# Devlog: Carving Pumpkins & Dwarfing dumplings

Welcome to the devlog for my game **Carving Pumpkins & Dwarfing Dumplings**.

It’s going to be a *really short one*, as this game is basically a spinoff from another game I was making. (It’s **Rolling in the Sheepe**, coming soon.)

In that game, I implemented a system that could *slice any shape (realistically*). So, for example, a player could be a *hexagon*. When I drew a line through that hexagon, it would *split* the shape into two parts. (Which, if you were to glue them together again, would represent the original hexagon.)

It was really cool to figure out *how* to do this. It’s “relatively easy”, though still quite challenging. (Especially when you get to supporting *any* shape, not just the “nice ones” like circles, rectangles, etc.)

However … as the game progressed, the mechanic just didn’t fit anymore. It was more *fun* to split players non-realistically. (Splitting a hexagon would just yield two smaller hexagons.) It fit better with the mechanics, the gameplay, the feeling of the game.

Determined to not let my code go to waste, I decided to create a quick little game that *did* use it!

As Halloween was coming up, it became a silly party game about slicing ( = carving) pumpkins.

**Remark:** I will *not* explain the algorithm for slicing shapes (in 2D) here. It’s quite complex and I discussed my journey of discovery *at length* in the other devlog. This is meant as a short devlog discussing only the interesting bits from *this* game.

## The idea

It’s simple. You can move and you can throw a knife. If the knife goes through another player, you literally slice them in two. The biggest part remains ( = *you* *are the biggest part*), the smaller parts will be lost and out of your control.

Any player who is too tiny, dies and is out of the game. The last one standing wins.

## Step 1: Sometimes you don’t need all the physics

At first, I implemented knives in the “traditional way”:

* I gave them a (narrow, rectangular) body
* When you threw them, I apply an *impulse*
* When they hit something, I decide whether I want to *slice* it ( = hitting another player)
* If not, I simply *bounce off of it* ( = hitting a wall) and let the physics engine do its thing.

This didn’t work.

Why not? Because *slicing something* is completely different from *hitting something*. They are, in most cases, complete opposites.

* To hit something, you need a body with some “area”. Slicing something means cutting it *along a thin, zero-width line*.
* It’s really hard to tell the physics engine to “delay” colliding with something. They’re not built for that, and for a good reason.

As such, the code would work 50% of the time. But the other times, one of these situations would happen:

* The *body* hit something. But when I shoot a line from it, the line *missed* that object. So we clearly hit someone … but still didn’t slice them.
* We sliced someone. But, the *collision* also came through, which means our knife had some random rotation/offset added *before* calculating the slice line. Leading to wildly unpredictable slices.
* Sometimes, if the knife was going fast, both cases would simply fail and nothing happened.

So let’s return to that first remark: **slicing means a zero-width line.** That means … we don’t need the physics body (on knives)!

I removed the body (and its shape). I added some code to handle *velocity* myself. (Simply move according to velocity each frame and dampen it a little.)

Then I added a **raycast** just ahead of the knife. If it hits a player, it shoots a line straight ahead, and slices the player across that line. All bodies that come out of it are saved as “exceptions”. These will *not* be picked up by the raycast from now on.

(Otherwise, it just keeps slicing and slicing every frame, because it will *keep hitting the player* until the knife comes out on the other side.)

If it hits something else, I simply *deflect* the knife. There’s a basic formula for deflecting a force/velocity:

<TO DO: CODE HERE>

With these simple steps, we have a knife that can both *slice* and *collide* (realistically) … without actually having a physics body.

I only use the physics engine for shooting that *raycast* into the world. Otherwise, the knives are completely handled by my own code, which isn’t more than 40 lines.

## Step 2: Throwing and catching

Each knife has an *area* attached to it. (It’s called Area2D in Godot, my game engine. Many others call this a *sensor*.)

If this overlaps with its owner ( = the player that threw the knife), you pick it up again.

Of course, this has one issue: when you throw a knife … it immediately overlaps and you pick it up again! Which means nothing happens.

As such, just after throwing, I “disable” this area for 0.5 seconds. (This has the added benefit that throwing the knife into the wall, and immediately deflecting, will make it go *through* you instead of nothing happening.)

Then I added some simple code to reposition the knives correctly. (On the edge of the player shape, whatever that shape is.)

It uses the **Shoelace algorithm** to estimate the area of the player. We know that, in a perfect circle, Area = pi \* r^2. We can reverse that to get an *estimate* on the player radius, which would be r = square root(Area / pi).

The knives are placed this distance away from the player, a bit offset from each other. This way, they stick out nicely, whatever your shape.

At first, I “repositioned” the knives to always be in front of you. (Which is logical, as that’s the direction you’re facing, and that’s from where people usually hold/throw knives :p)

But I soon discovered this had issues and that there was a better idea: just keep the knifes *wherever you caught them.* If you catch your knife with your back … well, guess you’ll have to put some extra effort in aiming it later.

## Step 3: Cleaning up the mess

Realistically slicing everything has one downside: you can end up with loads of ugly, tiny shapes floating around.

That’s why there’s a minimum area. If a shape falls below this – again, this is estimated using the Shoelace algorithm – it’s simply deleted immediately.

Similarly, the parts that fly off a player have some damping attached to them, so they don’t just endlessly float around. After a few seconds, they will have stopped rotating and moving, and just lie on the arena as a sort of “evidence of what happened”.

TO DO: Image??

Another thing that makes it “cleaner” is that I *separate* knives you grab. At the start, I define X “predefined angles”. Whenever a new knife arrives, it snaps to one of those angles. (If that number is high enough, say above 20, the difference between the real angle and the snapped one is negligible.) Is the angle already occupied (by another knife)? Try another one, until you find an empty spot.

It makes it *so* much easier to see how many knifes you have and where they are pointing.

Lastly, I’ve learned from previous games that it’s actually not a great idea to have separate menu screens. Many games, when the game is done, will go to a different screen that says something like “Game over! This player won. Press one of these buttons to continue.” (This is often an overlay as well.)

I’ve found this to take players out of the experience. Additionally, you *certainly* don’t want to switch to a mouse every time (when the rest of the game plays on keyboard/controller).

Instead, when the game is over, each player simply gets a “bubble” next to their head. The winner gets a crown! The losers get a “title” based on their accomplishments. (You’ve moved more than anyone else in the game? You are a *Runner!*)

One of the players gets a bubble with the instructions (Restart or Exit), which are completely controlled by keyboard/controller.

This makes the whole experience much faster and more streamlined.

## Step 5: Making a first level

Now we need these things for a first level:

* The core game loop. (Know when it’s game over, do something then.)
* An arena in which to play. (Some obstacles, a background, etc.)
* Some powerups would be nice. Something basic like “you get an extra knife”.

### Core game loop

First step is easy. Whenever a player becomes too small, I send a “player\_died” signal to the state manager. It checks how many players are still alive. If only 1, we go to “game\_over”.

In that state, all those *bubbles* appear next to the players. Additionally, I turn off anything I don’t need (like, we don’t need to check for “game over” again *if it’s already game over*), and turn *on* the keys for navigating.

TO DO: Image (game over bubbles)

### Arenas

From a previous game that has some similarities to this one (*Totems of Tag*), I’ve learned that it’s best to manually create the whole arenas.

(Instead of, for example, creating a bunch of tiles and reusing them everywhere.)

It easily allows each arena to be completely unique (with visuals and mechanics not used anywhere else), without requiring me to spend time “abstracting” or “generalizing” all objects and tiles in the game.

As such, the first arena will be the *graveyard*. I’ll just draw a background, some decoration, and of course the tombstones. Then I import these to Godot, give them the necessary physics bodies/scripts/groups

**Remark:** *Groups?* In Godot, you can put everything into groups. It’s *really* useful. Now I have three groups: Sliceables, Deflectables and Stuckables. These aren’t actual words, I’ve simply always named groups like this. Why? Because it immediately tells you what the group *does*: the first type can be sliced, the second deflects knives, and the third makes knives get stuck inside them.

TO DO: Image (graveyard arena)

### powerups

For the powerups, I invented something nice, I think. Instead of doing it the normal way (powerups spawn, if you like what you see, grab it) … what if powerups came inside a package? And you need to open that package to see what it is?

And to open packages … you need to slice them, obviously.

I like this for two reasons:

* Picking up powerups is still easy: walk onto them
* But there’s a gamble: do you think the powerup is good, or are you going to check by throwing a knife against it?

There’s one issue, though. If you *don’t* check the powerup first … you don’t know what you’re getting. So there must be some *very clear feedback* about what you just grabbed.

At first, I wanted to give each player their own “interface” in the corner and show your powerups there (as usual).

However, again, I’ve learned this isn’t ideal.

* It takes up a lot of space.
* It limits me to 4 players maximum.
* Players need to constantly switch between *looking at themselves* (and what’s happening around them) and *looking at some corner of the screen that happens to hold their interface*.

As such, I will simply create *clear* *icons* for each powerup. These appear above your head for 1 or 2 seconds, then disappear.

Additionally, I’ll try to give each powerup a *permanent* reminder. Easy example: if you’re a ghost (and cannot be hit by knives), you become 50% transparent.

All of this together, makes the game “UI-less”. Which is amazing, if it works.

TO DO: Image (powerups appearing)

## Step 6: Top-down perspective

At this point, I realized I never made a game with a top-down perspective before. (Well, except for some abstract puzzle games, but then it’s not really a perspective but just “geometric shapes in a grid for clarity”).

This led to mistakes. I drew (and programmed) some things, by force of habit, to appear *above* the player or to look good *when viewed from a certain angle*.

But in top-down view, you lose all that perspective. There is no “above” or “below” someone to show information. (The only “above”/”below” is in terms of *depth*. For example: players will be rendered *in front of* the ground.)

I tried some things, but nothing really satisfied me. It either didn’t look good enough *or* wasn’t clear enough during gameplay.

In the end, I settled on this:

* A “distorted top-down perspective”. Which means most things have no perspective, but the bigger elements near the edges *do* have some depth to them. It’s like watching down a hole, where things get flatter and flatter as you come near the center.
* Powerups are displayed *literally on top of the player*. The icon appears, does a bounce, then fades away. It’s not ideal, but it’s good enough.
* Knifes are drawn with a sort of side view, that still looks good from top-down perspective. (Because, if you draw a knife top-down “realistically”, you’ll barely be able to see it, as the blade is too thin.)
* Most (important) things have a thick *outline* to make them stand out more.

As I make this game, I’m learning more and more about how to deal with this perspective. It’s a work in progress :p

**Remark:** by now, I also removed the “Area” from the knives. Looking at the code again, I realized I could do *everything* with that single raycast I was shooting. So this simplified the code and made it a bit faster. (Just in case there are ever going to be *loads of knives* on screen simultaneously.) Additionally, my engine was complaining that it can’t re-parent physics objects during the physics calculations. And when a game engine complains, you better listen, or you’ll run into hidden and annoying bugs soon.

TO DO: Image?

In a similar vein, I modified the *raycast length* to look *further ahead* if the knife moves faster. Otherwise, if the knife goes *really fast*, it might miss a collision and “tunnel” through something.

## Step 7: Some interesting details (maybe)

In case you were wondering, this is how I implemented the more unique powerups.

**Grow/Shrink:** When setting the shape for a body, I already reposition all points so they are around (0,0) (locally). This ensures the shape is around the “center of mass”, which is how it should be.

This means that, to grow/shrink a shape, I only need to **loop through all the points** and **multiply each by a number**. Number greater than 1? The shape grows. Smaller than 1? It shrinks.

**Morph:** I thought about *actually* morphing from one shape to the next. Then I realized that was too difficult for such a simple game.

(After some research, I got the general gist of it: convert both shapes into a *signed distance field*, which is just a grid that tells you the distance to the closest edge from each cell. Then take a weighted average between the two fields, depending on how far you want to morph. But by this point I was like: nah, not worth it.)

So I just drew a bunch of basic shapes in the editor. (I used an image as reference and just placed points on top of it.) These are loaded when the game starts. When you morph, it picks a random shape from the list, resizes it to keep your “current size”, and then swaps the shapes.

**Reversed controls:** at the start of each frame, I collect player input into a vector. When controls are reversed, that vector is simply multiplied by -1 before sent to anything else.

**Curved shots:** there’s a simple formula for calculating *curve* on a spinning object. You simply calculate something called the **Magnus force** and apply it each frame to the velocity.

This force is defined in 3D, so to make it work in 2D (easily), you just need to fake it.

<TO DO: Code here>

**Boomerang:** boomerangs are easy to implement if you follow a simplified model. The boomerang has two states: “flying” and “returning”.

When you throw it, it’s **flying**. It will just do its thing as always.

As soon as it hits *something*, it switches to **returning.** It calculates the vector towards its owner ( = the player that threw it) and uses *that* as the new direction. I call this the *target velocity*.

Of course, this is a bit *too* precise. (It just goes back in a straight line, probably just the reverse line it just traveled.) To make it curve, you simply *interpolate* between its current velocity and the target velocity. To make it even nicer, do a *spherical interpolate*. (Because we’re talking about vectors and rotating here.)

**Ice/Skating movement:** The idea is the same as the boomerang curving. The player input is the “target velocity”.

Normally, the velocity immediately updates to the target.

When “walking on ice”, it interpolates, so that each input update is a bit “delayed” and you keep continuing in your original direction.

**A philosophical remark:** It’s interesting. Many of the things I use in this project I take for granted. Within a *two days* I had everything up until this point, and it still felt like I could’ve gone faster.

But … then I realized that 80% of the things I’m doing were *impossible* to me before the start of this year. Large parts of the code in this project are directly copied from other games I made earlier this year. There are things I use *a lot* here (e.g. directly checking the world for a collision *without* requiring an actual body) which I didn’t even know were possible 6 months ago.

It’s cool to see that progression. It also makes me wonder what stupid things I’m doing now which will, in 6 months, make me go “I wasted 3 hours on *that*?! That should be a 5 minute thing!”

It’s even funnier when you copy old code and immediately spot a *huge* mistake you made there. Which explains that odd bug that sometimes appeared in that specific game :p In a sense, my games literally only get *better* with age.

## Step 8: Teams & AI - The forgotten features

In that similar previous game of mine (*Totems of Tag*), there were some features for which I didn’t have time.

The most important ones were:

* Teaming up
* Computer enemies

(Both of these basically enable the game to be played with much more different *player counts* and *player types*. Totems of Tag has no single player mode. This game should.)

The first one is relatively easy to fix. In the menu, players should be able to press a button to switch teams. In the game, you cannot hit your teammates (or you can turn “friendly fire” on in the settings somewhere), and you win if only players from the same team are left standing.

Here's a mockup I made for the “game configuration” screen. (The final one will probably look slightly different, as I figure things out along the way.)

TO DO: Game Config Mockup

In *Totems of Tag*, I added the configuration as different “screens”. First you had input. Then you got a grid with *all* ball types in the game, and you could select which ones you wanted. Then a grid with *all* powerups. After 4 of those screens, you could start the game.

It had two advantages: it looked good *and* encouraged players to check out all the content in the game (and try different things).

But it had a huge disadvantage: you had to go through all screens. It took some time. It was annoying, especially on your first play.

As such, this game will only have that single screen from the image.

* Players can be added (or removed)
* It shows an overview of the *current* active settings for the game.
* And you can immediately start (or quit).

If you want to change or see those settings, you can press the indicated button. Only *then* do I switch to the old system of “individual full-size screens where you can pick the things you want”.

### How to make bots?

The second feature is, obviously, much harder. How do you make competitive AI bots? Ones that can provide a challenge (no matter the arena or situation), without being predictable?

These are things I’ve learned from another project I’m working on (which has many “AI”-like elements):

* There are actions that are “always sensible”. If you just let the computer do those randomly, quite often, it works surprisingly well. For example: throwing a knife towards the center of the screen is usually worth *something*.
* It’s better to give the AI **personality**. Instead of one AI with fixed parameters/decisions for everything, give them some leeway. Make one prefer hiding, another more aggressive, another powerup-hungry. Things like these can be *random numbers* or controlled by *probabilities*.
* That idea of “picking a target” and “slowly going to it” is usually what you need. In this case, it’s no different. The computer should just *pick a player to target (sensibly), position themselves for a throw, then throw*.

Of course, that last part takes time to figure out. Because throwing directly at players all the time is certainly not the best move.

* Instead of throwing directly at players, computers should *predict* where they will go.
* If an obstacle stands in the way, there’s no use in throwing, so they should just chase the player.
* The computer should be able to see if the *other* player can hit *them*. If so, prefer walking to a safe location.

Generally, the AI script should

* Read the situation around them. (Collect as much meaningful info as possible, such as the closest player, average distance to all players, etc.)
* Which gives each possible input a certain “score”. (If there’s nobody in sight, the score for “throwing a knife” should be lower than for “move towards the action”.
* This score depends on the personality of that AI, which is a (somewhat) random set of number and probabilities.
* And finally pick the option with the best score.

I’ll figure out the details over the coming few days, so let’s continue with something else for now.

## Step 9: Other game modes

So far, I’ve worked on the default game mode: deathmatch. You die if you’ve become too small. Last player standing wins.

Pretty basic stuff. Which is also why I wanted to add *more* game modes.

When playing, I noticed that the “leftover parts” (from a sliced player) were a bit annoying after a while. They clogged up the field.

I wrote a script to make them “fade out” after some time. But that felt like a wasted opportunity! Instead, what if we had a game mode where you had to *collect* parts from other players?

That was the first spark, which led to these game mode ideas:

* **Deathmatch**
* **Collector** => eat slices from other players by moving over them. The first to X slices wins.
* **Bullseye** => targets appear across the map, hit them to score points. The first to X points wins.
* **Dumplings** => players can eat dumplings to grow themselves. These dumplings, then, appear *inside* your body. When somebody slices through them, they take away your dumplings. First to X dumplings wins.
* **Dwarfing Dumplings** => eachplayer/team gets *one huge dumpling* to protect. If it becomes too small, you are out.
* **Ropes** => each player has gems attached to them with a rope. (Maybe not gems. Just something valuable, Halloween-themed.) Obviously, slicing the rope cuts those items loose. Lose all your gems and you’re out.
* **Capture the Flag** => one player from each team has a *flag* inside of them. However, this is hidden information. (The player who has the flag cannot throw knives, that’s how they know.) If you slice through that player, you capture the flag. The first to X captures wins.
  + This would require *teams*. I see no way to adapt it to single player or individual players.

I’m not sure if I’ll be able to make all of them in time. The further we get on the list, the harder they become. But the first ones should be doable.

Especially since they share a common core: collect things, win by collecting more than a threshold.

This did leave me with one issue though: **where do I show how many things you collected?**

Players can be any size or shape. I already show your *powerup* and *orientation* on top of you, so there’s really no space for a big number there. I still didn’t want to add *interfaces*. So what to do?

Then I remembered a trick I used in an earlier game: **making the interface part of the level.**

What if each player had a “home base” (just an image of a small castle, or something) that was simply **part of the level**? Then I’d have a logical, easily visible location to show how many a player has collected.

In fact, this can add to the challenge. Instead of increasing your counter *immediately*, you first need to successfully bring you items *to your home base*.

Here’s what these “interfaces” look like (a rough first version):

TO DO: Image (interfaces)

In the end, the first few modes were indeed relatively easy, as they share a common core: be the first to collect X of the same thing. That’s easy to generalize, even if the thing (and how you collect it) is wildly different.

The later modes proved much harder. And as I didn’t have much time, I decided to focus on some core gameplay elements first. (And only make the extra modes if time permits.)

## Step 10: Finishing the basics, adding the content

I’ve been testing the game a lot, obviously, and tweaked many things. Here’s an incomplete list:

* Increased physics FPS + fixed a stupid bug in my code to get rid of *all* tunneling issues.
* Added many improvements to things that weren’t bugs, but still didn’t suit the gameplay. (For example, if you only slice someone “halfway”, it does nothing. Because you didn’t *fully* go through them. It’s realistic … but not so much fun, and not what players expect. So I extend those halfway slices to go through something completely, in most cases.)
* You can only pick up powerups *after* you sliced them ( = “unpacked them”). I also made them way more visible, with a thick outline, a flickering animation, a bigger size, etcetera. It was just better, as it prevented “accidentally triggering powerups” and made the screen less chaotic.
* Added clear indications *who* owned a knife. (Its outline is the color of the owning player. If it’s free for all, it has a rainbow outline.)
* Added many effects and animations to slicing. This makes it more impactful, but also more clear. (Until now, it could happen that you sliced someone … and the body parts stayed together quite well, so you didn’t even know exactly *how* you sliced the other.)
* Added *probabilities* to powerups, because some are *way* more vital than others. (The “extra knife” powerup is huge, as you only start with a single knife, which you can even lose. Something like “move slower” is much less important, in that sense.)

I’m still unsure about the moving and aiming inputs. It feels like a different control scheme might be easier. This was the main contender: **Use left/right arrows to *rotate left/right*. Use up/down to move *forward/backward*.**

Another idea was: **shoot automatically on timed intervals (e.g. every 5 seconds).** This means players aren’t required to press/hold the button, making the game more accessible and easy to control … but also limiting my options.

As always: I don’t know until I test it.

What do I think? **Yes, I think this should be the default.** It’s not that it’s *obviously better*.

But here’s why I chose this new control scheme:

* It’s a game about aiming. If you can only rotate *by also moving in that direction*, it actually makes aiming quite annoying.
* (Usually, this is solved by adding a second button or a mouse. For example, shooters usually allow you to move with *left joystick* and aim with *right joystick*, independently. However, as this is a local multiplayer game for 1-8 players that also supports keyboard … I can’t do that.)
* It’s a top-down game where rotation is vital. Moving in four directions only makes sense (and is probably always the best option) when rotation does not matter.

The downsides of the new scheme are:

* Can’t control when you throw your knife. (Although we can modify this with, for example, powerups that increase the speed of throwing.)
* It might take some players an extra step to understand it. (“Always move in the direction of the key you press” is more intuitive at first glance.)

As such, the old control scheme will be fully supported and is something you can turn on in the settings.

With that done, all the basics and essential mechanics/systems/rules are in the game. Now it’s time to add the content: arenas, special elements/items/locations, and of course loads of polishing with sound effects and particles.

## Step 11: Arenas

Earlier you already saw the first few attempts at an arena (the “graveyard”).

These taught me that I need to make them *bigger* and leave *enough room for players of any size*. They also taught me that playing with lighting and weirdly-shaped physics objects is cool.

Lastly, as this is a game about *slicing* things (realistically, any way you want), I feel like that should be prominent feature in any arena.

First ideas were: an arena which is almost completely filled and you need to *slice* your way through it. (Like an overgrown jungle where you need to cut all these vines to get a path for yourself.)

An arena with big blocks that can be sliced sometimes, but deflect knives at other times. (This way, the possible deflections in the level constantly change.)

Things like this. All with a bit of a Halloween theme, although it’s not too strong, as I don’t like games having time-limited appeal.

This is the final version of the graveyard. It has some more decorations (such as the gates), which is mostly to reinforce the perspective and add more depth. (Otherwise it just looked too “flat” and “basic”). If I had more time, I’d add way more tiny decorations, such as bits of grass, imperfections on the tombstones, flowers around them, etcetera.

(As always, we’re doing a “one week game” here, which means I need to strip any fancy stuff that is “non-essential”. If the game turns out good, or I feel motivated, I can *always* improve it later. Conversely, if I spend all my time now on drawing one beautiful arena, the game might never even see the light of day.)

IMAGE of final graveyard image

I decided it might be better to start with a “simpler” map that was a bit more intuitive, so I created “the spooky forest”.

IMAGE of final spooky forest.

This screenshot made me realize the light was too dark and players probably need their own (weak) circle of light anyways. Some of the trees are also interactive – some can be chopped down (creating more space), others auto-throw knives once in a while.

I also made knives more visible with unique colors *and* a “flickering” effect when they are standing still (indicating they can be picked up by *someone*).

Then I wanted a map where *almost the full map* consisted of sliceable objects. This is how the “dark jungle” was born.

## Step 12: Controls, now Properly

I was able to do a quick playtest. (Very quick, just 10 minutes with a random family member.)

The problems were … stupid and obvious, in hindsight.

* **Too few knives**. (If you’ve thrown a knife … you also can’t open powerups. So there’s no way to get another knife, unless someone gifts it to you.)
  + **Solution?** Start with more knives. Regularly, spring open a powerup (automatically) and set it to “extra knife” type.
  + **What makes it worse?** The “quick slash” action is a bit overpowered, as you don’t lose the knife after doing it. This, combined with knives being a rare commodity, causes people to only use this! => There’s now a 5 second cooldown on it, and you don’t need to hold the button very long to make it a throw.
* **Controls.** The idea of “rotate left/right” (instead of move in four directions) was added to make *precise aiming* possible on keyboard. On controller, it’s obviously not necessary, as you can aim anywhere with joystick. *Additionally* … there’s no need to turn this on during movement. I can just switch to those controls *during aiming*.

Otherwise, things worked as expected, the menus worked (and looked really nice), and it’s starting to become a game!

Some other minor tweaks were:

* The fact that you lose ownership of your knife when it gets stuck is now an *optional rule* you can turn off if you want.
* When you’re smaller, you move slower. (So that the “relative speed” stays the same. Otherwise, it would feel like you were *racing* over the field if you were small. Which felt weird and “off”, but also gave you a significant benefit, as you were much harder to hit.
* By default, players start with random shapes. (I have a list of 20 basic shapes, like rectangle, circle, triangle, hexagon, etc.) Again, there’s an option *“everyone starts as a pumpkin”* you can turn on.
* Decided to add all “special objects” as powerups, to keep the game streamlined. It also prevents me from having to *explain* each special object with an in-game tutorial, as powerups are explained in the settings menu when you hover over them/turn them on.
  + Example: dumplings are powerups now.
* Also added some other really useful powerups to enable by default. Such as “repel knives”: repel any hostile knives near you, which is basically a shield but more fun/dynamic.

## Step 13: Bots

Earlier I gave a general idea of how bots should work.

After thinking about it for quite a while, I realized I needed to stop thinking about it. **I needed to draw a diagram.**

Below is the picture that explains how a bot thinks in this game:

Lala TO DO => show the graphic I made for it, probably the clearest

It can be summarized as:

* Immediate threat? Drop everything and avoid it.
* Can’t do anything because we’re out of resources? Find a resource (most likely a knife.)
* Something useful nearby for the long-term goal? Grab it.
* Still here? We have time and space to attack!

By changing the weights and probabilities, I can make bots more “defensive” (prioritizing the first few parts) or “aggressive” (prioritizing attack, even if the other conditions aren’t met).

Avoiding obstacles uses a basic dynamic physics check:

* Shoot a few raycasts ahead of us
* If any of them hit, we can’t move there. So try the same movement, but rotated to the left and right.
* Continue rotating further and further, until we have safe passage.
* If we had to rotate a lot, the bot goes into “stuck” mode and *tries to get away from all obstacles* for the next 0.5-1 seconds. (It doesn’t do anything else.)
  + If we don’t do this, it will keep rotating endlessly, without ever moving out of that space. This fix isn’t ideal, but it solves 95% of the cases.

This works quite well, better than expected! Bots are quite a challenge already and mostly feel like you’re playing a smart human being.

But as I said, it’s not *perfect*. Bots always have this issue: there are situations in which a *human player* would easily see what to do, but a bot just gets lost and does something idiotic.

Bots can still get stuck and will just stand there helplessly, especially if they’re shooting for a target that’s *just* on the other side of a wall.

To solve this, we could use actual *pathfinding*. Godot has “NavigationMesh” built-in, which I’ve never used before, but looked like the perfect fit.

To use it, I

* Draw one big mesh for the whole map
* Cut out all static bodies. (So they are “holes” in the NavigationMesh.)
* Tell the bot to find a path between its current position and its target, staying inside this NavigationMesh.
* Then just walk that path.

This *also* was surprisingly easy. With one big caveat: bodies that overlapped the edge of the screen would cause trouble. (It cannot create a proper navigation mesh if some of it extends beyond the bounds.)

Solution? Modify the points of that shape to stay within the screen.

The problem with that? Whenever a bot wanted to go near the edge, it would find a path *along the border of the screen*, as there was 0.0001 free space there due to floating point precision errors. Which led to paths that didn’t actually exist.

To solve it, I

* Check if a path does something like this. (One of the points is an edge point.)
* If so, set the target *halfway* the real target. Check if the path is still wrong.
* Continue until the path is valid.
* Because bots re-check targets each frame, it doesn’t matter if we get only halfway. Because, when we’re there, it would just calculate the next half from there.

With that done, bots are at least really good at navigating the map and aiming. Being fun to play against will have to be something I finetune.