# How I Created “Pizza Peers”

Hi, I am Pandaqi, an indie game developer who almost exclusively creates local multiplayer games.

I’ve been trying to create those kinds of games for several years now, which means I’ve hit every obstacle and roadblock imaginable.

In this article series I want to explain …

* **Why** I created “Pizza Peers” (and the underlying system)
* **How** I created it (with some code, some nice images, and a bunch of text)
* And why I think it might be the best thing I ever made and will be creating many more games like these.

If you’re reading this as a programmer, you’re in luck: I made the full source code publicly available! (I even did my best to clean it up and comment everything nicely.)

**Pizza Peers on GitHub (TO DO: Turn into a link.)**

These are the different articles in the series:

1. The problems with multiplayer and how to solve (some of) them
2. Web Sockets + Node.js
3. Peer to Peer connections
4. Using Phaser.io (version 3)
5. Player interface
6. Things to keep in mind (syncing players, disconnecting, smartphone weirdness)
7. Generating the city

## The problem with multiplayer

I want to create local multiplayer experiences for the whole family. The games must be accessible to all ages, to both gamers and non-gamers, and most of all: cooperative multiplayer.

(Not always, of course. I make single player games and competitive multiplayer. It’s just that cooperative is the best choice for families and non-gamers, because they don’t tend to go well with competitive environments …)

**First thing I ever tried**: one person using the keyboard, one using the mouse. As you might expect, this was a mess. (And didn’t scale to more than two players, obviously. Nobody is going to attach four mice (mouses?) to their computer.)

**Second try**: two people on the same keyboard. In fact, I even made games for three and four people on the same keyboard. While this worked and was kinda fun, it’s just very cramped and doesn’t *feel* fun. Pressing tiny buttons with letters on them is not an *intuitive* way to control a game.

**Third try:** controllers / joysticks / gamepads (however people want to call them). Controllers feel the most fun and intuitive when playing a game. The number of people with at least one controller is quite high (in my experience), and this scales perfectly to higher player counts.

I really like playing with a controller, and so does everyone around me (they are visibly having more fun if each of them gets their own controller), so this seems like the perfect option.

*However*, not everybody has a controller, and certainly not more than two. Additionally, to non-gamers a controller feels “intimidating”. They immediately get the feeling that this game is going to be difficult and that they won’t understand it.

(Additionally, the setup can be draining. Having to connect all the controllers to a system before getting to play a game, and having to *explain* the buttons, costs a lot of time and energy. By the time the setup is done, half the people have already lost motivation.)

**Fourth try:** what if … we could use people’s *smartphones* as the controller?

I know about the **Jackbox** games, but those are mostly (social) turn-based games.

I know about **AirConsole**, but that service uses a server that introduces noticeable delays, and the games on there are – in my probably worthless opinion – not that exciting or accessible. (Plus, it costs money. That’s not a criticism: it’s fair to ask money for such a service. It’s just that I don’t personally have the money to spend.)

Instead, I remembered an old technique that now seems long forgotten in the gaming world: **peer to peer**. (I’ll explain this more in the third article.)

If you are playing on the same Wi-Fi network, using a peer-to-peer connection is **as good as instant**. And indeed, in all the games I’ve played using my system, everyone immediately forgets that they’re playing over the internet because everything happens as if you were holding a controller.

Suddenly, everyone with a smartphone can join the game within 5-10 seconds! Most people *have* a smartphone and *understand* how it works, which means these games are not intimidating and do not require you to buy loads of controllers.

## So, what *is* “Pizza Peers”?

That’s what Pizza Peers is: **a cooperative multiplayer game in the web browser, where everyone can use their smartphone as the controller**.

More specifically, you’ll be running a pizza place and trying to prepare and deliver pizzas throughout the city.

To start the game, all you need to do is visit [**https://pizza-peers.herokuapp.com**](https://pizza-peers.herokuapp.com)on a computer. Press the “create game” button.

Now, everyone can whip out their smartphone and visit the same address in their browser. They simply enter the *room code* (displayed on screen) and a *fun username*, and then click “join game”.

It only takes 5-10 seconds to fully create and load the game. It also only takes 5-10 seconds for players to connect (which they can do simultaneously, by the way). No need to download or install anything, no delays, you’re playing in no-time.

I can’t stress enough how important this is. With my previous games, I’d need to ask, “do you want to try my game?”, and if they said yes, it would take a good 10-15 minutes to setup everything and get everyone ready.

Now I can ask that question, point everyone to the web address, and within a minute we’re already playing. Needless to say, I’ve already playtested this game more often than many of my other games.

## Why I wrote this article series

I would *love* to see a world with more games like this. Games you can play with anyone, games that are extremely easy to start and access, cooperative multiplayer with everyone being in the same room.

(Sure, you can also play remotely if the host screenshares his computer. But that will introduce delays and probably not make the game more fun.)

I’m sharing this project, my source code, and my ideas behind it to show people what is possible and to hopefully help other game developers.

**It’s not that hard.** Seriously, the code for connecting people and sending signals through peer-to-peer is only a few hundred lines. (Which includes error handling, comments, lots of whitespace because I like whitespace, etcetera.)

Additionally, I haven’t found anyone else talk about this or share these kinds of ideas. I might be the first one to do this, although I highly doubt that.

**It opens a whole new world of possibilities.**

A smartphone, for example, is not static (like a controller). You can *change* the interface during the game, you can send chat messages, you can shake the phone to do something in-game, you can even livestream everyone’s camera to the screen if you want. (Not sure how you’d use that in a game, but you *can*!)

I’ve tested this game on a nearly 10-year old iMac (as the host) and my own 5-year old smartphone (which was the cheapest available at the time). Surely everyone must be able to play these kinds of games :p

So, let’s get started! Topics like web sockets and Node.js servers, what an exciting time!

# [Pizza Peers] WebSockets + Node.js server

This is part 2 in my article series about how I created “Pizza Peers”. If you haven’t read the previous article, make sure to do so as it explains the basic idea behind everything.

So, first things first, we need:

* A server that hosts our game (and serves the game files)
* A server that receives connections from both the players and the computer, so that it can *connect* them directly. (Once connected, the whole game goes via peer-to-peer.)

Obviously, we’ll use the same server for both these things.

**IMPORTANT REMARK:** In these articles, I will not show the full code (as it’s too complicated and specific, thus not so good for teaching or explaining). Instead, I give a template and some pseudo-code where needed. If you want to implement these things yourself, check out the source code for all the details and exceptions.

## Node.js

For the server side, we’ll use Node.js. It is simple and small, it uses JavaScript (which we’ll also use in the game itself), and I have some experience with it.

Below is the template for this server. It simply sets up a server that servers both static files (which are the game files) and accepts websocket connections (which are needed to connect players with the computer)

<script src="https://gist.github.com/Pandaqi/1d5c9db5d60fb0612bd94cb6f29c6202.js"></script>

Of course, all the magic is going to happen in that “on(message)” block.

## Peer to Peer

Which messages do we need to send? For that, we need to understand how peer-to-peer works.

Instead of creating a connection with the *server*, a peer connection is a direct connection *between two devices*. So, once established, your smartphone has a direct link with the computer (that hosts the game), and vice versa. This is what makes it so incredibly quick and easy.

However, we cannot allow devices to just establish connections as they please. That would not be very secure.

Instead, there’s a handshake protocol we need to follow:

* Device A wants to connect with device B
* A creates a peer (on their side) and sends out an **offer**
* B receives the offer, creates its own peer, and formulates a **response**.
* Once A has received and validated the response, both are officially connected.

For generating the offer and response signals, I use the **simple-peer** library (TO DO: Link here). I don’t understand what it’s doing with those signals or what all the information means, and neither do you. I just pass the signals along.

Speaking about that: we’ve encountered an issue. A needs to send an offer to B. But … they are not connected yet. How is A going to find B?

That’s where our WebSockets come in!

## Signaling server

In order to exchange signals, we turn our server into a so-called “signaling server”.

The idea is very simple:

* A generates a signal and sends it to the server.
* The *server* determines the intended recipient and relays the signal.
* B receives it, creates a response, sends it back.
* The *server* determines that the response must return to A.
* And voila, peer to peer connection!

So, at our server, we need a way to receive and pass on messages.

In my case, this is even more simplified:

* The *player* (with the smartphone) is always the initiator of the connection.
* The *computer* (that hosts the game) is always the one responding.

See the code below for the basic structure of this system. (It will make even more sense once we’ve created the client side.)

<script src="https://gist.github.com/Pandaqi/bf2c6215922a302a042126bb60c245c3.js"></script>

In fact, this is almost *all* the code on the server. The only thing I added in the real game is more robust error handling and handling some extra cases/exceptions.

## Client side - Websockets

So far, we’ve created the server only. It accepts web sockets and passes along signals … but we still need a client side to actually create those web sockets and signals.

I assume you know how to set up a basic HTML page. If not, check out index.html in my source code.

Then, make sure the JavaScript below is run (*once the page has finished loading*):

<script src="https://gist.github.com/Pandaqi/33474e8bda094296af95e6b8d53a367d.js"></script>

This bit of code opens the web socket connection, then listens for responses from the server. When necessary, it creates a new peer.

The “peer.signal(…)” bit will be explained soon. Essentially, we just pass the “signal message” directly into the peer and let it formulate its response. (Remember when I told you that I don’t know what’s inside those signals? Those things are put in here.)

**NOTE:** All messages are JSON. However, we cannot (and don’t want to) send objects over the internet. So, before sending, we must always *stringify* the object. At the receiving end, we always *parse* it, so it returns to the original JSON object.

## Almost done …

All the code above still does not complete our system for connecting – so don’t try to run out – but we’re very close.

This is what we’ve achieved so far:

* A server that servers our game files.
* A server that accepts socket connections, gets messages, and then relays them to the right connection.
* A client side that connects with the server, and also sends/receives the right messages, and creates the proper peers when needed.

All that’s left to do, is actually create the peers. For that, see you at part 3!

# [Pizza Peers] Peer to peer connections

This is part 3 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so!

As stated earlier, I used the **simple-peer** (TO DO: link here) library. It’s free, it’s small, just grab it from GitHub and include it at the top of your index.html page.

In the code above, you saw the function “createPeer()”. In this article we’ll actually write that function.

## Creating peers

The code below creates the peer and then attaches the proper listeners (just like everything we’ve done before).

The “on(data)” call is where all the magic happens. As said before, once the connection is established, *all* communication goes via that listener and *none* of it uses the server anymore.

<script src="https://gist.github.com/Pandaqi/9553ee6e7646bc140bd4c0f77bcba4f4.js"></script>

At the top of the function you can see the actual peer being created.

The option “initiator” is true for players (because they *initiate* the connection), and false for the computer.

**NOTE:** The player only creates a single peer and then tries to connect with the computer. The computer creates a *new* peer for every player that wants to connect! That’s also why we can save variables on the peers relating to the specific player (such as its username or index in the game), because every player has a unique peer.

Remember: peer-to-peer is always a direct two-way connection between two peers. If you add a third player, for example, and want to connect everyone with everyone, you’d need to create *two peers per player*.

(I emphasize this, because it’s the reason it took me a whole day to get simple peers working. I didn’t fully understand what it did, what the “initiator” meant, and what ICE servers are. So, read on before you make the same mistake!)

## ICE Servers

The option “iceServers” requires some more explanation.

As said before, allowing any device to reach any other device is not very secure (and sometimes very hard to do). Thus, in some cases, you need so-called **STUN** and **TURN** servers to establish the connection.

* A STUN server basically acts as a mirror and allows a computer to find out its own IP-address, by bouncing a signal off of it.
* A TURN server is simply a middleman: it relays your signals directly to the other person.

As such, STUN servers are free and easy to get, because they barely do anything. TURN servers, on the other hand, will relay tons of data every second and are very expensive.

Luckily, these servers are rarely needed (perhaps 85-90% of the connections work without them).

Even better: **if you are on the same Wi-Fi, you should never need them**. You can just leave this option empty and it should work. (In the final version of Pizza Peers, I did acquire a free STUN and TURN server, just to cover all bases. But it’s not necessary for a *local* game.)

## What now?

All the code seen so far should be enough to connect to a server (and serve the game files), create a room, and allow players to create peer-to-peer connections within that room.

This is just the basic structure for networking. There’s no game or anything yet.

That’s our next stop along this adventure!

We’ll now look at the following elements:

* Starting and managing a game using Phaser.io
* Sending data from the game through a peer.
* The reverse: receiving data in a peer and sending it to the game.
* Creating the mobile interface => how to listen for input and other things to keep in mind.
* Creating the actual game => I won’t talk about everything, just the interesting bits like random city and kitchen generation

See you at part 4!

# [Pizza Peers] Using Phaser.io

This is part 4 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so now!

So far, we’ve created a system that allows connecting to a server, creating a room, and then directly connecting all players within that room (via peer-to-peer).

Now we just need a game that can send and receive information over those connections.

For browser games, there’s really no better option than the free **Phaser** library. I’ve been using it since the day it was created, and it’s only gained popularity (and features) since then.

(This surprised me a bit, to be honest. Feels like ages ago that I first learnt how to make browser games. I almost feel like a proud daddy who watched his kid grow and become one of the most used gaming libraries :p)

I must say, however, that I haven’t used Phaser the past couple of years. (For the simple reason that I wasn’t creating browser games.) Since then, version 3 was released, which made a ton of changes to the overall design and structure of the framework.

After using it for this game, I must say that Phaser is still awesome and that version 3 is again a leap forward! However, because it was my first time working with v3, my code probably can’t be called “optimized” or “best practice”.

Anyway, let’s get started!

*Remark:* By default, Phaser is a 2D library, but there are extensions for 3D in which I have a great interest. Hopefully I’ll be able to try these out soon and report the results.

## Initializing the game

I usually make the game full screen by putting it inside an absolutely positioned <div> element.

The rest of the webpage functions as an “overlay” on top of that.

This is one great advantage of browser games: they can use **canvas** and **regular website stuff** combined. For example, creating buttons, or links, or responsive UIs is *really* easy in website code. So I’d rather place those things in an overlay with minimal effort, than try to recreate it inside Phaser.

So, all the buttons for “create game” and “join game” and all that jazz are default HTML code. They are within the overlay and I will not discuss them here.

Once “create game” is pressed, however, a function “startPhaser()” is called that does the following:

<script src="https://gist.github.com/Pandaqi/a151bd4fecd00c4e91aa1d8629acc770.js"></script>

You’ll see here that **GAME** is a global variable. This is just an easy way to make it accessible in all modules – you can do this differently if you want to. I know global variables are generally bad practice.

Phaser works with **scenes**, which you can also view as “modules” or “components”. You can have as many scenes active as you like, you can toggle them on/off when you want, etcetera. If you want, you can structure your code to the extreme, creating a single scene for every bit of functionality. But we’ll keep it simple here.

The scenes **mainGame** and **gameOver** do what you think they do. We will not look at the game over scene, as it’s not interesting (and the source code is extremely self-explanatory).

Instead, over the course of the next few articles, we’ll look at the critical parts of the “main game”.

## Communication with p2p

How do we communicate over the internet?

Well, we need a two-way street:

* When a certain **player** does something, we want to send a message over the corresponding **peer**.
* When we receive a message from a **peer**, we want to relay that to the corresponding **player**.

During the game, we must keep a list that allows us to easily convert player <=> peer.

* The game keeps a list of all players. When we create a new player, we save its **index** in the list on the peer.
  + Yes, this means that the player list may never change order, but that’s a workable constraint.
* Conversely, once the player is created, we save the **peer** on the player.
  + This is easy: “player.myPeer = peer”

The addPlayer function becomes something like this:

<script src="https://gist.github.com/Pandaqi/60311f3acc1935e4ba6d977f34ae1133.js"></script>

**NOTE:** The variable **this.players = []** is initialized when the scene is created. I’ll remind you again when we talk about randomly generating the city and game world.

Now, whenever we *receive* a signal, we can convert the peer to the corresponding player.

And whenever we want to *send* a signal, we can convert the player to its peer.

That’s, in essence, all that is needed to communicate between the game and the p2p signals.

## Moving players

To hammer home the concept, let me give you an example of the most basic input in the game: movement. (Which is also the first thing I implemented and tested.)

The smartphone sends a message to the computer (using their p2p connection). This message contains a movement vector.

The computer receives this message and calls “updatePlayer()”, which is a very simple function:

<script src="https://gist.github.com/Pandaqi/db5c5842772525758dd4df6420a095b8.js"></script>

**That’s it!**

Well, we’re not completely done yet. I still haven’t shown you how to create the interface on the phone and send these messages. Guess what: we’re going to do that in part 5!

See you there.

# [Pizza Peers] Player interface

This is part 5 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so now!

Now that we can communicate with the game over the internet, we need an *interface* to make this communication easy and intuitive.

At the end of the previous article, I showed you the “updatePlayer()” function. We’ll be creating the interface for that in this article.

(The rest of the interface consists of buttons, buttons, and even more buttons. I’ll give one example of how such a button works, and that should be enough.)

## Receiving signals

In the previous article, I showed you how the *game* could send and receive signals.

Now I’ll show you how the *player* (the one holding the smartphone) can do this.

Remember the “on(data)” listener we placed on the peer? The one I told you would do all the magic? Well, that was no understatement.

It actually handles all signals for both computer and player. It’s just that the computer needs an extra step to relay this information to the in-game objects.

The code below is all placed within the on(data) listener on the peer.

This code is for buying ingredients. The first statement actually performs the buying action within the game. The second is triggered when a player *enters* a buying location, the third when that player *leaves* the location again.

<script src="https://gist.github.com/Pandaqi/d7b1fd9b15c034849239b78dcf573253.js"></script>

**NOTE:** Even though computers and players use *different parts*, it’s all collected within the same listener. (A player will simply never received a “buy” event, so it will always ignore that if-statement.)

Anything you want to happen within the game, just use the structure above. Make up the signals you need and send/receive them on the peer. If it’s the computer, relay the signal to the game object (if needed).

## Moving players, again

Phones do not have a joystick, or moving parts at all for that matter. I also don’t want to emulate a keyboard and print a bunch of (arrow) keys on the screen.

So, how do we allow 360 degree movement on the smartphone? Well, I chose the simplest route: **simply treat the whole screen as the joystick**.

Whenever you touch the screen, it calculates the vector between the *center of the screen* and your *touch*, and that’s your movement.

The code below is all you need.

(In the full code I also allow using a non-touch screen to control the game, but I want to keep it simple. I mainly allowed mouse input because it made testing much quicker on my laptop.)

<script src="https://gist.github.com/Pandaqi/fe547cf4a78b0a8b1460b62821f3228b.js"></script>

All we need to do to send a signal, is call “peer.send( message )”.

The message is, again, stringified JSON. We can put in any properties we like, but I recommend keeping it as small and simple as possible. You don’t want to waste any bandwidth with online games.

## What now?

You might be thinking: woah, it’s that simple? Why doesn’t everyone use this technique if it offers realtime (online) multiplayer?

(If you weren’t thinking that, well, eh, I’m gonna continue this line of thought anyway.)

Of course, there are downsides to this. There are things you simply cannot do and exceptions you need to take into account every step of the way.

The next article will talk about those!

# [Pizza Peers] Problems with P2P

This is part 6 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so now!

So, let’s talk about our current setup:

* The game is started on a single computer. This computer is the host and also the boss about any game logic.
* Phones can connect to this computer and directly send input.

As I created “Pizza Peers”, the big problem with this setup gradually came to light: **the game is basically played on X screens simultaneously**.

If you have 4 players, there are 5 screens that need to be updated (players + computer host). Most importantly, they need to **stay in sync**.

Let’s say you are standing at a table to drop an ingredient (for your pizza). Another player comes in and swiftly drops his own backpack on the table, just before you pressed your button.

What happens now? Is your input ignored? Are both your inputs valid? Do I need to update your screen every time something changes at the table, and how do I do that?

And what if one of the players disconnects? A few articles ago, I mentioned the importance of keeping the player list intact, because we use it to convert the *peer* to the corresponding *player*.

Let’s see how I tackled these problems. I don’t know if it’s the best way, but it’s *a* way.

## Syncing players

Synchronizing players and keeping an online multiplayer game “fair” is one of the hardest things to do. I’ve tried it several times and still fail to grasp some of the concepts.

(For example, a large part of it has to do with the server being ahead of all the players and being able to go backward/forward in time to evaluate the game state for a given player at a given timestamp. Yeah, try to code that.)

**Fortunately**, we do not need this. Because there is a *single screen* on which the game is played and hosted, we do not need to update the whole game on multiple screens.

All we need to do, is update the *interface* on each smartphone to match the current game state.

Because updates are (as good as) instant, we don’t need to be careful about this either.

In the final game, I simply do the following:

* Player A stands at a table and changes something.
* Now the game checks if any other players are at the same table.
* If so, it sends out a message to all of them with the *new* composition of the table.

I’ve had several games where two or three players were using the same table, and it never led to issues.

Of course, you do need to be very diligent with **error checking** within the game. Before any transaction, check if players are allowed to do this transaction. Don’t blindly assume that input is correct by any means.

For example, whenever someone tries to update a table, I *always* check the following things first:

* Is this player valid?
* Is this player actually at this specific table?
* Is the ingredient he wants to add a valid ingredient? (Not all numbers correspond to a valid pizza.)
* Is he allowed to do this (given the current ingredients on the table)?
* (And sometimes even more)

So far, this has never errored or caused glitches. Whenever somebody changes a table, everyone connected to it is instantly updated on the new game state.

The same principle applies to all other things.

**A good alternative** would be to change your game’s design. Simply do not *allow* more than one player to use something. Limitations like that often lead to cleaner, less error-prone, more elegant games designs.

**Another alternative** is a sort of “voting” system. If the game is uncertain about the reality or current state of things, it simply polls all players. The value that occurs most often is deemed the right one. (Really, in such a tiny local multiplayer game this doesn’t matter. Might even add to the fun.)

## Smartphone weirdness

A huge drawback with creating a browser game, is that all browsers have their own ideas about how things should work.

I’ve created many games that worked flawlessly on my device or browser, but completely broke down elsewhere. Sometimes, the fix was simple (add a vendor prefix to the CSS or look up when a certain JavaScript feature was introduced).

But sometimes I found out the hard way that I had to redo all my code, because a browser simply didn’t *have* a feature or implemented it *in the complete opposite way*.

Things to keep in mind are:

* Vendor prefixes in CSS
* JavaScript versions and supported functionality.
* Supported file types (e.g. for audio)
* Different screen sizes: usually best to use “overflow:hidden” (never allow scrolling on the interface) and minimize the use of dynamic elements that might mess with your size (such as images, videos, blocks that appear/disappear, …)
* Apple is annoying. They only update Safari when they release a new version/new product, and they are often late to the party on all features.
* Try to include *every single property* in your CSS. Why? Because each browser and device has a different *default look* for buttons, input, links, etc. You want the game and interface to look consistent.
* If you work with touch events, make sure to prevent propagation, otherwise the same event gets fired twice on most systems. Also make sure you manually check all your values, because I’ve found error logging to be quite useless when working with event listeners.

## Disconnecting & Crashes

When a peer falls away, the whole connection immediately breaks down. This is a side-effect of direct communication: there’s no server in between to mitigate this or solve the problem.

If the computer host falls away, the game is simply terminated and you need to restart. There’s no proper way to save the game state and restore it, unless we want to copy the whole game state to all smartphones at all times.

If a player falls away, however, we can fall back to the idea of “the host is the boss”. Whenever anything goes wrong, I simply pause the game and show an error message with possible causes and solutions.

Most importantly: I set a flag on the computer so it knows people will try to reconnect.

When you connect to the game, and the game is in “reconnect” mode, it just searches through the players until it finds the one with a matching username. It swaps the old peer for the new one (which you used to connect the second time) and tada: you’ve regained control of your old player sprite!

I’ve found this to be the most elegant and quick way to solve the issue of disconnecting and crashing.

There are still some problems, though, that I must solve at this time:

* If a smartphone goes into standby, the connection is also lost. In between games, people usually go for a snack or a drink, and by the time they get back they are all disconnected.
* The first player to connect becomes the VIP. This means that they get the button to “start the game!” or “play again!” If they disconnect … well, then I need some way to transfer that VIP to someone else.
* What if a player simply wants to *leave* instead of reconnect?

It’s my experience that gracefully dealing with disconnects and crashes is a never-ending story, that’s why I write this bit even though not all problems are resolved. They will never all be resolved :p

## Conclusion

Hopefully you now have an idea of the pros and cons of this system, and how to make it all work smoothly.

You also know how to setup the connections, the game, how to send and receive data, and how to properly act on that data.

I could leave you here and you could create your own peer-to-peer games!

(In fact, if that’s what you came to do, you can leave now and make your dream project!)

There are some parts of “Pizza Peers”, however, that I find too interesting not to share. I’m talking about the algorithms I used to randomly generate cities and kitchens. They are quite simple and naïve approaches, but they worked wonderfully (to my surprise).

I also have some things to say about Pixel Art (this is my first attempt at creating a pixel art game) and other aspects of game development in general.

So, see you in the next (and probably final) article!

# [Pizza Peers] Random city and room generation

This is part 7 in my article series about how I created “Pizza Peers”. If you haven’t read the previous articles, be sure to do so now!

This is what the average city in the game “Pizza Peers” looks like:

<IMAGE HERE>

There are ingredient locations, lots of regular buildings, and buildings that logically (and efficiently) connect them.

On top of that, there are three kitchens (or “workspaces”) which must contain X amount of tables and ovens, and which must be walkable. (If you can’t go inside the kitchens or reach any of the tables/ovens, they are worthless and the game becomes impossible.)

Today I want to share how I generate these elements.

(When expressed in code, these algorithms are quite heavy, which is why I’m not sharing any code. Check the source code if you want to see the exact implementation.)

## “Extend Road”

All the algorithms are powered by the the “extend road” algorithm.

It does the following:

* We’ve just placed something at grid location (x,y)
* Check if we are already connected with a road.
* If not, pick a random side (that is available)
* Now continue this road in a straight line until you are …
  + Adjacent to an existing road;
  + Or you’ve reached the edge of the map

<IMAGE HERE>

The first few things we place will get a road leading to the edge of the map. Those automatically become the “main streets” of the city.

All subsequently placed buildings will create a road connecting to one of the existing roads, which leads to a mostly realistic and good-looking road network. (It’s not perfect, of course, but I chose simplicity over complexity for this game.)

Because of the grid structure, once an *adjacent cell* is already a road, we are automatically connected to a road. Because this adjacent cell could be anywhere (left/right/top/bottom) we automatically get corners and curves in the road for free!

(Additionally, the road is as short as possible, and we don’t get ugly rectangles with multiple parallel roads.)

## Kitchens

As expressed in the introduction, kitchens have one important property: **you need to be able to walk from one entrance to another**. No tables/walls may block your way. Instead, they must be to the side of the path, so you can easily reach them.

Whenever you encounter such a problem, it’s usually wise to take the reverse approach. Don’t build a kitchen and then check if it’s walkable. Build a path and then model the kitchen around it.

<IMAGE HERE>

**Step 1:** create a random path, starting from a random location (x,y)

For this, I used the random path algorithm expressed here: <https://gamedev.stackexchange.com/questions/162915/creating-random-path-in-grid>

**Step 2:** use “extend road” on the start and end point.

This ensure we have two openings, which are reachable and connected to a road.

**Step 3:** randomly overlay the path with rectangles

Pick any cell from the random path. Now draw a rectangle around it of random size (2x2, 2x3, 3x2, …) Mark all those cells as part of the kitchen. Continue until the kitchen is large enough (cells.length > someNumber).

**(Step 4:** loop through the cells and place walls whenever an edge is connected with the outside world.)

**Step 5:** pick random cells within the kitchen, *excluding those from the random path*. Place tables and ovens there.

The random location (x,y) of the kitchen can be anywhere, although I made sure it had at least distance 10 to another kitchen, and distance 3 to the edge.

## Buildings

Knowing the previous algorithm, placing buildings is trivial:

* Place building (regular buildings can be anywhere, ingredient buildings are spaced apart as much as possible)
* “Extend road”

… that’s it. As I said: the “extend road” algorithm does a wonderful job jelling these naïve algorithms together.

## Pixel Art

In an effort to keep this game simple, I imposed a harsh restriction on myself: all sprites had to be 8x8 pixels. That’s not a lot of room.

As it turns out, buildings aren’t square, so this requirement changed to 8x11 along the way.

I must say that this restriction was a blessing. It allowed me to very quickly make art for the game, keeping everything simple and clean. At the same time, I think the game looks quite cute and fun, even though I only had very few pixels for every element.

(Some elements are more detailed, such as characters and vehicles. They are 16x16. But most are 8x8 or 8x11.)

<IMAGE HERE (animated logo or something)>

Using pixel art also makes animating much easier. In most cases, it’s a matter of smartly displacing some pixels or shifting colors, then playing a spritesheet animation. It’s a simple thing, but making everything *move* and *respond to input* makes a game feel much more alive.

It’s my first attempt at pixel art, but I like it so much that all subsequent peer-to-peer games might just be using this art style …

(Yeah, yeah, I know this game isn’t exactly a prime example of gorgeousness. I’m just trying to explain why I like pixel art and why I think some parts helped the game a lot.)

## Conclusion

It may sound very simple now, but it took me a long time to find such elegant approaches and implement them (without entering infinite loops or creating awkwardly shaped kitchens).

The kitchen algorithm was already my third try. (At which point, I must admit, my positive state of mind was quickly deteriorating.)

However, I think this does provide a wonderful introduction to procedural generation to anyone wanting to learn about that topic. Using something as simple as these building blocks can already get you very far.

I think that’s all I want to say about this game.

Play it! Have fun together! Introduce your (grand)parents or your kids!

Hopefully you learnt something from this article series, and hopefully I am able to play *your* cool peer-to-peer multiplayer game soon.

Until next time,

Pandaqi