Blunt object, meet nail

Choosing tools and wrangling Unity

About me

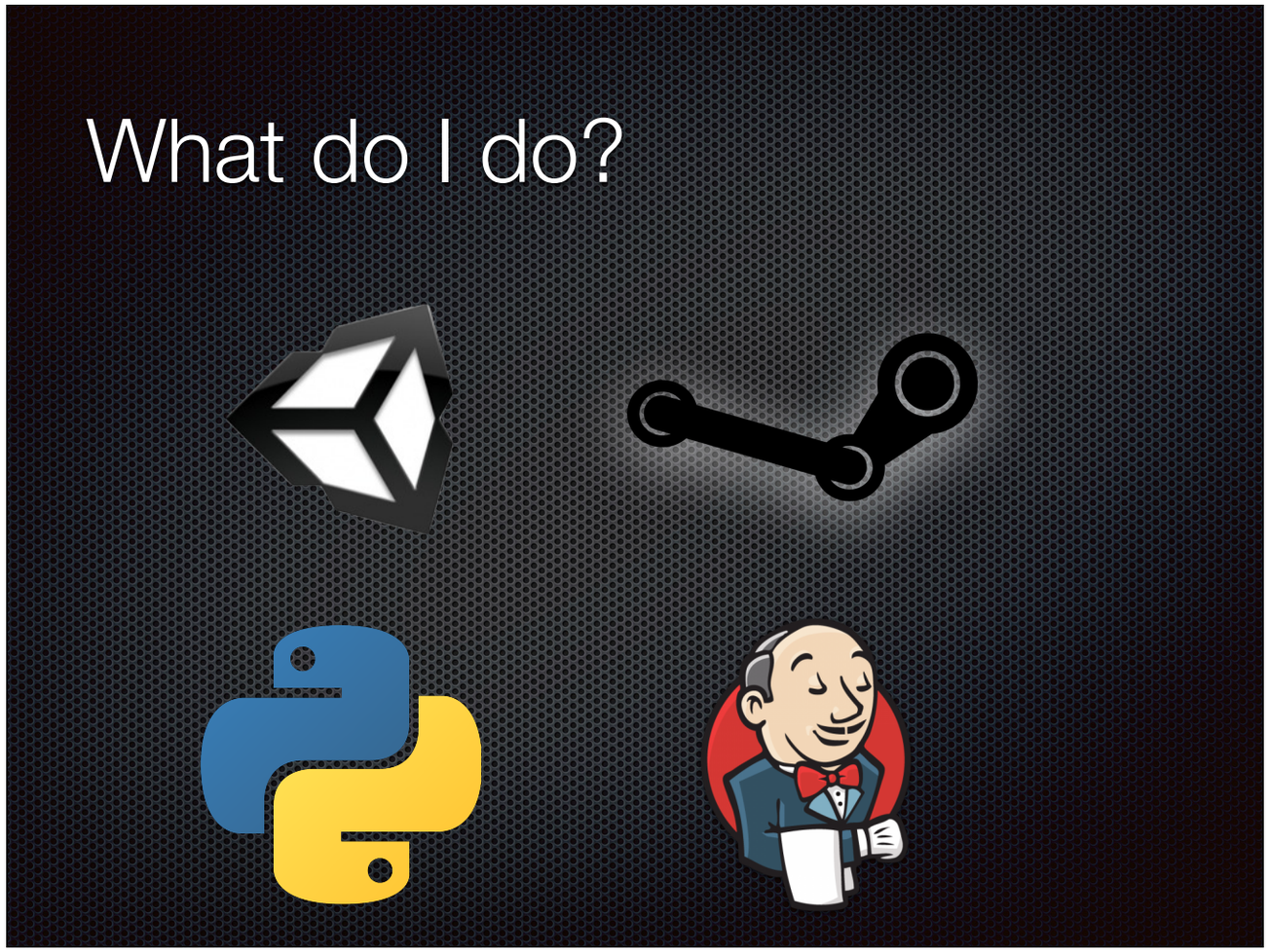
UBER ENTERTAINMENT

Norwegian, moved to the US 6 years ago for a year at UCSD, and never went back. I now work for a company called Uber Entertainment, who have created games like Monday Night Combat and Planetary Annihilation.

What will I cover today?

- What do I do & how did I get started
- Development philosophy that I find useful
- A couple of small Unity tips

- Lots of Q&A!



I mentioned I work at Uber, but what do I do there?

I was hired on to do Linux & Mac porting and general programming. These days it's more general programming than anything else.

My experience is mostly in engine programming, but I am very interested in gameplay programming and tools programming.

Right now I'm working on an unannounced project using Unity — I mostly implement systems and game mechanics, like input handling and points systems.

I also handle all of our publishing to Steam and other download sites.

I manage all of the build scripts, analytics tools, and other tidbits written in Python.

And I maintain our infrastructure like Jenkins (which does automatic build publishing, runs analytic scripts, etcetera) and Socorro (which collects crash information.)

I also have many other jobs! Phew. Working at a small company can get really hectic.

I used to play a lot of games as a kid, as I imagine many of you did too. I took an early interest in programming, but never in game programming.

It was not until I played the original Half Life that I realized I wanted to make games. Half Life really showed me what kind of emotional impact games can have on people (because it had on me), and what great creativity you can express with games.

After that I have fiddled with game development in many forms, including a game project in college, graphics programming, and other things.

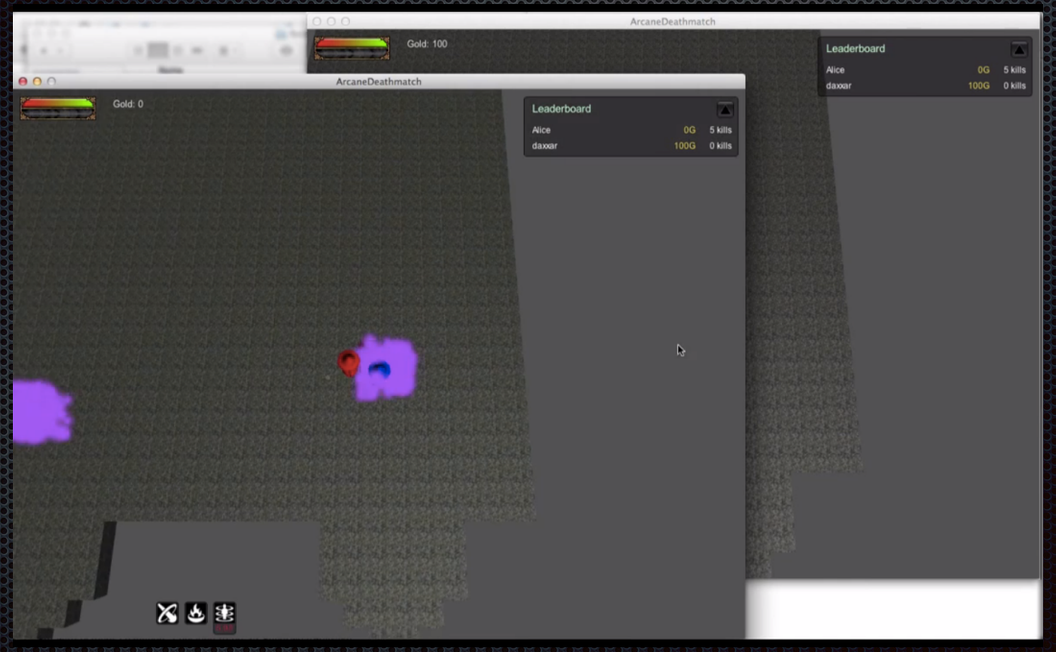
I worked at NVIDIA making graphics drivers for Linux, because graphics drivers were kinda related to video games.

Then I took another job that's not really relevant, and I took it because it paid well and had great coworkers. After a year, I started working on games in my spare time, and then later I found out that one of my former coworkers at NVIDIA had started working for a game company — so I applied there.

… And that's how I was hired by Valve. I worked on Linux & Mac OS X ports of Dota 2, Team Fortress 2, and on Steam. I have code changes in Half Life 1, and that is the most exciting thing to me.

For me, networking was the reason I got that job. I worked there for a while, and then I started at Uber Entertainment where I work now.

Just make it work



The most important part of making a game is actually making a game. It sounds silly, but as a programmer, it has been a really difficult thing to master. Since I am passionate about writing code, sometimes I obsess over code — I want everything to work “just right” and be super fast and awesome and and and …

Then I lose interest.

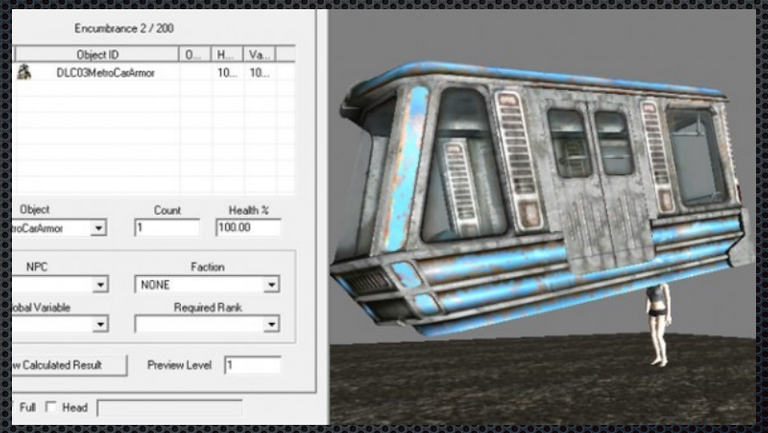
I have found that the most important part about getting a game done when you’re alone or in a small team is actually having it playable as soon as possible, and make sure you frequently do work that you can see change how the game looks, plays, etc. If I can *see* the changes I make, I find it a lot easier to stay focused and motivated.

Accept that things will be ugly or janky sometimes. Make a game, no matter how bad it starts out. It’s a lot easier to go from a “pretty bad game” to a “good game” than it is to go from “no game” to a “good game”.

Here’s a screenshot from the first game I worked on in my spare time before I joined the games industry. It looks pretty gnarly, but even at that point it was fun to play it (even if it got boring pretty quickly.)

Here’s a screenshot from the second game I worked on in that time period. It was even more fun — I watched my coworkers yell at each other as they played! It was extremely exciting. Playtests are awesome.

Both of these games were made with Unity, because Unity lets me focus on getting something working.



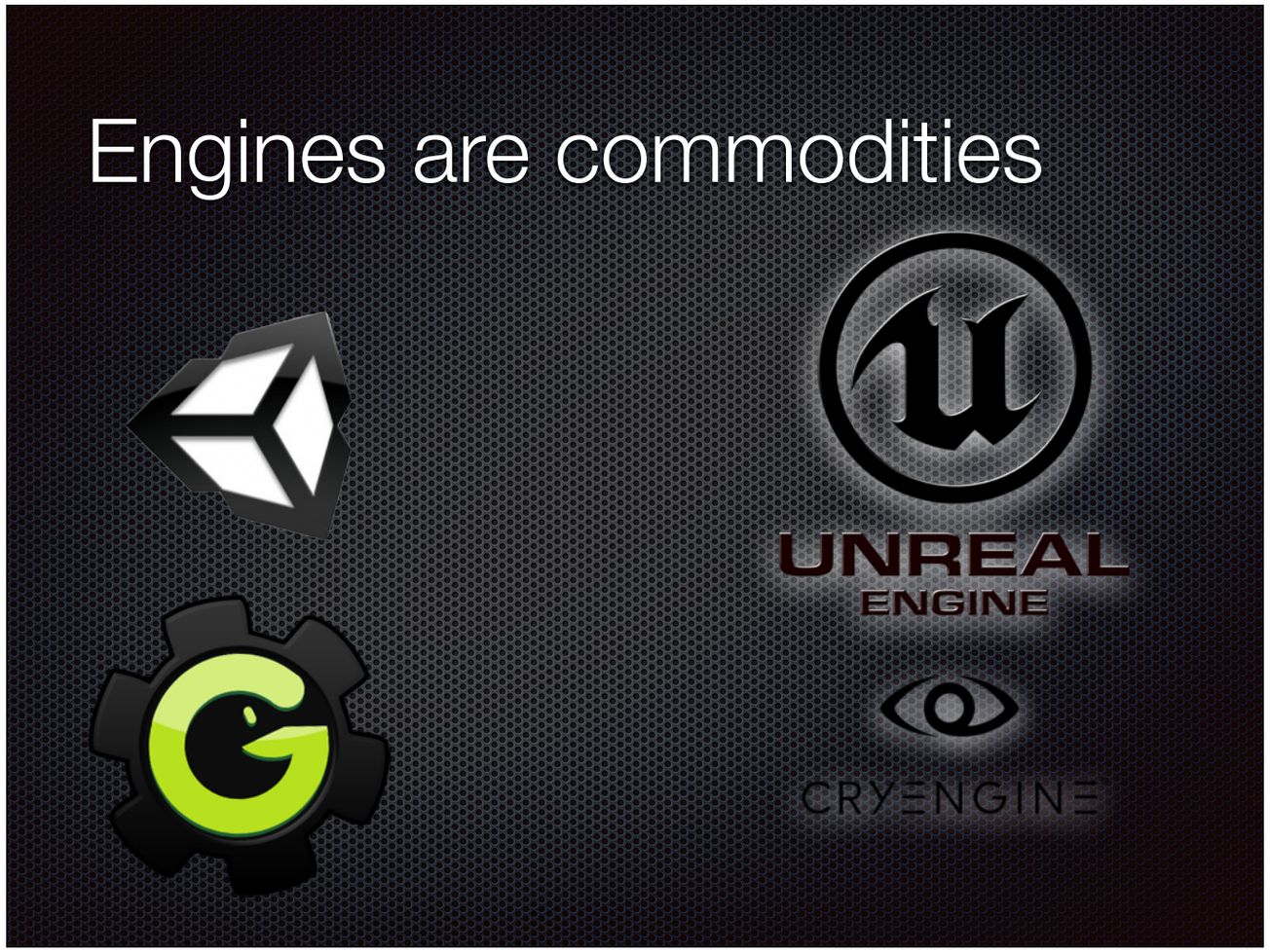
Some of you might know this screenshot — it was circulating on the ‘net a few months back. This is how the metro train was implemented in Fallout 3. It is literally a piece of armor.

It’s important to pick the right tool for a job — which is why you should never get involved in a flame war about programming languages or engine choices — but the tool you have is better than the tool you don’t. This is a hack, of course, but it solves the problem completely. Spending more time implementing a “correct” solution adds absolutely no value.

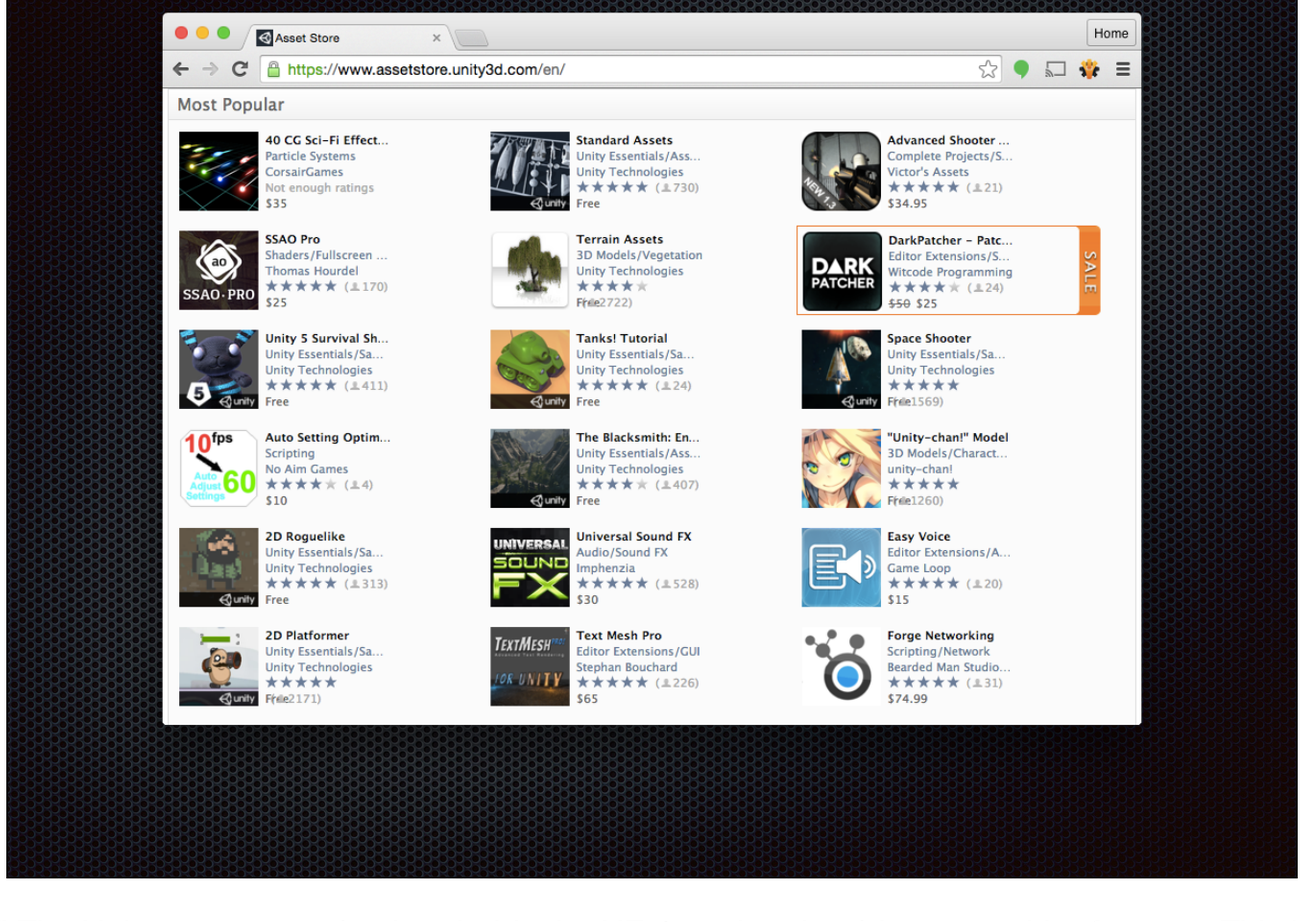
You'll hear this mantra repeated many times through your career (hopefully.)

This does not just apply to traditional optimization, but also optimizing the "code quality", and the optimizing the end result (e.g. how nice it looks, make it handle all the cases, etc.)

The key here is to make it work *first*, then make it work well. What "well" means can depend on the problem you're solving, and the context you're solving it in. If it's code that runs infrequently, you might never need to make it *fast.*



This is a relatively recent development, but now engines are cheap and plentiful. There are multiple choices for engines that literally cost you $0! I doubt any one of them can be said to be “better” than any other, but rather they all have different trade-offs (including monetization models.)



Don’t insist on doing everything yourself! The Unity asset store (and probably the UE 4 marketplace) has a wealth of assets. This includes both code and art. When you buy an asset in the asset store you get a full license to use it as much as you want, no matter how big your team is (the exception are “Editor Extensions”.) You can even modify them (many of them come with source code.)

The asset store is a big part of why Unity is so powerful — at Uber we probably have at least a dozen asset store components in use in our newest title (most of them are code.)

Your own tool is a last resort

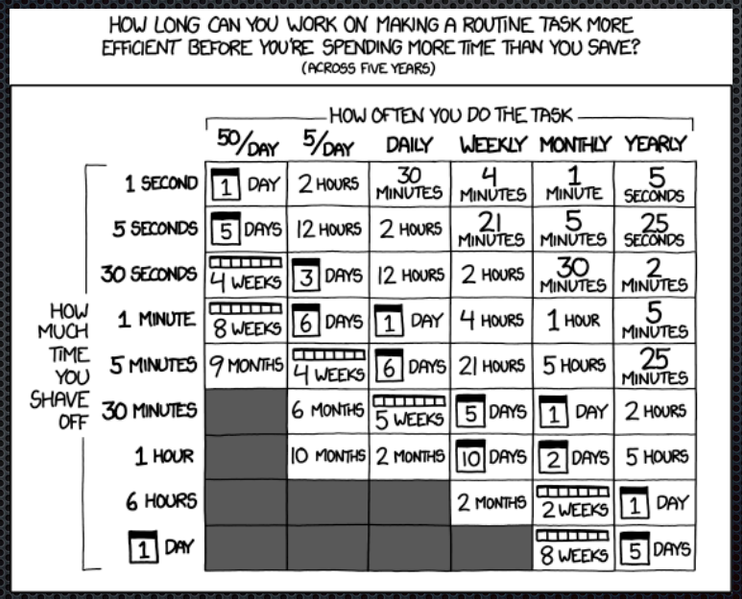
Until you know you can’t solve the problem with the tools that are available, typically don’t make your own tool. Someone who knows more about the problem space could already have solved it, so it’s worth spending a little bit of time looking around at what other people have done.

Even if they’re not giving it away or selling it — you could try reaching out and asking them if they’re willing to share it. The game dev community is very friendly.

Never wait for someone else's work, unless you're paying them

Unity *always* has a "really nice feature" on their upcoming release list. Maybe you read somewhere that some guy was working on a plugin to solve the exact problem you're having. Whatever the case is — never wait for someone else, unless you (or your employer) is paying them. It's a great excuse to procrastinate, and you might be waiting for a long time.

Tools & automation

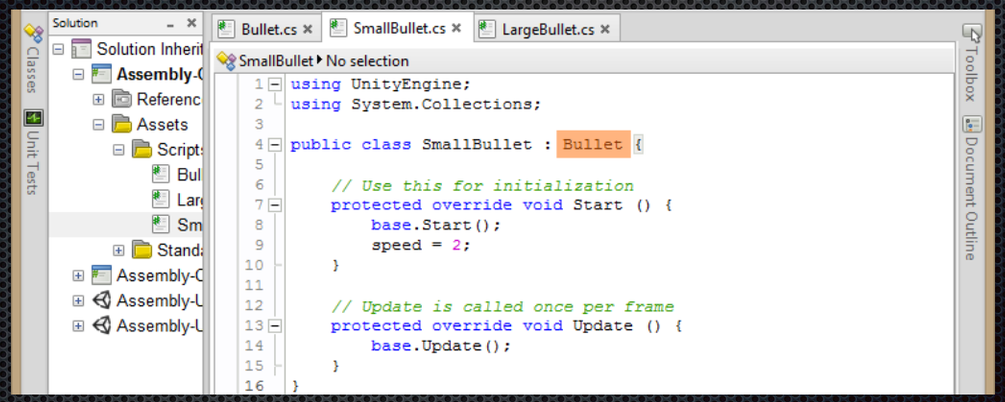


XKCD has this neat comic on how much time you can spend on something and still save time. Try to make reusable tools if you're doing something frequently — everything from publishing to development tools to verification tools. The bigger the team, the more useful tools become.

I don't know if you're familiar with ".meta" files. They're these additional files that Unity generates to track your assets if you're in version control mode. Sadly, they're very fragile, and people making the wrong change to them can waste a lot of development time. At Uber I wrote a tool that sits on our version control system (Perforce) and verifies that no-one checks in these obvious breaking changes. It has been great — it has even prevented me from accidentally checking in broken changes myself!

Another thing that's worth investing in is a quick way to build and publish your game. This is useful both for testing inside your team, as well as when you have people outside of your group testing. Being able to really quickly go from making a change to having people playing your game with the changes will add up to a whole lot of time saved really fast. There is nothing more frustrating than having people complain about a problem that you fixed an hour ago, but that is stuck in your publishing pipeline!

Use a common base class for your scripts



You can do this even if you don’t have anything to put in it. That way you don’t have to change all your scripts to do this change after the fact.

In your class, implement most of the Unity methods and delegate to the other methods.

But maybe not Update() & FixedUpdate()!

Learn the Unity behavior lifecycle: http://docs.unity3d.com/Manual/ExecutionOrder.html

Object pooling!

Performance in Unity is a bit tricky — it's easy to do a lot of things that end up getting pretty slow. One of the things is that instantiating objects (say .. bullets) can be very slow.

To solve this, a lot of people use object pools. An object pool is just a collection of objects, that instead of being destroyed, are just deactivated and hidden. When you want to create a new object, you simply pick one of the objects from the list, activate & show it in the right location.

You'll have to make sure that all the behaviors on these objects use OnEnable / OnDisable to set themselves up / shut them down, as when you use a pooled object it will not have Start / Awake / OnDestroy called when you expect it to be.

- http://docs.unity3d.com/Manual/ExecutionOrder.html
- http://jorgen.tjer.no/post/2015/10/21/experience-america-presentation/

-  @jorgenpt
- ✉ jorgenpt@gmail.com

Here’s my contact info — feel free to reach out about any questions you might have!