# 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?

## 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.

The second 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.
* 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”.
* And pick the option with the best score.