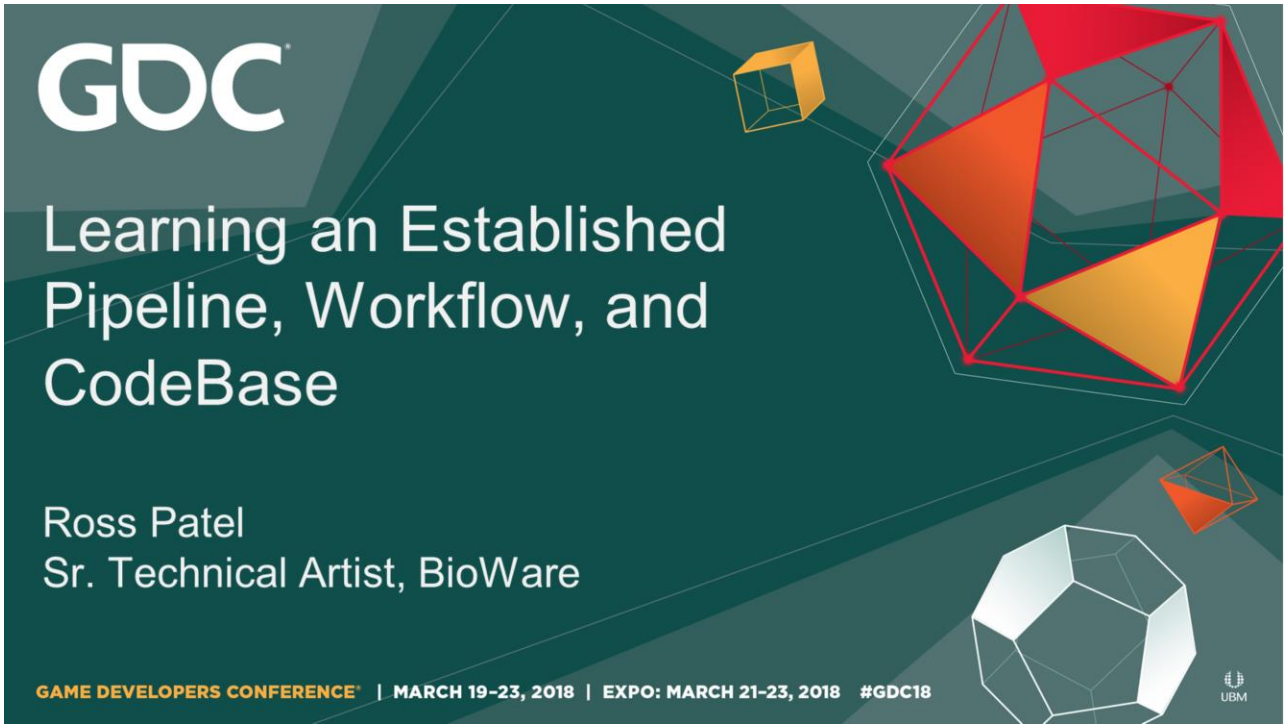

GDC
Learning an Established Pipeline, Workflow, and CodeBase
Ross Patel
Sr. Technical Artist, BioWare
GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18
UBM

Ross Patel

I've worked on 11 projects, 7 as a technical artist.

My current project is Anthem.

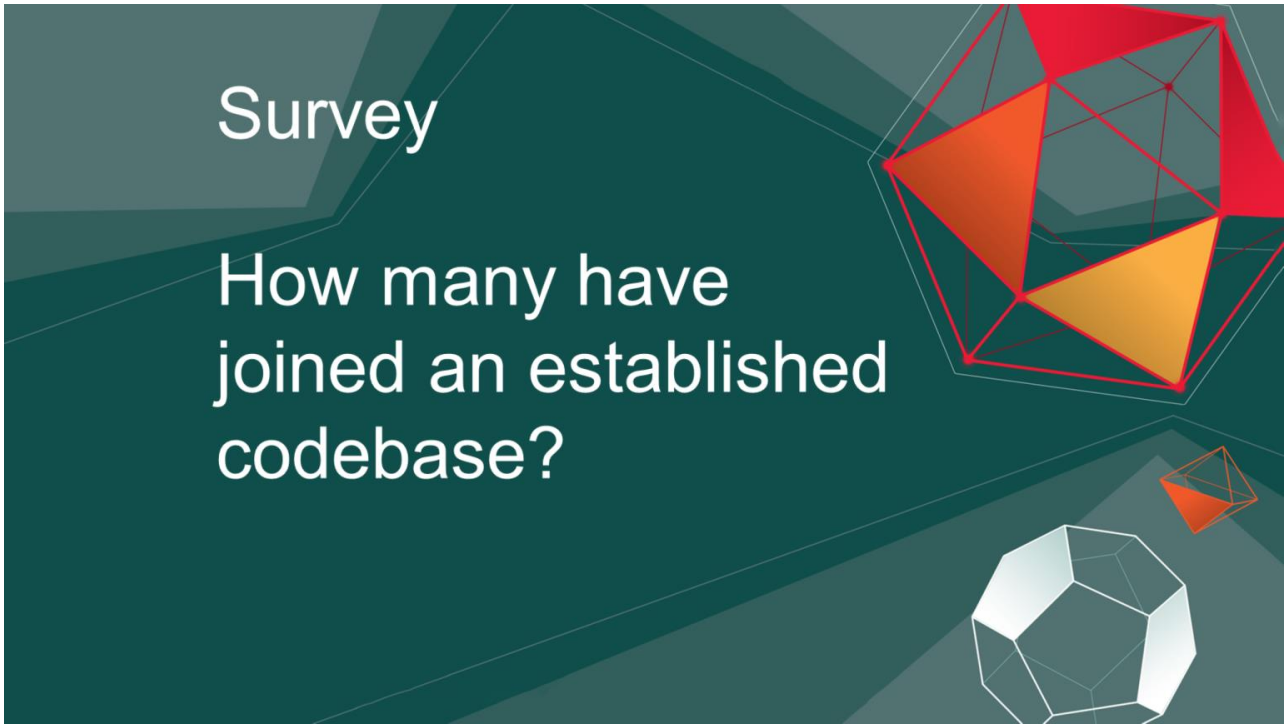
I've been on:

Huge efforts that required tons of tech-artists and tools engineers.

I’ve also been the lone technical artist on a locked down pipeline.

The reason I wanted to do a talk on this subject is that I’ve onboarded properly and had a great experience, and I’ve also onboarded poorly and struggled.

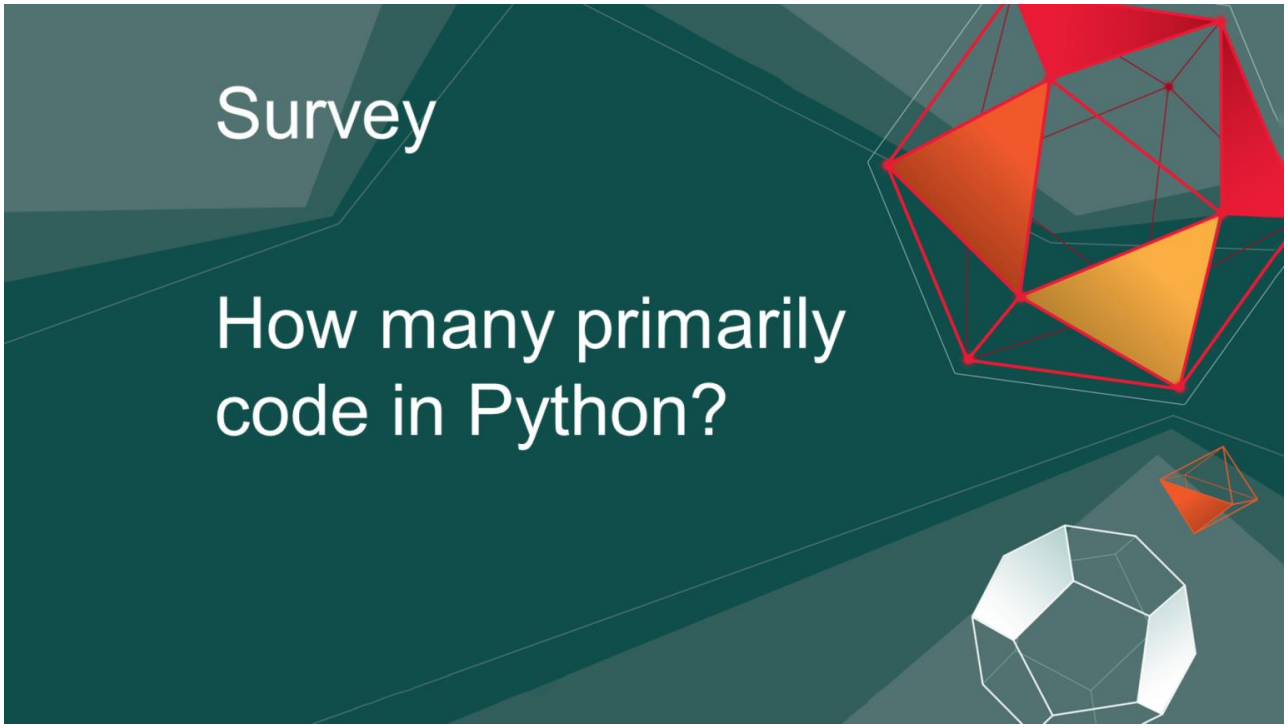
This talk will expose the things that have helped me the most.



How many of us primarily work on a pipeline where the primary language is

- Python?
- C#
- C++
- How many of you support pipelines with two languages.
- More than 2?
- Languages I haven’t mentioned?
- Transition to how this pertains to

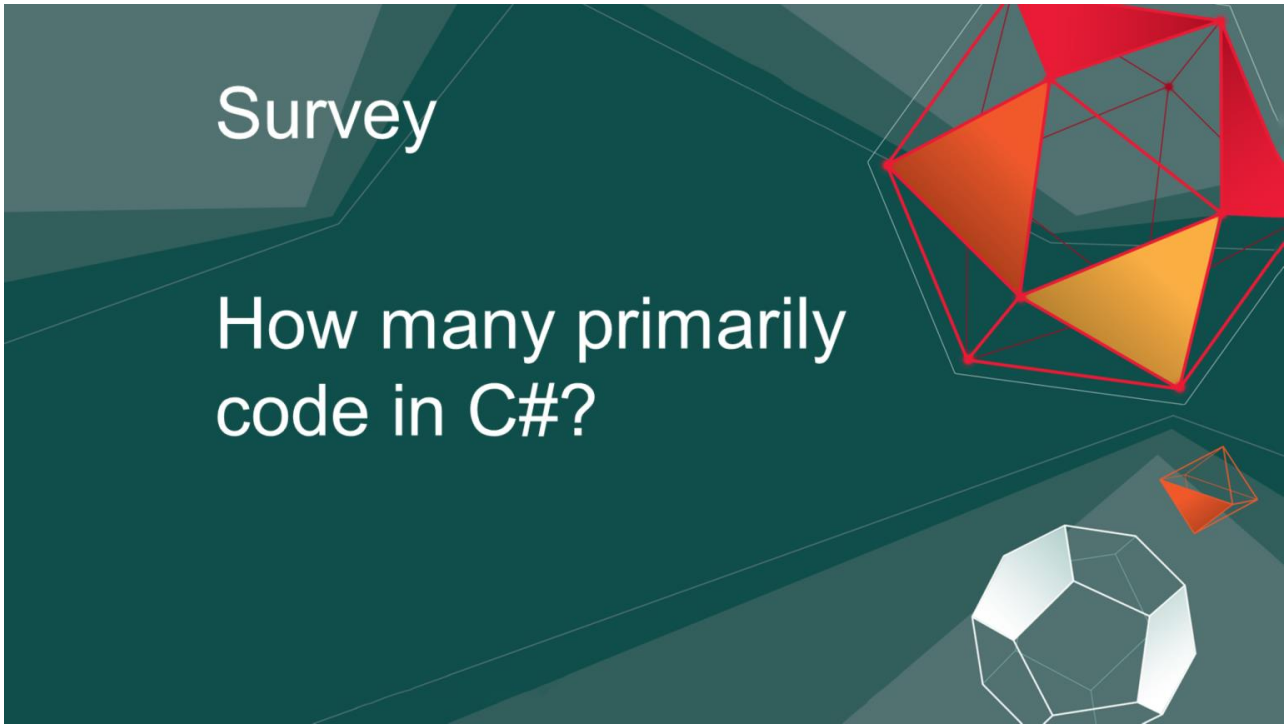
scope.



How many of us primarily work on a pipeline where the primary language is

- Python?
- C#
- C++
- How many of you support pipelines with two languages.
- More than 2?
- Languages I haven’t mentioned?
- Transition to how this pertains to

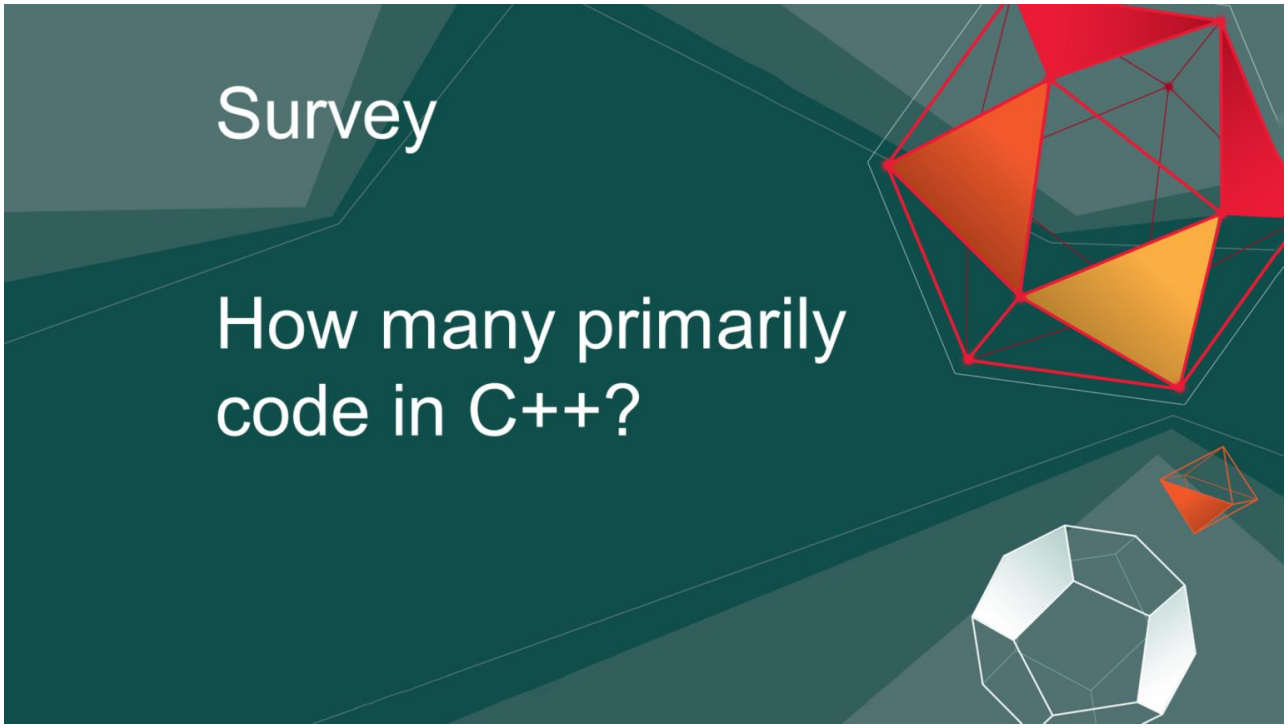
scope.



How many of us primarily work on a pipeline where the primary language is

- Python?
- C#
- C++
- How many of you support pipelines with two languages.
- More than 2?
- Languages I haven’t mentioned?
- Transition to how this pertains to

scope.



How many of us primarily work on a pipeline where the primary language is

- Python?
- C#
- C++
- How many of you support pipelines with two languages.
- More than 2?
- Languages I haven’t mentioned?
- Transition to how this pertains to

scope.

What We Will Cover

- Gauge onboarding scope. Learn Standards and Style.
- Find entry points: Step-thru the pipeline and workflow simultaneously using breakpoints.
- Understanding / Embracing debugging principals.
- Validation as a learning tool.
- Documenting and Flowcharting as you go.

Scope:

- First week:
  - get your workstation stood up, read the onboarding wiki docs, skimmed the art bible, now what?
  - It’s time to get a 10K foot view of what you are up against. Audit the pipeline and workflow.

Find Entry Points:

- Start by step-thruing the workflow:
  - push an asset(texture, material, shader iteration, geometry, vfx etc) through the creation phase pipeline as a content creator would do.

Start contributing by debugging:

- Debugging is like taking a dime tour through the pipeline.

Content validation:

- Unpacking the content validation system, or establishing one is the next most efficient way to learn the codebase after debugging. Cuz this is all the I/O (or should be)

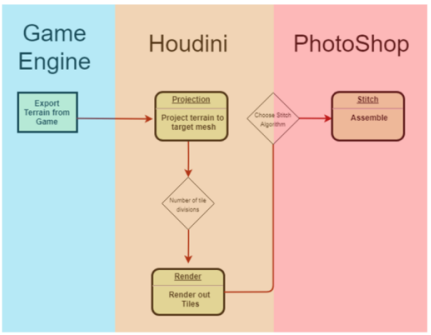
As you step-thru through the workflows initially, run the pipeline code in debug

mode. Set breakpoints in your exporter code and see the different phases the data goes through as it makes it way in-game.

- Keep notes through all of this exploration, because we'll use them later to make flow charts, tutorials and wiki docs for the team (and yourself).

Traversing Pipeline and Flowcharting

- Traverse the pipeline yourself. Use breakpoints when data changes state.
- Flowcharting helps you visualize issues in a different context all at once.
- Catch poorly placed functions that belong in another class/module.
- Data models can be evaluated together.



- All the best systems involve user input. You should be the first user you meet. Try things out right away as a content creator. Take exhaustive notes, more on this later.
- Now that you have test driven the workflow and pipeline, record a desktop video with camtasia or similar desktop capture utility.

- Preferably by/with an artist who is familiar with the process (for your subsequent runs).
- Have the artist drive, and you ask questions, or if they have time, you drive and ask them questions.
- Ensure that you have a decent framerate with sound.
- This isn’t a tutorial, quality isn’t paramount. It’s a snapshot of how things work so that you can rewatch it and have a known working instance.
- Talk your way through it so that when you watch it later you have some context.
- Keep notes of observations, opportunities for automation.
- This is the best time to capture your own untainted observations.

You won’t get a 2nd chance at your first pass impressions, and capturing them is so valuable. This is gold.

Learning Coding Standard and Style

- Coding standard implies code reviews.
- You will be held accountable beyond bug fixes and features.
- Modernize legacy code as you find it.
- If there is no standard or single style, establish one.
- Be consistent, and tread lightly.
- Find a mentor for advice.

If your team has a coding standard, chances are good they are doing code reviews.

- This means your work will be scrutinized, and your adherence to this standard will be a part of that.
- It’s up to you to gauge how closely and how many of these standards are realistically followed.

Modernizing:

- If code doesn’t meet the standard, your team will probably have a modernization plan. If not, push for one.
- My policy is to modernize code modules or classes that that I’m bug fixing or involved in a feature request.
    - Sometimes this rabbit hole can distract you from your mission of onboarding.
    - Digging deeper than necessary to responsibly fix or augment is not the goal at this phase of onboarding.
    - In this case I write Todo’s in comments in the code, and make personal notes to go back later and address.

Establishing a standard and style:

- Establishing a coding standard isn’t a simple task. Luckily, there are great resources.

    - The .Net framework has great examples for C#
    - pep8 for python.
    - Javascript is all over the map.

- The most important thing is to make a choice and be consistent.
    - Even consistently mediocre is better than the wild west hodge-podge of different styles.

- Tread lightly, not everyone wants to adhere to this concept.
    - You must get buy-in before you start editing code you didn't establish.
    - This can be as much a cultural battle as it is one of labor.

Find a mentor:

- If this is your first time establishing or majorly refactoring a code standard, seek advice.
- Find a mentor. Preferably

someone who is on your project, or at your studio.

- They can help you gauge the scope and cadence of these changes you’d like to make.

Learn By Fixing Bugs!

- Bug fixing is learning with purpose and impetus.
- Stay abreast of every emergent tech-art bug.
- Downsides of Unfocused Learning:
  - Retention is poor without context.
  - No definable endpoint.

- As soon as my preliminary explorations of the workflow are complete, I like to jump in and start fixing pipeline and workflow bugs as soon as possible.
- Bug fixing usually isn't fun, exciting work. But it's one of the best ways to learn a new pipeline.

- This does many important things for you as a newb:
  - You start making meaningful

contributions with your first few tasks.
    - Bugs can be quick wins and give you confidence, sea legs.
    - Bugs expose where the pipeline is weak. This is where you should be focusing your efforts, or, at least it should lead to them.
    - Opportunity to leave ‘todo:’ comments in your code. More on this later.
    - Gets you interfacing with the content creators: your customers. You are there for them. More on this later.

- This is why it’s important to stay aware of bugs in all the systems you work in.
    - Even if you won’t be fixing these bugs,
    - They might be related to bugs you ‘are’ fixing, or will soon address.
    - Each new issue is a vertical slice

through the pipeline, an opportunity to see how something works.
    - Think of it as a haphazard tour through the codebase.

- The reason I advocate these methods for learning your codebase, workflow and pipeline, is that unfocused learning is prone to some issues:
    - If you don’t have an actionable reason to review technical things like source code, then it’s much easier to forget what you’ve learned.
        - Retention needs context. Association and repetition helps us with long-term recollection. This is the association part of that equation.
    - Bugs have built-in endpoint - when the bug is fixed.
        - This is important because there is always more work you

can do,

- more things to explore,
- more code to refactor

- It’s your job to scope this properly, more on this later.

Bugs

- Always reproduce the bug. Make no assumptions.
- Understand each class/module in its entirety while fixing a bug.
- Building this baseline understanding of your environment is an inevitability. Take this opportunity to learn and retain.
- Bug-fixing builds a bond with the content creators. Hero status!

Reproducing bugs:

- I’ve talked about spending too much time, and diving too deep too early, focusing on bugs and step-thru-thru’s to explore the codebase and workflow environment.
  - But the opposite is true too, you need to slow down and reproduce your bugs carefully. It’s very easy to make assumptions about what’s

wrong when you are new to things. We'll review debugging principles in the next slide.
    - Even if you have a QA department verifying each bug, reproduce it yourself.
    - Even if you have already tread this part of the pipeline, do it again.
    - This speaks to the other part of recollection: repetition.

Understand everything before you edit/add:

- Once I'm in the code and I've found where I need to implement my bug-fix, I still make sure to read the classes and/or modules in their entirety, beyond what is needed for the fix.
- If there are data-structures I haven't seen I make sure I understand their I/O and how they are referenced throughout the codebase, extra thoroughly if I plan on changing

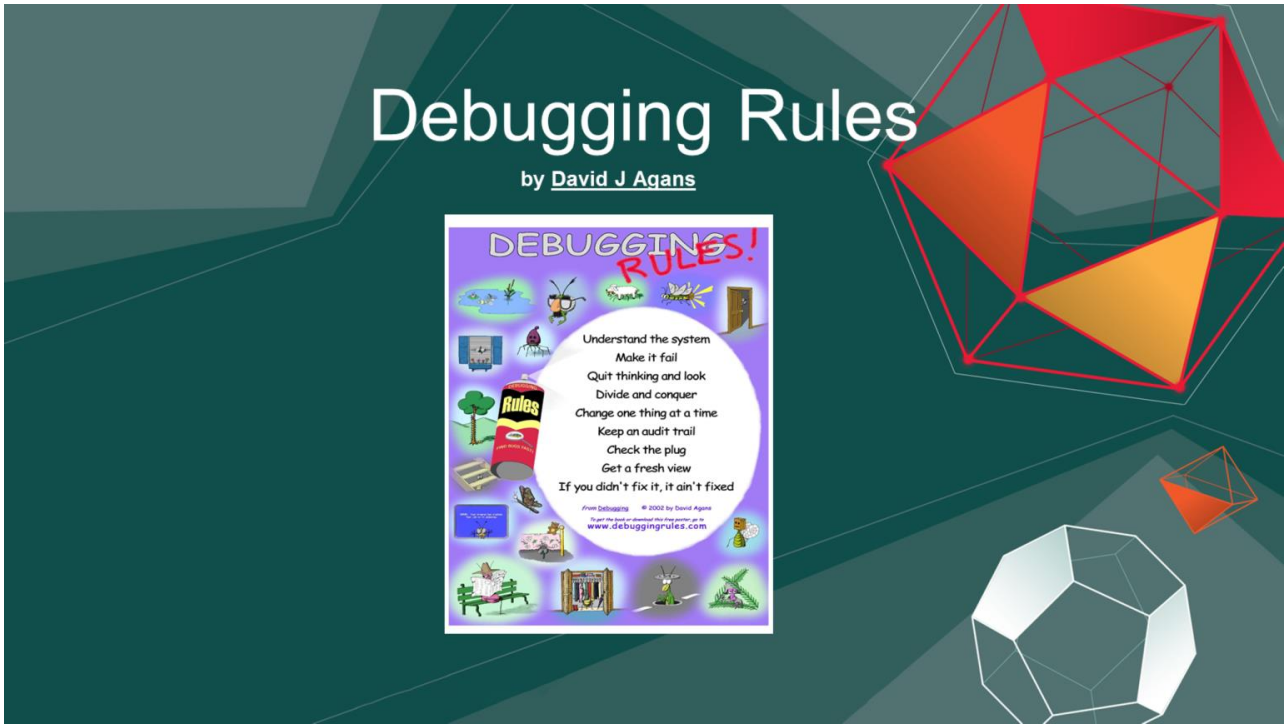
them.

Building a baseline of understanding of all the systems you work in will have to happen eventually.

- Building this general familiarity of each system you are in speeds up the next related bug fix. You might find low-hanging fruit while you do this.
    - This is a stark difference than fixing code you have written recently or become very familiar with. You know your way around, you have some sense of scope.
    - Now you are doing this from scratch, so you cast a wider net when diving into a bug.
    - Gauging how wide to cast your net comes with experience. None of this is wasted time if it’s done with the correct mindset.

- Bug fixing also allows you to interface with your team. Unblocking

them is a great way to bond with your teammates, you’re the new hero!



- Okay this is the last slide on debugging, but it's an important one. All the best lessons of debugging I learned by following this book.
    - **Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems Paperback – November 5,**

2006

- Unpacking all these rules is too much for a 30 minute talk. But understand that they are vital to solving bugs effectively and many of the principals I’m speaking of were learned or crystallized after reading this book.
- If you try and follow these rules while unpacking the pipeline codebase and the workflows, you’ll be learning them in the best way that you can.

Try to Extend Before You Append

- Attempt to extend a class/module instead of making a new one.
- Forces you to understand and use what's already there.
- Reduces the fragmentation of systems
- If feature is closely related to others: data-structures and classes might exist where your additions belong.

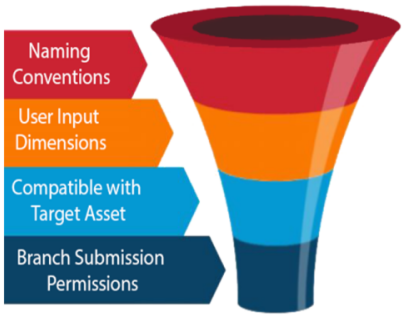
- Eventually you'll do moire than fix one-off bugs:
- extend a system or add a new feature,
- fixing a time-consuming bug with validation.
- he temptation could arize to refactor everything, or just append a new class to implement your change. First try and squeeze this change into an existing system.

- This is good for so many reasons, but the ones pertinent to this topic:
  - Forces you to understand what is already written.
  - Appending new blocks of code could reproduce procedures already completed somewhere else.
    - This fragments and/or duplicates functionality.
  - Forces you to adhere to a different design pattern than one you might have chosen. That’s a learning opportunity.
  - This is not to say that appending or rewriting is always 2nd choice, but when you are learning a new codebase you’ll learn more about onboarding when you extend before append.
  - Features in related systems usually have data structures and wrappers you can

leverage. These are great entry points or resources for your work. Learn them, use them.

Has Validation Been Implemented?

- What is validation: Screening and filtering methods for user input.
- Validation is an excellent learning resource.
- I/O is the gateway between data-exchanges, great spot for debugging.
- Read validation code for every system before you interact with it.



- Content creation validation refers to procedures that filter content before it enters the pipeline. It ensures that the downstream procedures have all the data they need to complete.
- It is one of the pillar issues I always address early when learning a new pipeline.
  - Validation gives you a thorough understanding of your I/O (inputs and outputs)

required by your pipeline.
    - Once validation passes output, you know you have a minimum viable product. Everything else speaks to usability, structure, efficiency, and readability, but you can't have success without valid I/O.
    - I always start at the end. Establish acceptance criteria, and then work backwards all the way to user input.
- Like our debugging explorations, we want to understand the I/O systems in their entirety when interacting with them.
    - I/O junctions are the gateway between data transformations. This is the best point to validate.
    - We want to take advantage of the context validation provides, and learn as much as we can while we make our validation contributions.
    - Make sure you explore all paths the input will take once it's been

validated at that specific stage of the pipeline. (diagram)

- If there are multiple references to the data after this point, you could break something if you change data structures.
  - This is the biggest sticking point for validation: Don’t validate one procedure by changing a data structure and in the process break another.
  - Understand how data is referenced and validate accordingly.

No Validation?

- Establishing validation is a golden opportunity.
- More validation means fewer bugs.
- Shorter investigations for the bugs that get through.
- Preserves user confidence in tech-art.
- Promotes user autonomy = less interruptions for you.

- Starting validation from scratch is my favorite part of this process. You get to dig into your pipeline and really learn it.
- Validation sells itself. More validation almost always translates to fewer bugs or more efficiency.
- Bug fixing with validation means your first investigations are at least partially directed because you know what not to look for.
- Entry points for validation is usually a junction point in your pipeline. It’s a great location to place additions to the codebase, because it’s either the start or stop of ‘something’.
- A well validated pipeline usually translates to handled exceptions for the content creators.

This does many things:

- If they didn’t deliver assets in the right format, we can tell them with a dialog or log entry.
- It shows tech-art competence with the workflows and pipelines we maintain.
- Preserves/restores the artists confidence that we’ve stepped through this process and made sure everything works.
- Artists with good validation learn the pipeline faster/better. They have more inf about validation failures and can help themselves through some problems.

Talk to Veteran Artists

- Always find an owner or veteran for a bug or feature before you do work. This keeps us on track.
- The code won't tell you which features are being actively used.
- Greater incentive to learn the code that is actively used by the artists.
- Don't waste time on unused systems. Veteran are good guides towards deprecation of old code.

- Chances are: workflows and pipeline stakeholders are the veteran content creators. When you start working on a bug, or a feature request, it's always a good idea to find out the history of the bug/request. Old school artists will know. Use Level Scanner feature example.
- The code can't tell you which

features are being actively used (unless there is code for that…). But old schoolers show you best practices.

- There should be a great incentive for you to learn the systems the artists spend their most time using. This is where tech-art should focus, because the best places for optimization see high traffic.

- Old school artists know which systems are dead or dying, and why. This can help guide you towards code rot.
  - Allowing unused code to sit in your codebase is a bad idea. If it's not being used, fix it so it's usable or get rid of it.
  - The longer this kind of code hangs about, the greater the chances it'll fester into a problem.

Veteran Artists Hold Tribal Knowledge

Veteran artists know:

- The short-cuts.
- The pitfalls (often unaddressed bugs).
- What are the game-engine limits/edges/exploits.
- What holds us back iteration-wise.
- Best new features that leverage the most potential.

- They know:
  - They often know the fastest way to do something
    - Or the most fool-proof way.
  - Hold the secret tricks that circumvent unreported bugs.
  - These are the suggestions and awareness that often lead to the best project-specific performance optimizations.

        - Junior developers often know the latest and greatest techniques. Old school artists are aware of how-to push the game engine/assets to their limits.
    - They often know where the non-creative iterative cycles are. The ones you need to automate.

    - veteran artists can tell you ‘if I only had “X, I could make Y”. Newer artists might not know what is possible because they just haven’t had the time to dig into the cracks of the workflow. The old schoolers have probably done this.

When to Seek Help

Cognitive Resource Theory:

- High stress we lean on tribal knowledge, experience, habits and patterns.
- Low stress we lean on intelligence.
- Use this to figure out when and what kind of help is needed.
- I grind a problem for no more than 1 hour before seeking help.

- There was a by Fred Fielder and Joe Garcia that addresses how stress relates to a leaders tendency towards experience vs. intelligence.
- My takeaway from that is to learn your pipeline and workflow as thoroughly as you can, because eventually you'll be point-person on a bug or feature during a high-stress period of production. You'll

be applying all this experience in these high stress moments.

- Grinding away at a bug or a feature implementation can cause stress when we need it least, when we need to think creatively.
  - This is where I usually take a moment to gain perspective.
  - I ask for help after 1 hour of grinding. Anymore and I'll start sacrificing cognitive resources. The lower they get, the less effective of a thinker I'll be.
  - Experience helps structure thoughts and unpack complexity.

- Cognitive resource theory is a very interesting read, and I urge you all to understand the conclusions and apply they to your work.

Synopsis

- Traverse the workflow and pipeline yourself. Take notes, record step-thru's, and flowchart what you find.
- Learn the coding standards and style of the codebase.
- Modernizing is a great entry point to pipeline discovery.
- Open bugs and validation as entry points to the pipeline/codebase.
- Resist the urge to refactor. Try to extend before you append.

Synopsis cont.

- Use breakpoints to help unpack the codebase. Step thru the workflow/pipeline in this manner.
- Glean from veteran artists to learn things that only experience can teach you.
- Be aware of cognitive erosion caused by stress.
- The experience you compile during onboarding can be leaned on during times of high-stress.

Online Resources

- **Cognoitive Resource Theory**: Wikipedia
- **Debuggingrules.com** – best book I've read as a tech-artist.
- **Tech-Artists.org** Forum –help for tech-art specific issues, discussions
- **Stackoverflow.com** – answers for most of my general coding problems.


Good Luck
rosspatel@gmail.com