Pipeline Handbook

Parker Britt v0.2.0

Contents

Introduction

This document outlines general guidance for pipeline and development, specifically within the context of a final year Herts film. This is based on my experience and what I wish I knew when I started my pipeline. Some of the content I'll swear by, some of it's just a loose suggestion. None of it's gospel. Different contexts require different practices, so just take what's useful for you. Feel free to message me if you have any questions or suggestions on improving the document.

Environment

Naming Schemes

Parker Britt

When building your pipeline it's important to create standardized naming schemes for everything from directories to Maya objects and file exports. This is especially important if you plan on creating any tooling, as a single deviation can break any tool that needs to work around these schemas.

I recommend create a naming scheme bible if you haven't already. You can ask your teammates to contribute the sections relevant to them (with some oversight) if you're not as familiar with their workflow. Just make sure your team sticks to it, you can even create tools to enforce conventions. Do this early on before your group starts naming stuff.

You might also want to keep a single case convention as well such as snake_case, CamelCase, kebab-case or pascalCase .

This is especially relevant when working with USD. When exporting a model from Maya, the names and hierarchy tree of the objects in your scene will propagate to your USD primitives. So, if one of your group members is working on a version of the model called `rhino` and another renames it to `rhino` in their Maya file it's going to be a problem when layering the two files in USD. You would end up with two different models, instead of a single model that derives the textures from one and the animation from the other. This is a real example.

Pipeline Codename

Parker Britt

Come up with a short "codename" for your pipeline. This can be a shortened version of your team or film name if you want. Keep it short (ideally 2-3 letters) as you're going to be typing this a lot.

eg: `2AM`, `COG`, `HOP`

Use the name as a prefix for environment variables, `eg: COG_FPS`, tools `eg: COG_MayaExporter`, modules, APIs, whatever you want. It's up to you what you use this for but I do recommend using it at least for environment variables to avoid name collisions.

Project Directory Structure

There's lots of ways to structure your project folder. Here's how we did ours for Rebirth.

`build` - Contains build files for assets. When completed the USD and texture files are placed in `render_payload`.

I recommend putting a `main` subdirectory in each build process as seen on the right. This way you can have alternative versions/variants of the rig, model, texture, etc.

`misc` - Documents, images, general stuff that doesn't fit into the other categories.

`editorial` - Plates, film edits, etc.

`pipeline` - Tools, programs, code, anything pipeline specific goes here.

`renders` - Contains rendered EXRs, each shot should have it's own directory with subdirectories for versions. Exclude this if you're doing backups or version control.

`shots` - Where shot specific files go, like animation .mb files and FX .hip files. I recommend padding shot numbers 3 → 30 so you can add 9 extra shots without reshuffling everything.

`render_payload` - Directory exclusively containing files needed to render. Usually consisting of usd, texture, and vdb files.

Example Structure

```
job
├── asset_build
│   ├── props
│   ├── fx
│   ├── environments
│   └── characters
│       └── rhino
│           ├── cfx
│           ├── groom
│           ├── look
│           ├── model
│           ├── rig
│           └── texture
│               └── main
│               └── battle_worn
├── editorial
├── misc
├── pipeline
│   ├── tools
│   └── packages
├── renders
├── sandbox
│   ├── francis
│   └── parker
├── shots
│   └── SH0010
│       ├── anim
│       ├── comp
│       ├── fx
│       ├── plates
│       └── scene.hip
└── render_payload
    ├── assets
    │   └── rhino
    │       ├── textures
    │       ├── asset.usd
    │       ├── asset_data.json
    │       ├── geo.usd
    │       ├── mtl.usd
    │       └── thumbnail.png
    └── shots
        └── SH0040
            └── scene_layers
                ├── anims
                │   └── rhino.usd
                ├── cfx
                │   └── rhino.usd
                ├── fx
                ├── render_targets
                │   ├── main.usd
                │   └── rhino_holdout.usd
                ├── lights.usd
                ├── cameras
                │   └── render_camera.usd
                └── scene.usd
```

By storing everything needed to render in a single place you're able to easily render across different machines/farms using just this one directory with no missing files and no extra bloat.

While textures are also stored in the `/assets` directory, they should be symlinked, copied, or converted (e.g. rat or mipmapped exrs) by your asset publishing tools into the `render_payload` directory.

The `scene.usd` file contains the finished scene without render settings. `render_targets` references `scene.usd` and adds render settings and render specific overrides such as holdouts or lighting changes. These are the entrypoints to your scene for rendering and can be directly passed to husk.

I talk about render targets a bit more in the USD Referencing section.

`sandbox` - A flexible directory for team members to test and do RND without structure. Each member should have their own subdirectory.

Backup

Parker Britt

I recommend keeping a backup of your team's work. Depending on how into networking, servers, sysadmin type of stuff you are this may be more or less applicable to you. The benefit's always there but some people just can't be bothered and that's okay. But personally I don't trust the university's network drive (for good reason). There's lots of options for this. The simplest would probably be a b2 bucket and a periodic rsync script. If you use a remote versioning software like Perforce you already have a backup and can skip this step. You can also host your own backup.

Version Control

Parker Britt

Version control is the process of keeping and managing new versions of your files for every notable change. This is almost the same concept as source control which only applies to source code while version control applies to any type of file.

I have a separate section talking about git and source control.

Version control can manage your textures, models, scene files, and anything else in your project. This can really help get you out of a jam and having the experience can also sound pretty good on a CV.

I used a self hosted Perforce instance and built our custom tooling (like exporters) around that. Harry took the homemade approach, storing .usd files in versioned directories and reading and writing to them using custom HDAs that use the correct folders. Both are great options.

If you have the time and interest, adding version control to your project can be pretty useful.

Environment Variables

Parker Britt

Environment variables are a very important tool for pipelines and computing in general and are used extensively in industry. Make sure to take advantage of them. Check out the Application Launcher section for a common use case.

Environment variables are similar in concept to variables in math or programming. They have a name and a state. What makes them unique is they operate across the entire system. Any program can read these variables and use it to inform their behavior.

Even the operating system relies on these, for example the `$PATH` environment variable tells the system where applications are located, allowing them to be executed directly on the command line, without a full path.

Environment variables can be referenced with a `$` before the name in Linux e.g. `$PATH`. Since most DCCs follow Linux conventions, you'll probably have seen this with Houdini's scoped environment variables before e.g. `$HIP`.

On Windows you surround the name with `%` like `%PATH%`. Try it out by typing %USERPROFILE% into the file explorer path.

On Windows, environment variables can be set system wide (they can on Linux too but it's a little different) but only if you have administrator access which, meaning you won't be able to set them system wide on school computers. This is fine, however, as you'll likely want to use the more Linux like approach of setting them per shell/terminal. When processes are created they inherit environment variables from their parent process.

This means if you set the environment variable `FOO="bar"` in a shell, then open Houdini from that shell you can type `$FOO` in any node parameter and get the value.

Environment variables should conventionally be all caps with underscores separating each word `e.g. FOO_BAR`

Portable Applications

Parker Britt

Portable applications are programs that don't require a permanent installation, and therefore administrator privileges to be able to use them. This is a huge advantage on locked down systems like in a university or VFX studio. There are a fair number of these available on windows, portableapps.com has a bunch in one place. A lot of the time the software will just natively offer a portable version, you just might not have noticed it before if you weren't looking, I've definitely been surprised by this before.

On linux you can also install Flatpaks with the –user flag without needing root access. On top of that, most open source software just has portable zip files in their Github releases.

Development

Separation of logic & interface

Parker Britt

When developing software separating logic and UI can be an easy way to avoid spaghetti code. It's usually a bad idea to store state in your UI, instead have your UI read data from a class that stores the state, often referred to as a model. For large and complex data storage use a database.

Some find the MVC (model-view-controller) paradigm useful for this. The view code creates a non-function, but visually complete interface. The controller code connects the logic, and the model stores the data. Just like mentioned above the model can be a data class or database.

Using this paradigm can definitely help keep your interface code clean, especially for interfaces with a lot of logic.

Unique State

Parker Britt

Avoid keeping the same state in two separate locations to whatever extent you can. This means keeping your shot data all in one place and have everything read from there.

> "Give someone state and they'll have a bug one day, but teach them how to represent state in two separate locations that have to be kept in sync and they'll have bugs for a lifetime" -ryg

Directory Structure

Parker Britt

Directory structure when building programs can be a surprisingly tricky thing to get right. This will vary depending on what language you're using and what type of project it is, but there are some common practices across most projects. Generally the top level directory is reserved for configuration: CMakeLists, .gitignore, .git, README. Then you have your directories. The most important being you `/src` directory. Some people prefer using the project name instead of src, but I prefer `/src` for consistency and quickly navigating new projects.

For compiled projects you'll have a build folder.

An `/assets/` or `/static/` directory can be used to store images, fonts, icons, etc.

For python projects specifically you'll generally want a pyproject.toml on the top level, then src/package_name to contain your package.

You'll want to add any generated or irrelevant patterns to your .gitignore. This included `/build` for compiling languages and `/dist` for python.

Build System

Parker Britt

Make sure to package your projects with some kind of build system so you can easily distribute and install them with pip. (I don't know exactly how annoying they've made it to install python packages on uni computers this year, but where there's a will there's a way.)

Data Flow

Parker Britt

Avoid passing data back and forth between parent and children. In general, children should not know about their parents, i.e. class foo holds a reference to class bar that holds a reference to class foo. This gets confusing pretty quick and is often the sign of messy code that needs refactoring. I know Qt often passes parent widgets to their children, but this is an exception. Instead, signals can be a good solution, notifying parents of a change without actually storing the parent as a member variable.

Use Classes

Parker Britt

Use classes instead of holding everything in complex dictionaries or lists. Obviously still use dictionaries when appropriate but it's easy to sometimes use them to make a "poor mans class". eg: `myShot = {"number": 10, fps": 24, "description":"foo", etc.}`. When using a class you have a lot more power to protect and handle data with getters and setters. For example you could have 10 different formats for representing shot number, just store the shot number once, `number=10` then return it however you want `getFormattedShotNumber(padding:4)=="SH0010"`. You can change the implementation later as well, for example if you wanted to transition to using a database to store your shot data then you could change the class definition to read from the database and leave the rest of the code the same.

Getters and Setters

Parker Britt

Getters and setters are very important for transforming/validating data going in and out of a class. Even for simple variables, it's good practice to only allow private variables to be interacted with through a getter/setter.

Unit Tests

Parker Britt

A unit test is additional code that tests the smallest units of your source code. Rather than testing the whole thing at once like a computer would, a unit test usually runs individual functions. So rather than testing the calculator function, you would first test the add and subtract functions.

Usually you'll use a testing library for this like Pytest in python or Catch2 in C++.

Tests are a development tool and should live separately to your source code and not be included in releases. So while your source code is in `/src` your tests should be in `/tests`.

While it can sometimes be tedious or even slow to implement tests for every part of your code, it's an important practice for large codebases and improves the stability of your code.

When writing code built to be run inside a DCC it can be particularly hard to test your code in isolation. How can you test your `create_camera()` function if it runs `hou.node()`? While difficult it can still be necessary at times and always encouraged when outside of a DCC.

Depending on the company there's a good chance you'll be required to write unit tests for your code.

Test Driven Development

Parker Britt

Test Driven Development or TDD for short is a development practice where every step of development is linked to a unit test.

Generally before writing any code at all, you should first make a test. Once the test fails, write code and rerun your tests until it passes. As soon as you pass you must write another test before writing more source code.

e.g.

```
test add()
test fails because add() doesn't exist
implement add()
test add() with negative numbers
test failed because negative numbers aren't handled
implement negative numbers

test subtract()
test fails because subtract() doesn't exist
...
```

While most VFX companies won't require TDD, some software development companies will.

Programming Tools

Git

Parker Britt

Manage your code projects with git, and ideally push them to a remote git repository like Github, Gitlab, Gitea or any of the others. It's very helpful to have a history you can revert to or look back on for reference, and storing them remotely is a lot safer if your computer dies. You'll definitely be using git a lot in any programming related career so it's good experience to have. It also helps motivate you to make stuff a little cleaner because your code is public. Code reviews and collaborating are easier. When applying for jobs you can send them a link to see your work. (some application forms will have a specific spot to put your github link)

Regex

Parker Britt

Regex is a super handy skill to have under your belt. If you have the time or are doing any parsing related project I'd definitely look into it. It looks like complete gibberish, and may seem really complicated at first, but it's deceptively easy, after just a day or two of practicing you can become very familiar with how it works. I recommend starting with a website like regex101.com to learn and practice. Especially when utilizing capturing groups Regex is a super powerful tool for parsing files or paths and extracting information.

Hashes

Parker Britt

Hashes are an amazing programming tool with many applications. Specifically they're a mathematical way to store arbitrarily long files in a short, finite string. You can think of it like an ID that will basically always be unqiue. It sounds like magic. Imagine comparing two files. You could go line by line and checking differences, but this is slow, especially when comparing many files against many other, an On^2 operation. Instead precompute a hash for every file (you can store these for later) which is just On, then do the On^2 comparison against short strings.

I used these in one of my asset processing HDAs to check if the file had changed since the last time it baked. For every bake it would compute the ~10 digit hash string and store it in a hidden parameter on the HDA. Then to compare if the file had changed it would just have to recompute the hash and compare against the one stored in the HDA parameter, allowing it to work across restarts.

Qt

Parker Britt

Qt is the industry standard UI framework for VFX. It's extremely powerful and what's used to make and extend programs like Houdini, Maya, Shotgrid, Usdview, Perforce, and most other applications in VFX. Because of this, understanding how to build UIs in Qt is an important skill to have.

There are several ways to build Qt interface including the C++ API, Python bindings (PySide and PyQt), QML, and Qt designer. I highly recommend using the APIs and avoiding Qt Designer.

If you're unsure whether to use PySide or PyQt, I would recommend PySide as it's widely supported and officially developed by the Qt Company.

Implementing UI with code is slow, but designing a UI with code is even slower. That's why I always recommend doing the actual design first, using interface design software like Figma, Penpot, Photoshop or whatever you prefer. Nail down the look and experience before writing the code.

Qt Resource Compiler

Parker Britt

Make sure you use Qt's rcc system for static assets (fonts, images, etc.) on Qt projects. It drastically simplifies installation and management of dependent files.

Houdini & Rendering

Animated Keyframes

Parker Britt

Animated/keyframed textures are a pain in Solaris/karma, but they are possible. You have to check the "allow shader parameter animation" parameter on the material library node. Keyframes must have a starting keyframe on your timeline's first frame. Changes may not reflect in the preview material or even when rendering in the viewport (especially displacement). To test if it's working use mplay or render to disk.

Solaris Viewport Issues

Parker Britt

On that note. The render viewport lies all the time, whenever you have a weird render issue, always confirm if the render is working with mplay/ render to disk. This is especially common with displacement (karma doesn't re-slice the model when moving the viewport camera by default, can be turned on in the settings but it's pretty slow)

USD & File IO

USD Referencing

Parker Britt

USD is built on the concept of referencing and layering those references, if you're putting everything in one file you're not taking full advantage of what USD has to offer. An asset can be split into `model.usd`, `payload.usd`, `mtl.usd`, `entry.usd`. All these files are layered together. If a shot needs a damaged version of the model or textures, swap out `model.usd` or `mtl.usd`. Each asset in each shot should have its own animation and CFX usd files. The animation is used in the scene as a placeholder and passed to the CFX artist. Once the CFX is finished you can use it to replace the animation out.

Using this system you can export several different versions of your scene all using the same references with just slight changes on top. This is extremely useful in production. For a single scene you might need several renders with different holdouts, lighting, materials, render settings, etc.

For example some on Rebirth some of the transformation shots needed different versions of the rhino texture that the compositor could fade between. All it took was a single material bind on top of the base scene.usd.

As another example, you might want to render volumes with a CPU engine and the rest with an XPU engine. Or have more samples in one holdout than the rest of the shot.

We used this a ton on Rebirth and it was extremely useful. Message me if you want to know more.

USD Layering

Parker Britt

USD is designed so that different disciplines can author files independently, only containing their contributions to the scene. The lighting artist doesn't export the entire scene, just the lights, and the animator doesn't export the entire character, just the motion. Or at least in theory.

The Layer Break LOP allows you to use this feature of USD within Houdini. With layer breaks you're able to separate where the previous work ends and where your work begins. You're able to modify and build on the previous work, but when exported the file will only contain the changes you made. It's only when you layer the original file and your new one that you see the full picture again.

Lets look at how you should use this concept with characters. A finished character animation will be made up of a bunch of parts: point positions, vertex connectivity, normals, UVs, materials, material bindings, and much more.

A simplified character USD pipeline will look something like this: `model → lookdev → scene` while `model → rigging → animation → cfx → scene`. The two branches can be performed in parallel and are then combined at the end in the final scene. The first branch only happens once for each character where the animation/cfx step happens for every shot.

`model` - Exported from Maya as USD, the file describes UVs, point positions, vertex connectivity and normals.

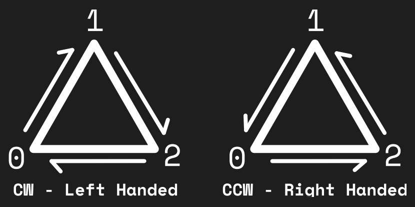
`lookdev` - The model is referenced and layer breaked in Houdini before adding anything. Your lookdev USD file should only contain the materials and their bindings, not even a geometry file reference. This way you can mix and match different versions of your model with the same texture.

`animation` - The rigged Maya file is animated then exported out to a USD file. This file, to the best of your abilities, should only describe the movement, and properties that change with the movement like normals. It should have no duplicate information like UVs, material bindings, etc. that were described in the model or lookdev steps. Ideally it wouldn't even contain connectivity information, but that's not possible with Maya's current USD exporter.

`cfx` - The animation is imported here, deformed then exported. Same idea as the animation, it should only contain the changes made, not re-describing what was already defined in previous files.

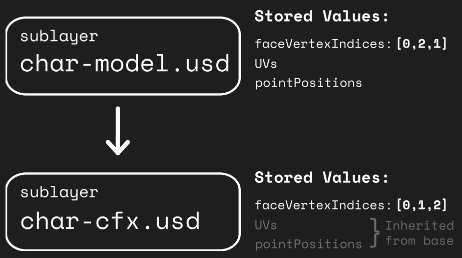
Vertex Winding Order

Parker Britt



As you likely know a model is ultimately made up of vertices in space. These vertices are then told how to connect to each other to create faces. Given 3 vertices, connecting them in a clockwise order `0 → 1 → 2` will create a triangle. But now imagine we order them counter-clockwise, `0 → 2 → 1`. This will also create a triangle. They look identical but this difference can create big problems, specifically when layering in USD. Different programs use different winding orders, CW (clockwise) or CCW (counter-clockwise).

Here's one of the many examples of how you might encounter this issue. Imagine a USD pipeline where a character model is created and UV mapped in Maya (CCW) then exported to a USD file. You then send you model to your CFX artist who imports it into Houdini. Here's where things go wrong, because without any indication, the import has completely inverted the geometry's winding order to match Houdini's (CW).



Now the CFX artist exports their movement and the two files are both sublayered on top of each other in the scene file. You'll immediately see something odd. The texture seems to be rotated differently around every face. This is because each face has been rotated CW from where the UVs were created CCW. When we layered the two models not only were the point positions overwritten, but also the faceVertexIndices property.

The ideal solution would be to exclude connectivity information from the CFX file, only storing position data. Since neither Houdini or Maya's exporters have this functionality built in, we have to find another way. Ultimately it's important to keep the winding order consistent between exports.

Maya's exporter isn't great and doesn't include the ability to reverse winding order. I started working on my own exporter to allow clean exports directly out of Maya, but until I got back and finish it we'll (if I ever do) we'll need another solution.

One option is to export your models and animations in a temporary USD file out of Maya, then use the USD API to copy just the parts you want over to a new file, modifying or removing the undesirable parts.

Another option is to just stick with Maya's CCW winding order. While Maya's exporter can't reverse winding order, Houdini's can. You can find this toggle under `Geometry Handling → Reverse Polygon Vertex Ordering` on the SOP Import LOP (and therefore USD Export SOP).

DCC Tools

Railroading

Parker Britt

Railroading users is a big part of the job, meaning building tools to stop them from doing the wrong thing by accident. Everything in a pipeline is so delicate and most artists won't know what they're doing. One incorrect export setting can lead to hours debugging the scene. Building tools that railroad users and prevents them making these mistakes can save you a lot of anguish. These tools provide abstraction to the artists, but with abstraction usually comes with a loss of control. Eventually you'll run into a situation your tool doesn't account for. The solution to this is adding in overrides, hidden advanced options that allow you to make your tools work for uncommon, one off, situations.

How you add them depends on what you're making but most commonly you'll need these in Houdini, you can use an "advanced" parameter group and dive in targets for most things.

Headless instances

Parker Britt

Most DCCs have a python interpreter that allows you to run commands in the DCC headlessly (without a gui). This can be super powerful for processing data across programs. Imagine, you could edit a Maya file directly from Houdini, and edit Houdini files from Maya. Anything you normally can do with the API you can do headlessly. It's much quicker and easier than opening a full DCC and executing a script because it doesn't need to load the entire program. And your users won't even realize it was ever used.

As an example you could generate a thumbnail for each shot or asset from one of your tools by opening a headless instance of Maya or Houdini and rendering a low res openGL frame for each shot.

The interpreter executable will usually live in the program's bin (binary) folder. For example Maya's mayapy and Houdini's hython will allow you to start headless sessions. For Maya, you'll need to use a standalone initializer.

Show Launcher

Parker Britt

Create a simple (or complex if you want) wrapper for launching applications that sets up environment variables. These environment variables can be read from a file or database. Some of the variables will be universal to the project (FILM_NAME) and some will be more specific (SHOT_NUMBER, ASSET_NAME). The wrapper can be a full gui, tui, or cli. The important part is all your tools, specifically the tools inside DCCs, can now read these environment variables.

For example your render HDA knows to render files to `` `$RENDER_EXPORT_PATH/$SHOT_NAME.exr` `` which will evaluate differently depending on what shot you opened from the launcher. You might have 30 parameters all reading `$SHOT_NAME`, imaging changing every single one of those manually. All the information is coming from the same place, your application launcher and wherever that's reading from. So by changing it there it will update everywhere.

Importantly, set a `PROJECT_ROOT` enviornment variable pointing to root of your project directory structure. Use this in all your DCCs and code instead of writing it manually. This can be easier in some cases but is also very important for portability, allowing your team to do work on their own computers if they need to.

Optionally, if you have time, you could make a tool to check that there are no hardcoded paths and replace them with PROJECT_ROOT, as your team will likely forget.

Servers and Self Hosting

Introduction

Parker Britt

While servers are not at all necessary for a good student pipeline, neither is half of the stuff in this document. That's why most teams are able to make fantastic short films with truthfully awful pipelines. But the point of University isn't to make a film, it's to learn as much as you can and prepare yourself for industry, and in industry it pays to know this stuff. But more importantly (since you're reading this) there's a good chance you're nerdy enough that managing your own server sounds cool. And you'd be absolutely right, servers are cool as hell.

What Is A Sever?

Parker Britt

When most people think of a server they think of massive data-center with server racks the size of fridges, but a server can also be your old desktop, a Raspberry Pi, or a smart fridge (but don't do that). Any computer that acts like a server is a server. Usually servers are always on, they also usually serve network requests, either locally or over the internet.

Aquiring A Sever - VPS

Parker Britt

There's two paths to getting your hands on a server.

The first is to rent one online through a VPS (virtual private server) provider. It's called a virtual private server because the system is virtually partitioned to share the cpu cores between multiple users, it's private so no other user can interact with your server, and it's a server (obviously). There's many services that provide this service, usually starting at about £5 a month and going up depending on how much storage, memory, and cpu cores you need. The biggest price sink is surprisingly storage. Storage drives have gotten crazy cheap in the last 10 years, but companies still charge crazy prices for VPS and s3 bucket storage. The price of VPS storage is about the same as if you bought a hard drive of the same size, only you have to pay this amount every single month, forever.

I'm not a big fan of VPSs for a couple reasons. They're often terrible value compared to the equivalent owned hardware (especially if you know how to get a deal), you miss out on a lot of the valuable learning that comes with setting up your own, they're less versatile, and owning and having control over your stuff is cool.

That said there are some benefits to using a VPS. They're easier to scale up or down depending on your needs at a time, you don't have to worry about unexpected downtime as much (power outages, internet outages, moving house), VPS providers may have faster connections than your internet plan provides, networking is easier if you don't have access to the router where you live (uni accommodation), and you don't have to figure out where to store it or how to connect it to your router.

Aquiring A Sever - Self Hosting

Parker Britt

The Second option is to own your hardware. This is the solution I would suggest. If you have an old desktop or laptop lying around then your're sorted and can just skip this section.

If not don't worry, while it may seem expensive to "buy a server" it's surprisingly cheap and will usually pay itself off in less than 3 months compared to the equivalent VPS. What you want to look for is second hand, low performance SFF (small form factor) pc. Companies will shed these things like crazy as they're only just powerful enough to run Windows. With the recent (as of the time of writing) end of Windows 10 support you should be able to find even better deals on perfectly good hardware that doesn't support Windows 11. Our advantage here is that a headless (no gui) Linux installation can run amazingly on even the lowest end hardware. Trust me, these servers are more than enough for our use.

These type of office SFF PCs aren't quite as upgradable as full ATX hardware, but can serve you well for years to come depending on your needs.

There's lots of places to find these computers and a lot of models to choose from but a good starting place is looking at Dell Optiplex listings on ebay. On the lower end you can find a perfectly capable Optiplex 3020s for less than £30. That's half the price of a modern Raspberry Pi and significantly more capable!

Just to really drive home my point, after scrolling for about five seconds I found a Dell Optiplex 3020 8GB (from a buyer I trust) for £27. It has an i5-4570, which is about twice as powerful as what's on the Rasbperry PI 5. The cheapest UK approved reseller lists the 8GB model at £59.54. So, like I said half the price, twice the performance. It's also worth mentioning that the PI's use of the ARM instruction set architecture makes them a suboptimal choice for a server, as it drastically reduces software compatibility compared to standard x86-64 CPUs.

Operating System

Parker Britt

While you can choose whatever you want there are some better and worse options. Most of these aren't technically operating systems in themselves, but that's not important for what we're talking about.

Linux - Bare Linux is simple and flexible, paired with Docker or Podman it's a great starting point and you'll learn a lot in the process of setting it up. This is definitely what I would recommend choosing. It's the dominant server OS, open-source, and what's used to run the top 500 fastest supercomputers.

Check out Linux Distros for the next steps.

TrueNas - Based on FreeBSD, TrueNas makes a fantastic choice for storage heavy servers thanks to ZFS's maturity and feature set. It's virtual machine and docker support make it more than just NAS software, it's capable for servers as well. While there are paid versions these are only relevant to companies, so it's basically free. Once installed it provides a web UI to manage everything, but keep in mind this doesn't make it easier than a vanilla Linux installation. This is because using TrueNas requires you to become familiar with many ZFS concepts such as pools, vdevs, SLOG, ZIL, RAIDZ, ARC, etc. I would definitely recommend this OS if you're focused on storage or using more than about 3 drives.

Proxmox - Based on Debian Linux, Proxmox is a virtual machine manager. It's similar to TrueNas in that it provides a web ui, is technically open source, and has paid-versions that you won't need.

Windows - Please don't. Funny enough this is what Herts use(d) for their render farm.

Choosing a Linux Distro

Parker Britt

Because of it's open nature the Linux "OS" is incredibly modular. Despite being used to refer to the entire OS, Linux is technically just the kernel, an important but incomplete part of the operating system, you'll need to choose a specific distribution that contains all the other modular bits and pieces that make up a complete operating system.

There's hundreds of distros for every use case. The one's we want to look at are very stable and Enterprise focused, meaning they update very infrequently and use old versions of software that are know to be secure and work well. These are what companies choose to use for their servers and workstations. To keep it simple I'd recommend either Rocky, Ubuntu Server, or Debian. You can't really go wrong here, but if you really want a suggestion I'd go with Ubuntu Server. I currently use Debian on my home server, but have had a good experience with all three.

Because Linux is so modular you can choose whether it comes with a desktop environment or not. A desktop environment is what most people think of when they hear the words operating system. It's the GUI and all the apps that come installed by default: the calculator, settings menu, wallpaper, task bar, file explorer. These are all things built on top of Linux and for our purposes are just unnecessary overhead.

If you're planning on going this route and are confused, just message me and I'd by happy to walk you through it or point you in the right direction.

Use Cases - School

Parker Britt

There's an endless amount of projects you could use a server for in your final year. You could build a custom render farm, version control, backup, discord bots, web apps, etc.

Use Cases - Personal

Parker Britt

Beyond University there's a ton of videos online showing off what you can do with a home server, but to name a few self hosted services to get started with you can look into:

vaultwarden, nextcloud, immich, wg-easy, portfolio websites, pihole, gitea, jenkins, plex/jellyfin, searxng, mumble, matrix, excalidraw, exo, copyparty, home assistant