong!

## Task 4: Splitting shapes

Now, this is where it gets interesting.

One of the “main features” of this game should be **shape splitting.** When you roll into a spike, it should actually **slice your** **body in two.**

### First try: easy and convex

I drew some quick diagrams in my notebook until I saw a pattern. This pattern was really easy to implement and created perfect slices … *for convex shapes*.

What’s a convex shape? It’s a shape without holes in it. A circle is convex. A rectangle as well.

(Mathematically precise: you can take *any two points inside the shape*, and the *line between those points will be fully inside the shape as well*.)

Even though players might only *start* as a simple convex shape, over the course of the game this *might not be the case anymore*. So I needed something that worked for *concave shapes* as well. (Which are simply all shapes that are not convex.)

<TO DO: Show diagram>

But first, let’s take a look at my first algorithm:

* Loop through the edges of the shape
* Intersect each edge with the “slicing line”
  + No intersection? Continue
  + Intersection? Save the current *index* (in the shape array) and the exact *point* at which they intersected (just coordinates).
  + We’ve found two intersections? Great, we’re done.
* Now *extract* everything between the first and second *index* and save it as a new shape: shape2.
* Whatever is left of the original array is shape1.
* Destroy the old body, create new bodies for the new shapes.

As I said, this works flawlessly. As long as you don’t forget to:

* Transform the shape to *global* coordinates before you start. (Taking into account the rotation and position of its body.)
* Transforming the shape to *local* coordinates when done. (Calculate the average position of the points, also called the *centroid*, and reposition around that.)

Here is some (simplified) code:

<TO DO: Show code>

It executes extremely quickly, doesn’t take that much code, and works for all *convex shapes*. (And if you slice a convex shape … it will stay convex, so no issues there.)

### Second try: breaking it down

But when I tried it with a concave shape, my shapes somehow *tripled?* I was astounded at first, as I was certain the algorithm only ran once, so it could only create two bodies (at most).

But then I turned on the “debug physics shapes” option. And I saw what Godot was doing: it automatically **triangulates** concave shapes.

In other words, if I give it a concave shape, it breaks it down into separate **triangles**. Then it saves each triangle as a unique shape of the body, so I can access them separately in the code.

(Why? Because triangles are *convex* and easy to work with. I’m not surprised this happens, I’m surprised Godot does it without telling you and then *lets me access it*.)

So this is great! It’s just what we need actually!

We can make this work if we:

* Create a list of shapes that contains *each triangle individually*.
* Create a new empty list.
* Run the slicing algorithm for each shape in that list
  + Any new shapes created, are added to the new list
  + If untouched, the original shape is simply copied to the new list.
* Then we loop through the new list and *stitch together* any triangles that should be together.

The first three parts are easy. (Just modify the algorithm we already have.)

The last part is not. How on earth do I *merge triangles*? And how do I only *merge the correct ones, not those that were just sliced?*

### Merging triangles

Let’s think about this.

* Insight #1: They are *triangles*. Two triangles will share at most two points. If they share only a single point, I consider them “separate” and they shouldn’t be merged.
* Insight #2: The points are *ordered* (clockwise in my case). If we find one point that matches, we only need to check the next point to see if we have a matching edge.

So, for each triangle we loop through its points, and check if any other triangle has a point in it *at the same coordinates*. That is a “matching point”. Then we check if the point after that *also* matches with that triangle. If so, we have a “matching edge”.

If we’ve found a matching edge, we add the *non-matching* point from triangle2 to triangle1 in between the matching ones. Then we delete triangle2; it’s been successfully merged with triangle1.

Reconsider triangle1 until it doesn’t match anything. Repeat until all shapes have been considered

### ignoring the right ones

Well, what do we know about the triangles that should *not* merge?

They have matching points which *lie along the slicing vector*. Those points were just created, in the slicing algorithm.

In other words, if we find a matching point, we first check if it lies on the slicing vector. If so, ignore it and continue.

There’s a fairly standard algorithm for checking if a point is on a line segment:

URL: <https://stackoverflow.com/questions/328107/how-can-you-determine-a-point-is-between-two-other-points-on-a-line-segment>

<TO DO: Code here>

### The issue here

So I wrote this algorithm. And … I ran into issues.

Do you spot the issue here? It’s rather obvious, in hindsight, especially now that we have the code and some drawings.

*After merging two triangles … we obviously don’t have a triangle anymore*. So the first merge might be fine, but then it all goes haywire. I tried some hacks around this, but in the end I just had to admit I learned my lesson and “merging convex polygons” is a *terrible idea* which you shouldn’t even try to do.

No, merging isn’t the solution here.

Instead, I think I should *keep* the separate shapes that I have. Once I’ve sliced some of them, it becomes a matter of **reassigning them properly**.

(All shapes that have matching points, should stay together in one object.)

### third try: complex and concave

And that works!

To summarize, this is the algorithm:

* Detect which objects are underneath our slicing line
* For each object …
  + Get all its unique shapes
  + Slice each of them. (If it doesn’t hit the line, it just returns the original shape. Otherwise the two new shapes.)
  + Once we have the list of *new* shapes, put those that share matching points in the same “layer”
  + For each unique layer, create a new object, and assign all the new shapes.

The slicing algorithm is identical to before. (Because, remember, the unique shapes that make the object *are* guaranteed to be convex.)

The only new (and perhaps difficult) part is “assigning shapes that should be together to the same layer”

For this, I used the following algorithm:

* Initialize the list of layers (for each shape) to -1 (or null, or whatever)
* For each shape
  + No layer yet? Create a new one and put the shape in there
  + Check all other shapes.
  + Do we have a matching point?
    - Copy our layer to the other shape.
    - Or, if the other shape already had a *layer* and its lower than ours, take over *their* layer.
    - Now start the loop from the beginning, because our layer has changed.
  + When checking matching points, *ignore any points that lie along the slicing vector*.

<TO DO: Code here>

### About floating point precision

That last part is actually where I got stuck for a bit. The algorithm would work … erratically. Sometimes it was perfect, sometimes it didn’t do anything. I couldn’t spot any errors or logical reasons why.

In those cases, you simply try a lot of different simple situations, and check the outcome. Hopefully, this highlights a pattern, or you can isolate the part where it goes wrong.

In this case, that never happened. Even the *simplest* of situations would fail … sometimes.

But I have experience with those kinds of situations! And a voice in the back of my mind said: *floating point precision error*.

Computers cannot save *all* numbers with infinite precision. There are a limited number of “bytes” reserved for each number, and any precision that needs more bytes is lost.

This means that the exact same *point* (of shape) could actually have a *slightly different coordinate*. Checking “point1 == point2” would *fail*, because they’re not *identical*.

Checking whether **a point lies on a line segment** is impossible this way! Because a line is (mathematically) defined as having “zero width”, so the point only needs to be *slightly off*, and the check fails.

That’s where that variable **epsilon** comes in. It designates a “margin of error” we will allow and which will still be counted as “these coordinates are the same”.

The issue? My epsilon was too low. I set it to something like “0.005” (which is quite standard). But upon further inspection, the algorithm works with quite big numbers, so I bumped epsilon up to a way higher value.

That fixed the whole issue. Simply setting **epsilon = 0.1** (or even higher maybe) fixed everything and was the only reason I got stuck for an hour or two.

There you have it. If something behaves erratically, and you’re working with *floating point numbers*, it’s probably something like this. And never, ever, do a “==” check between two floats :p

## Step 5: Nicer slicing

So we have a slicing algorithm, which will *very precisely* cut any shape we give it.

If we’re unfortunate, this might cause *very tiny shapes* (which are barely visible). That’s ugly and unplayable.

Therefore, we need to check if a shape is *too small* (by calculating its area), and do something about that.

I see two different approaches:

* If too small, *don’t allow the slice*. (Just pretend it didn’t go through the object.)
* If too small, *destroy the second body* (that was sliced off).

I eventually chose the second option, because it simplified the system *and* allowed future gameplay possibilities:

* Getting sliced is always bad and works in predictable ways, which means clarity and consistency.
* The *smaller* your shape, the *slower* you move
* You need a *minimum size* to finish. (Any time you get bitten by the wolf chasing the sheep, you lose something. But during the game, you can also find new pieces and grow yourself again.)

Our last problem becomes: **how do we approximate the area of a polygon?**

There’s no need to be precise. Most of these polygons will be *triangles* or something close to it. What to do? We’ll just pretend they are a triangle and use the formula for calculating such an area: 0.5 \* width \* height

Then I just played with it, printed the areas of things I sliced off, until I had an idea of what a good “threshold” was.

(In my case, it was higher than I expected. Because we’re calculating an *area*, even a tiny 10px by 10px square … has area 100. I settled on a number around 400-500.)

## Step 5: Following the players

We need a camera that always keeps *all parts* of *all players* in view. Preferably it should:

* Stay zoomed in, so things don’t get too small/far away
* Also show what’s “up ahead”
* Not be janky or stuttery

From earlier (local multiplayer) games, I’ve learnt some hard lessons about camera management. Namely, that you **shouldn’t try to create a camera that keeps all players in view**, but instead should **create a *game* that ensures all players stay together**.

You have to think the other way around. Because no matter how hard you try, if you allow players to get *far away from each other*, you’ll never find a camera setting that stays close and zoomed in.

And so I settled on the solution of **locked-in sections.**

* The map consists of multiple “sections” placed after each other.
* Each section *ends* with some sort of lock. This can be a physical obstacle, a minigame you need to complete, anything that stops you (for a while).
* This lock ensures that, 99% of the time, the first players are slowed down and the last players can catch up. If that doesn’t happen, *any player that’s more than 1 section behind is simply teleported forward*.

In my opinion, this is the best solution.

* Players are never “out of the game”. (Either by being eliminated *or* by being so far behind they can’t practically win anymore.)
* Players doing well (which are in front) are not *punished* for it. Instead, they simply need to overcome *extra* challenges to maintain their lead, while allowing other players to catch up a little.
* Breaking the map into sections creates a nice, visible sense of progress. You’re never lost. You’re never unsure about why you were teleported forward. The sections give clear indication.

In conclusion:

* The camera is simply placed on the *average* position of all players, but *slightly* forward (to show what’s coming.)
* By calculating the *maximum* distance between players, I know how far we need to zoom out to keep everyone in view.
* That’s it. The camera itself has no other logic, it’s up to the *game* to keep all players nicely together.

## Step 6: Creating the map

At first, I wanted to make a game that only goes to the right. (Which is typical with these kinds of “runner” or “platformer” games.)

This, however, presents several issues:

* If you have good speed, you’ll just *fly* forward and nothing can really stop you. (Because you only need to go right, is there any reason to slow down or roll to the left?)
* It makes it *much harder* to keep all players in view. (We’d have loads of unused space *vertically*, whilst players are far apart *horizontally*.)
* After a while, if we go to the right long enough, we run into those same “floating point precision” errors, because our coordinates are just too large.

That’s why I decided to make a map that is more like a *maze* and can go in any direction.

This is the idea:

* Predefine a “world size”. (For example: max 100 tiles wide and 100 tiles high.)
* Start at the top left corner.
* Create a random route through this world, ensuring that …
  + All of it is reachable
  + It is long enough to warrant a full level
  + It’s broken into these *sections*
* Place a finish at the end

My first instinct is always to reach for some *perfect* algorithm to generate a *maze* or something. That’s just how programmers work :p But I’ve learnt over the years that trying a naïve/dumb/simple solution first is usually *all you need*.

(Additionally, we don’t need or want a *maze* for this game. It is *side view*. Gravity is always pulling us down. We can only roll (with random shapes). Preferably, the route will mostly flow downwards and switch between left<=>right once in a while.)

All we need is:

* A route that regularly changes direction
* And that keeps players together, so we don’t need to zoom out a lot

Well then, *let’s only fulfill those wishes* and *nothing more.*

There’s *no need* to generate a full map beforehand. There’s *no need* for the route to make sense. (We can just re-use a location we’ve already been later on, with a completely new room.)

This is the idea:

* Check where the first (“leading”) player is
  + When they move into a new room, we immediately instantiate a *new room* at the very end. (This way, we build the map as we go, ensuring players can always move forward.)
* Also check where the last (“trailing”) player is
  + When they move out of a room, there’s no use for it anymore, so remove it.
* When picking a new direction for the next room …
  + The longer we’ve been going straight, the higher the probability of changing direction
  + Prefer a direction that keeps all players in view.
  + After placing new rooms X times, we *end* the current section (with such a “locking mechanism”).

**Remark:** and when we simply cannot place a new room? We stop there. We place a *teleport* or something. It waits until everyone has arrived, and then we simply *zoom* to a new level/part completely.

### Does this work?

Yes. It works great!

It’s not that hard to program, whilst allowing the game to basically be as varied (and *endless*) as possible.

(Additionally, it’s great for performance, as there will only be ~10 chunks in the game at a given time. But that really won’t matter much, unless I decide to port this game to mobile.)

<TO DO: Code example>

There’s only **one issue left: how do we select/place rooms?** How do we ensure a room fits onto the previous one, and there’s always a path forward?

I’ve done this in the past with this approach:

* Each room has several openings
* I save these locations in the room. (For example: openings left = index 2 and 4.)
* When selecting a new room, we simply pick one that *has the right opening*.

This works well. But those previous projects were different from this one, given that:

* There was no need to create a single route. As such, rooms often had *multiple openings* going in multiple directions.
* Rooms had to keep their orientation. (I couldn’t *rotate* them, for example, to match them.)

So, we need to modify the approach for this game:

* When matching edges, we **are** allowed to rotate the room. (There’s no reason not to.)
* Any edge we **do not** connect, is closed.
  + When a new room is placed …
  + We check all openings in the *previous* room, ignoring the ones we actually used.
  + For each one left, we place a solid block on that location to “fill” the gap.

### What are “rooms”, actually?

I notice I’ve been saying “room” all the time, without giving a clear example what that entails. That’s partly because *I* wasn’t sure yet.

Now I can explain this a bit better.

Each “room” is a block of grid tiles (probably 4x4) that has **one unique challenge or mechanic**.

* The most basic room is just empty.
* But an “obstacle room” might be filled with all sorts of bodies you’d need to navigate through.
* And when you’re in the “glue room”, you’re able to glue yourself back together (if two of your pieces touch each other).

By chaining these rooms together, you are constantly presented with new challenges to overcome, as you progress towards the finish.

Additionally, it gives me great *control* over what appears and how often. (You wouldn’t want a game that, by pure chance, consisted of 100% glue rooms and nothing else.)

## Step 7: Trying something completely different

So I implemented everything I talked about in the previous section. And I tested it. And I played it.

And it … just didn’t work. These are the reasons (I think):

* Gravity is always down. As such, most of the game you’re just *falling down through a bunch of rooms*.
* It created quite a static layout that wasn’t very pleasing.
* There’s no way to go upward, or jump, or anything. You can only *roll left and right*.

It’s much better if

* Players have some *solid (horizontal) ground*
* Moving down/up only happens sporadically. (And if it happens, you get support for it. Like a trampoline on parts going up.)
* Players can *roll onto walls*. (So when you roll into a wall, you *cling* to it, so you can follow it.)

So let’s turn it around.

* The map is one big *chunk of blocks* at the start.
* Instead of placing new rooms, we just *erase* part of the blocks. (Essentially creating something like a cave or tunnel through the map.)
  + (Conversely, instead of deleting rooms on the tail end, we just *fill it back up*.)
* Regularly, we change the width of that eraser. (Sometimes it removes blocks 2 wide, sometimes 1 wide, sometimes 3, etc.) We also randomly add an “offset” so new rooms don’t all start at the exact same level.
* When changing width, we add *slopes* to make it gradual.
* (The probability of moving vertically is much lower than moving horizontally.)

### Did that work??

Yes! It did! (In hindsight, it’s obvious this was the better choice. But hey, lessons learned.)

Now players can actually roll onto stuff. They stay contained within the grid, while having more than enough space to maneuver. The routes are varied and can already be quite challenging. (This is *before* I implemented nice slopes and clinging to walls.)

It’s surprisingly *rare* that the generation fails. (It has painted itself into a corner and can’t get out.) If it happens at all, it’s usually after 30-60 seconds of playing, which is already a good length.

So, how exactly *do* we add slopes between height differences?

* When erasing a new part, check all cells within that section *+ their border*. (So, increase the room size by 1, then check all cells within that rectangle.)
* For each cell that has:
  + Exactly two neighbors, which are *not* opposite each other, add a slope.
  + (If the neighbors are opposite each other, this is the perfect location to place a door or a laser or something. But that’s for a later moment.)

**Remark:** Rotating the slope correctly is a minor implementation detail (which depends on how Godot handles tilemaps and how I happened to draw the slope), so I won’t bore you with that. Also because – and this is me from the future – I ended up using a different system anyway (“autotiling”).

**Remark:** technically, we only need to check the *border* right now, not the inside of the room. However, as I plan to *use* the empty space inside for things, I already set up the loop to check those as well.

**Remark:** we *also* check for existing slopes that have become useless. Because we placed a new room, the environment changed, so older slopes might not be needed anymore.

### In conclusion

Right now, we have randomly generated maps which are fun, playable, and even finishable (with our limited toolset).

It still crashes whenever it can’t find a solution, but we’ll solve that soon.

For now, I want to add several more (interesting) ways for movement …

## Step 8: Better movement

Right now, you can *roll left* and *roll right*. Because we’ve added slopes, this already gets you quite far.

But you still can’t go up. And you still get stuck on high vertical jumps/obstacles.

I see *two* interesting things to add:

* **Clinging to walls =>** whenever you roll against a wall (with enough force/close enough), you stick to it
* **Jumping** => whenever you **release both buttons simultaneously**, you jump.

Jumping is simple to implement: apply a force *away* from the ground.

Clinging is, interestingly, kind of the opposite. Use a *raycast* to detect whether we hit something next to us and the *normal* of that collision. Then apply a (strong) force *in the direction of that normal*, pushing us into the object.

Because of the default “friction”, this causes us to stick against the surface and roll along it.

At first, I created a bit of a “rough” implementation of both features. Jumping was endless (you didn’t need to be touching the ground). Clinging only happened horizontally (if a wall was to the left/right of you).

But … experimenting with this led to some amazing insights! With this system, you could:

* Cling to a wall and then *stand still*
* Press *jump* to release yourself again.

Not only was I able to finish *any* route this way (with some trial and error), it just felt *cool*. It felt cool to roll up to something, then stand still in mid-air, waiting for the perfect moment, then launch myself again.

If I can perfect *those* behaviors, this game will certainly be fun to play.

Additionally, there’s not much more to do here. We only have two buttons. They each do something separately, they do something combined, *all my inputs are taken.* So, any more variety/mechanics shall have to come from the levels themselves and the elements within them.

## Step 9: Making the first finishable level

What do we need to make a first finished, playable prototype?

* A finish => reaching it first, with *all* your pieces, wins you the game.
* A “section lock” system – breaking the route into pieces, keeping it all manageable and on-screen.

That’s it! So let’s make that and then *test the game*.

Creating the *section lock* and a *teleporter* (in case you were stuck) were relatively easy. I simply:

* Place a large room
* Wall it off with edges, except for the direction you came from.
* Add a “module” (just a script and some objects) with a certain minigame
* Once completed, the edges are removed and you can continue

For teleporters, the same is true, but once all players have arrived *the route restarts completely somewhere else* and *players are teleported there.*

However … the question of “how exactly do you win?” haunted me during that period.

At first, saying something like “reach the finish with *all* your pieces” seems great.

But there are problems. This encourages players to *lose everything* (and destroy themselves), because it’s much easier to get through obstacles if you have only *one tiny piece*.

At the same time, if you are split between a few pieces, it might be really hard to bring them all to the finish. One of them might be stuck somewhere, and there might be no way to get it out. And because it’s stuck somewhere in the back, the camera needs to zoom *way out*, and the route generation gets stuck (because there’s no space).

It’s just not great.

I want to change it to the simple objective: “finish to win” Just one piece. Get one piece over the finish (first) and you win.

Of course, this still has the first issue: the best strategy is to always destroy yourself.

How do you solve that?

* Option 1: *penalize* this behavior
* Option 2: add *other* strategies that are just as viable (or even better)

Penalizing is annoying. It can easily lead to frustration and a “stale game”. So the second option will have to be used 90% of the time.

But it’s a party game. I don’t want to teach new players “these are the 4 ways to win”. No, the objective needs to be that simple one-liner: reach the finish first.

Instead, I will try to (invisibly) **nudge players in the right direction** by being smart about what elements I place.

For example, let’s say I want to encourage players to stay big. How do we do that?

* Idea: Being big makes you faster
* Idea: Add a “lock” or “gate” you can only pass if you’re large. (Otherwise it takes a while, or it’s harder.)
* Idea: you can *slice* other players by bumping into them. The bigger you are, the higher the probability of successfully slicing someone else.

I don’t need to *teach* these things or *explain* them specifically. Just place them in the game/in a level. Yet they give players a reason to vary their strategy and try new things.

That’s what I’ll implement now: you win if you reach the finish first. Any part of you.

**Remark:** of course, I could add powerups later that modify this. A “time penalty”, for example. Or a “curse” that requires you to finish all your pieces anyway.

## Step 10: Better Routes

Right now, there’s quite a large probability of getting stuck (and having to place a teleporter). Even if I can clearly see some free routes that could be taken.

I want to lower this probability of placing teleporters *as much as possible*. (It’s more fun if the route keeps flowing and players aren’t stopped artificially.)

To do so, I tried these techniques. If we can’t place anything …

* *Reduce the room size*. (Because a room of size 1x1 most likely *can* be placed, even when a 3x3 one cannot.)
* *Backtrack*. Try to attach a new room *earlier* in the route. Continue backtracking until you find something, *or* you’ve reached the room that holds the current leading player.
* *Loosen the restrictions*. Normally, I disallow overlapping *and* adjacent rooms. But if we seem stuck, I can start allowing adjacent rooms. And if we’re still stuck, I can allow some overlap.

This improves it somewhat. It’s not *amazing*, but it’s a good start and yields good routes for now.

By loosening the restrictions, and allowing rooms to overlap, I did create an extra problem: it’s not *clear* what the route is anymore. What way is forward? Sometimes you don’t know.

After some experimentation, I found the best solution was simply to allow this, but *add edges (“outlines”) around the rooms* (like I did with the locks) This way, it’s still clear what the rooms are and what path you should take.

In a general sense, I learned: using **edges** to separate stuff requires less space than using **full blocks** to do so. So maybe I’ll use this tactic way more.

Another thing I learned, just from playing/testing the mechanics so far, is that:

* Clinging to walls should *not* be automatic. It’s too strong for that, which makes some parts ridiculously easy, and others (near) impossible. It should only be activated on certain areas.
* The less “round” your shape, the less you’re able to roll. (Which is obvious, I know, but …) The difference is *so big* that it’s basically impossible to move well if you’re not *somewhat like a circle*. I should invent something to “help” the flatter shapes, I think.

## Step 11: Throwing sand against the wall

And now we’re at the point where I simply **implement a bunch of stuff I thought of**. Then I check what works best, and what doesn’t, and keep the best things.

(For example, I invented ~10 terrain types, a few ideas for powerups, and a few general game rules. I can’t *predict* what will be the most fun. Nor do I know the best *order* in which to teach them to players. So I just implement all of it (as quickly as possible) and then *test*.)

The results?

* Most of my ideas work great! They are a fun challenge, without just being impossible or impossible to understand.
* Nevertheless, I *do* need to finetune physics parameters. And I *do* need to restrict terrain placement to avoid some bad situations.
  + (For example, if the room goes *down* it’s extremely annoying if it has a *reverse gravity* terrain. Because it constantly pushes you up, it’s near impossible to get through this room.)
* By implementing these things, I’ve generated tons of new ideas. I also learnt that I should probably simplify the game. (Now there are: terrains, powerups, locking rooms, and obstacles/items. Perhaps the last three should all just be shoved under “special room”.)
* **And the route generation … needs a serious rewrite.**

In this period, I also implemented many other features I would need anyway:

* The camera cannot show anything out of bounds. (So no ugly spaces outside of the grid if we’re zoomed out.)
* There are “light circles” around players, the rest is dark.
* I implemented “autotiling” with prettier tiles. It means that my game engine automatically choose the correct sprite based on surrounding sprites, so it all connects well and looks good (and organic).
* I implement “catch up mechanics”. If you’re too far behind, you are teleported forward. If you’re not moving for 10+ seconds, same thing. All of this, obviously, has a cost in the form of a *time penalty*.

<TO DO: Image?>

## step 12: Better routes, For real now

The current route generation algorithm is “fine”. But now that the game has evolved, this isn’t enough anymore, and I have requirements it cannot meet.

(Additionally, the *actual route* is *the backbone of this game*. It’s the most important thing. It’s what makes or breaks the game. I think it’s more valuable to spend extra time here, than try to cover it up later with all sorts of powerups or other mechanics.)

To remind ourselves, here’s how the current algorithm works (in simplified form):

* Get the last room we placed.
* Try to place rooms of random size in random directions *next to this room*
* Once we find something, place it.
* If we haven’t found something after loads of tries, it’s clearly impossible, so place a teleporter and stop there.

This has the following problems:

* Trying this algorithm *thousands of times* is very resource intensive and introduces lag/stutter.
* Additionally, there’s no *guarantee* that it finds something sensible. If we’re out of luck, it only tries stupid configurations that would never work, and misses the obvious one.
* It’s terrible at using the space it has. (Often it has *loads* of space on the right … and only places a 1x1 room before going left.)
* It’s hard to control this with extra requirements, such as “no more than 2 rooms vertically after each other”.

This got me thinking. I’ve implemented a “grow rectangle” function, which is *fast* and already *used a lot*.

**Insight #1:** why don’t we start each room at size 1x1, and then simply *grow* it until it hits something? (Or reaches a maximum size of 6x6; we don’t want humongous rooms spanning the entire world.)

This would guarantee we use *as much space as possible*. Additionally, we don’t need to retry the same location at different sizes, as this *will already have happened*.

Which got me thinking again. When you look at a rectangle … there are only a limited number of spots to place an adjacent rectangle, aren’t there? Currently, we’re running this loop *hundreds* of times … but there aren’t even that many options. Not even close.

Let’s calculate an example. Our current room is a 2x2 rectangle. We are trying to place a 3x2 rectangle next to it. (Which are “medium size”.) Then we only have … 14 possibilities.

<TO DO: Image>

**Insight #2:** just generate a list of *all* possibilities at the start, filter those we do not want, and pick the best option.

It should be much faster, we shouldn’t miss (good) options, and it’s easier to manipulate with extra requirements. (Even on two huge rectangles, the number of possibilities is below 50.)

And that’s what I did.

**Remark:** I also took the time to clean up the *map* and *room* scripts. Both were just one giant script with 1000 lines of code at this point, doing *everything*. Now each of them has about 8 “modules” that do just *one* thing in ~50 lines of code.

**Remark:** interestingly, when I started this project, I didn’t know I would be using this concept of “placing rectangles”, and thought the route would just be a series of 2D points. Hence, the *main variable in the game* (“path”) was saving *coordinates*, not the *rooms themselves*. Over time, this led to hundreds of lines of code like:

“var my\_room = get\_room\_at(get\_location\_at(get\_my\_index()))”

Which is just horrible. Saving the *rooms* in that list instead, saved me tons of code. Yet I would never have made that “obvious” optimization if I didn’t completely rewrite and restructure this script.

**The results?** Again, works wonderfully, and it’s stupid (in hindsight) that I didn’t try this first. Using these tweaks, I was able to increase the *quality* and *performance* of the maps in the game:

* **Controlled variation** => Take an average over the last 5 rooms. The bigger those were, the smaller we must be. (This means the route alternates between tighter sections and huge open spaces.)
* **Preferred movement** => check directions in this order: continue horizontally, go in opposite horizontal direction, go down, go up. Usually, the first direction yields a result fast.
* **Stay away from edges** => if close to the edge, simply *do not consider that direction at all*. (Saves a lot of performance.)
* **Sneak peek** => The game would lag/stutter when there was *loads of open space in front of us*. Because it would try every rectangle and then grow all of them until maximum size, before making a decision. As such, before starting the algorithm, I try *one* room at maximum size (with some random size/displacement). If that fits, great, just use that.
  + This is a way bigger improvement than you’d think. In about 75% of the cases, it saves us all the computations of the algorithm (hundreds of rectangles to create and check), while placing the ideal room.
  + And because I control the “maximum size”, it works for big rooms and small rooms alike.

IMAGE: debug\_sheepe\_5

I disabled some things for the image here, such as the lighting effect. I feel like it shouldn’t be so dramatic. Or only used during “night time” sections, or something like that.

Another thing I disabled was my algorithm for adding slopes in 90 degree corners, because it stopped working after this update. Speaking of that …

## Step 13: Better slopes + “inner tiles”

Now that we have **big rooms** (consistently, in a controlled way) I can **fill the rooms** with things!

The easiest first step is to fill them with *more tiles*. (Which might split the room into two, or add a few islands, or something like that.) However, we don’t want these to block entry to the room.

As such, we need a function that:

* Given a position in a room
* Checks whether any of the *neighbor* tiles are part of a *different room*.
* If so, disallow that position.

Once we have this, we can **also** use it for adding back slopes! (Because, again, in a “rolling game” we want as many smooth transitions as possible.)

This works great, as you can see here:

IMAGE: debug\_sheepe\_6, debug\_sheepe\_7

Because anything *outside* the room is *also* “a different room than us”, none of the tiles are placed against the edge, ensuring we have a path through the room.

Of course, for variety’s sake, I *could* allow this as well. But this has to remain a “One Week Game” and I’m stopping here.

## Some tricks

I *did* end up having to implement an extra trick.

Because of the “autotiling” system (and adding slopes), it happened quite often that you could enter a cell (with a slope) … that didn’t belong to any room.

It was ugly, it was messy, yet I really wanted to keep the smooth slopes.

So, from now on, each room *is actually one size bigger than it appears*. This allows me to paint the terrain correctly *and* always know which room a player is in. But when it comes to (visually) placing the room, I use the *shrunk version*, which is just one size smaller. So everything looks the same as before, just much cleaner (and bugfree) under the hood.

Another trick had to do with *performance*. Right now, I checked whether I needed to add a new room *every frame*, and checked whether I had to delete a room in the same function.

Of course … if players were speedy, this meant *every frame, loads of rooms were being added and deleted*. Which caused huge stuttering.

To solve it, I simply put it on a timer. It only checks if it should update once every second. And it checks it *separately* for removing and adding rooms ( = two separate timers that never fire in the same frame).

(Because removing a room is just as heavy (performance-wise) as adding one. It needs to remove all the terrain tiles, fill the space with solid tiles, update all entries in the map, remove itself from the path, and overwrite *many* pixels in the terrain mask.)

## Step 14: Running with the wolves

When I started this game idea, I thought of it as an “endless runner”.

You were sheep. A *wolf* would be chasing you, always from the left. And if it caught up with you, you died.

Now that the maps are *more varied*, it’s impossible for me to write a “computer player” (aka the wolf) that follows the players around. Additionally, such a wolf would only *punish* players who are already behind.

Instead, I invented the following idea: **the last player *becomes* the wolf.**

When you’re in last place, you turn into the wolf. You become faster, can skip certain challenges, but most importantly: **hitting another player takes a bite out of them.** (Which, most likely, causes them to fall behind and become the wolf.)

It will be a constant driving force, pushing you forward and making you take risks. But it’s not controlled by the computer or manually programmed by me. It’s a player, which is way better.

**Remark:** in *single player* mode, you obviously can’t have this. I think single player mode will either be *survive as long as possible* or *finish in the shortest time possible*. But I need to think about that some more.

Here’s what the wolf looks like:

IMAGE: debug\_sheepe\_8

Of course, it had to be toned down a little. After biting, it takes a second before it can bite again. (In general, if a “bite” would result in a player losing all their bodies *or* becoming too small, it doesn’t go through.)

And the wolf has to be *way* more visible than it currently is. I just drew a very quick sprite to test it.

**Remark:** when testing this behaviour, I was constantly disappointed about how … ugly it looked? Just boring and … weird. Then I realized: slicing a body is quite an operation. It shouldn’t be this surgical precision slice, it should be more like an *explosion*. So I added a *force* that pushes the new blocks away from each other (after slicing). This solved *so many issues* and made it look much better in general.

## Step 15: The Rolling Factor

All this time, I’ve been thinking: “the game is called *rolling in the sheepe*, it’s about rolling … yet I spend an awful lot of time in the air”

Why? Because non-circular shapes are just really hard to roll. (And most powerups/rooms, for now, require jumping to solve.)

And then it hit me. A new rule for the game that would be amazing, if it worked.

**Rule: the *more you roll*, the *more round your shape becomes*.** (Conversely, the more time you spend in the air, the more it deforms.)

It took quite a while to implement this properly, as I’d never done anything like it before. But this is how it works:

General module:

* Each frame, I track if you’re in the air or not. (Which simply means: are you touching terrain or not.)
* After a few seconds, I take an average from that.
* If you’ve been in the air 50% of the time (or more), you deform
* Otherwise, you become more round

To become more round:

* Check the total number of points in the shape. If it’s below 5, you’re a rectangle and can’t be round. So, I “enrich” the shape by adding a new point halfway each existing edge.
* Loop through the points and make them global. (By default, these are local to their own center.)
* Approximate the radius of the body. (Simply get the approximate area. For a circle, A = pi r^2, so r ~= sqrt(A/pi).)
* Calculate their *angle* and *distance* to the center of the body.
* Move the point to stay at the same angle, but with its distance closer to the *radius* of the body. (In other words, each point moves closer to the position *it would be* if the body were a perfect circle.)

Do this a few times (for about 30 seconds), and your body has become a (near) circle.

After a while, I noticed this has issues if points are really close to each other (on the original shape). This would prevent the shape from rounding well, as they all moved to the same new location. Solution? I *snap* the angle to the 8 predefined directions (horizontal, vertical and diagonals), ensuring that most of the points will space out around the circle.

For deforming shapes, I did the opposite: move points to a rectangle. (In this case, I already have the *bounding box* of each shape, so I can just move points towards that.)

This is *fine*, but a rectangle is also quite a smooth shape. Which means it doesn’t really “deform” you, it merely “gives you sharper corners”. I’ll have to see what I do with that.

**Remark:** I switched my area approximation code to the **Shoelace algorithm (by Gauss)**. It’s just as fast, but *way* more accurate. Didn’t know it existed before now :p

I want to do even more with rolling, like buttons you can only activate by rolling over them. (And I have to become worried about performance at this point … manipulating dozens of physics shapes constantly.)

## Step 16: rolling with the punches

By now, I’ve made these comments several times:

* I should do more with the fact that you’re rolling.
* The game should be simplified so much that anyone can play. If I have to, the first level should just be *pressing one button* (to roll right).
* I have way too many different mechanics outlined (terrains, obstacles, special rooms, powerups floating in said rooms, things that can protrude from the terrain …) These should be streamlined and simplified.

Just now, finishing all the previous tasks, I had an insight.

I’ve done all this work to ensure a route that flows smoothly, where you can always reach everything, with enough space between rooms to fill it with solid tiles.

This means that there is **one place where I can always put something special: inside tiles.**

Coincidentally, rolling is also something you do **on a tile.**

Which led me to the following “rule” for this game: **all special elements are simply an extension of an existing tile.**

* Want to place spikes? Find an existing tile and place the “spikes” object on top of it.
* Want a button? Find an existing tile and place a “button” object on top of it.

This means *no* powerups (or other obstacles) just floating in mid-air. Which is great, as it gives us back some clarity and space. (Especially when players are sliced up, in smaller rooms, it becomes *quite full*.)

This means *no* special cases for certain elements, like a laser that spans the full width of the column. (Instead, I can repurpose that to the new system: I can stick a laser gun into the wall, protruding just slightly. Once in a while, it *shoots* that laser across the room. Same functionality, but the underlying code and mechanics stay consistent everywhere.)

So, how do we create a system like that?

* Pick some “special thing” we want to place
* Get a list of all tiles within our room.
* Pick a random one. Look at its neighbors to find an open side.
* Check if it’s a slope. (If so, our sprite just needs to rotate to match the angle.)
* Place the thing at the desired location and rotation
* (Remember it was placed there, so we remove it properly when the room is removed.)