# 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>

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. (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.

**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 you 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*.