# 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*.

That last part is actually where I got stuck for a bit. The algorithm would work … erratically, but I couldn’t spot any errors or logical reasons why.

**TO DO:** Explain the importance of “epsilon” in the line segment algorithm