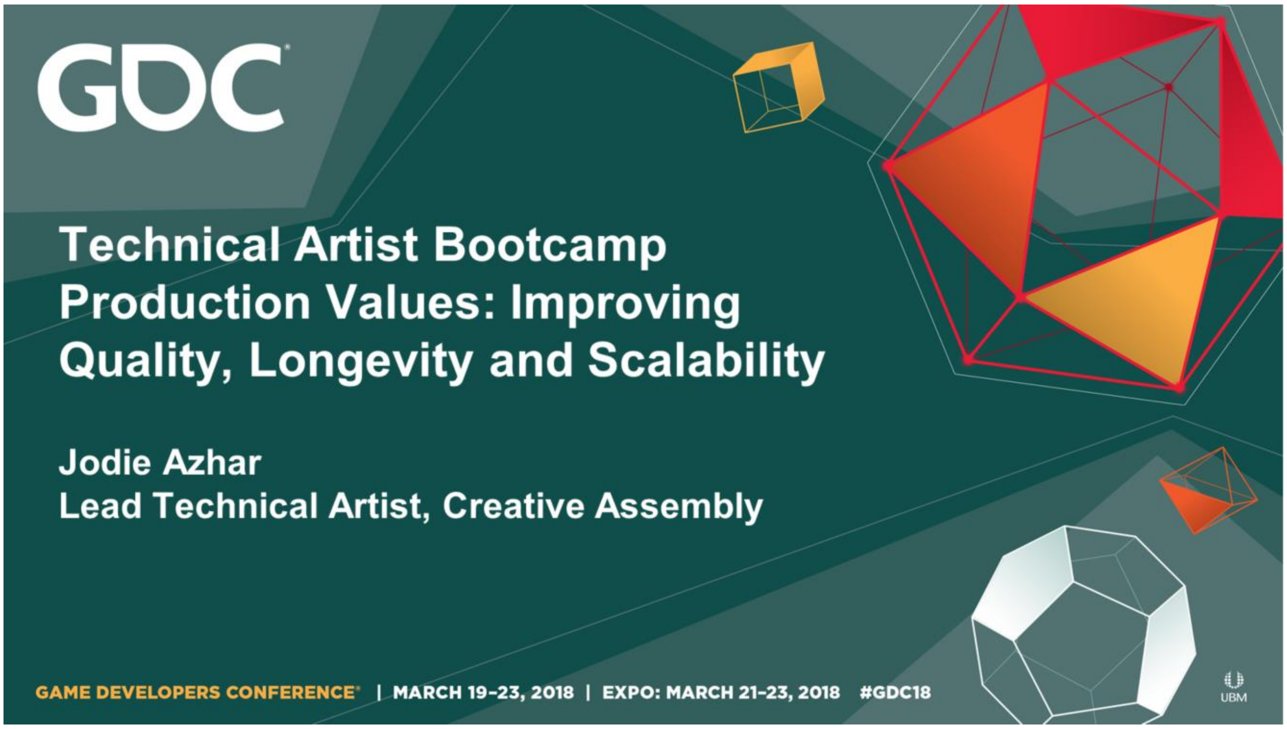


GDC

Technical Artist Bootcamp Production Values: Improving Quality, Longevity and Scalability

Jodie Azhar
Lead Technical Artist, Creative Assembly

UBM

Ask Questions

@JodieAzhar

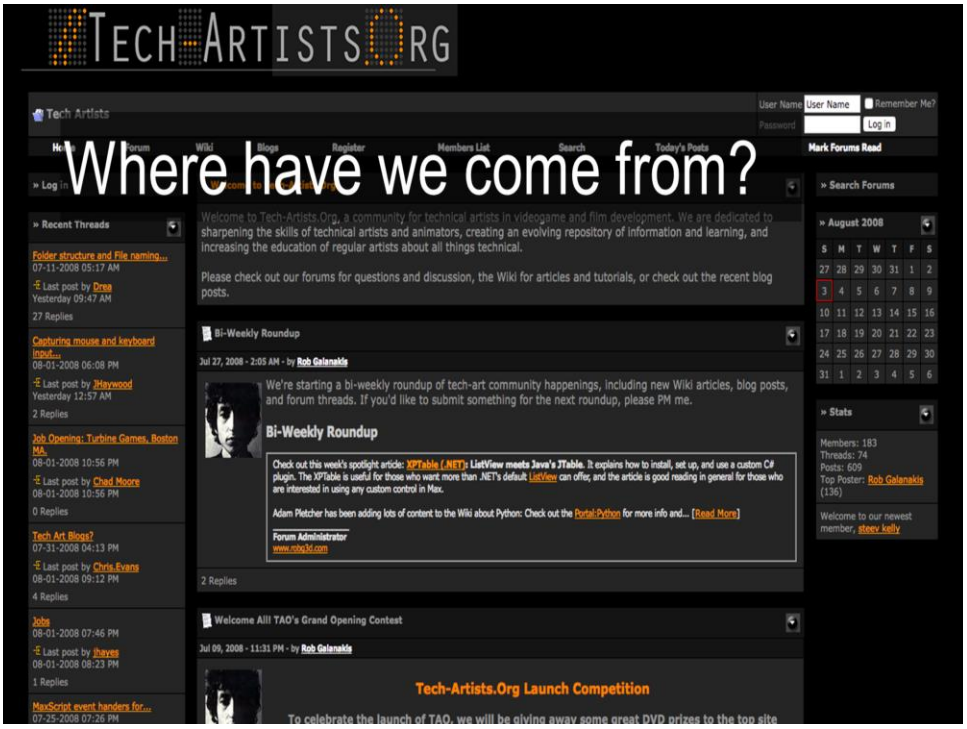
jlazhar@gmail.com

ask who's a TA, artist, programmers, student etc.

The irony of this slide is that I wont be taking questions at the end, but you are more than welcome to message me on twitter, email me, come have a chat after the presentation in the break out area, we'll be at Jillian's tonight and the roundtables all week are here to answer your questions by every TA here at GDC who attends etc.

Within your job learn to constantly ask questions. Whether that's to the artists you need to support, programmers on why they do things a certain way, designers on how they made a particular decision or producers on how they organise 1000 different tasks that need doing.

We as technical artists are naturally curious people and everyone on your project team has the aim of making the best game they can. So for those here who are early on in their career, or have yet to enter the industry - get over the fear of sounding stupid as quickly as possible. Whatever you have to ask we'll likely have heard before, and you will learn so much faster by just listening to others' experience and knowledge, and using it as a shortcut rather than trying to learn everything yourself and trying to figure out what search words you need to find something useful



Where have we come from?

Welcome to Tech-Artists.Org

Welcome to Tech-Artists.Org, a community for technical artists in videogame and film development. We are dedicated to sharpening the skills of technical artists and animators, creating an evolving repository of information and learning, and increasing the education of regular artists about all things technical.

Please check out our forums for questions and discussion, the Wiki for articles and tutorials, or check out the recent blog posts.

» Recent Threads

Folder structure and File naming...
07-11-2008 05:17 AM
Last post by Drea
Yesterday 09:47 AM
27 Replies

Capturing mouse and keyboard input...
08-01-2008 06:08 PM
Last post by JHaywood
Yesterday 12:57 AM
2 Replies

Job Opening: Turbine Games, Boston MA.
08-01-2008 10:56 PM
Last post by Chad Moore
08-01-2008 10:56 PM
0 Replies

Tech Art Blogs?
07-31-2008 04:13 PM
Last post by Chris.Evans
08-01-2008 09:12 PM
4 Replies

Jobs
08-01-2008 07:46 PM
Last post by jhayes
08-01-2008 08:23 PM
1 Replies

MaxScript event handers for...
07-25-2008 07:26 PM

Bi-Weekly Roundup
Jul 27, 2008 - 2:05 AM - by Rob Galanakis

We're starting a bi-weekly roundup of tech-art community happenings, including new Wiki articles, blog posts, and forum threads. If you'd like to submit something for the next roundup, please PM me.

Bi-Weekly Roundup

Check out this week's spotlight article: XPTable (.NET): ListView meets Java's JTable. It explains how to install, set up, and use a custom C# plugin. The XPTable is useful for those who want more than .NET's default ListView can offer, and the article is good reading in general for those who are interested in using any custom control in Max.

Adam Pletcher has been adding lots of content to the Wiki about Python: Check out the Portal:Python for more info and... [Read More]

Forum Administrator
www.robg3d.com

2 Replies

Welcome All! TAO's Grand Opening Contest
Jul 09, 2008 - 11:31 PM - by Rob Galanakis

Tech-Artists.Org Launch Competition

To celebrate the launch of TAO, we will be giving away some great DVD prizes to the top site

» August 2008

S	M	T	W	T	F	S
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

» Stats

Members: 183
Threads: 74
Posts: 609
Top Poster: Rob Galanakis (136)

Welcome to our newest member, steev kelly

Because we've already done the boring bit of filling up stackoverflow and the rest of the internet with answers to weird problems.

Technical Art as a role has been quite organic in its growth. It's made up of artists who realised there were technical problems to be solved with the art they were making, and programmers who wanted to contribute more to art quality and art generation. We've grown out of a time of convincing managers that technical art is a thing, and more studios are now realising that they need someone, or a whole team of someones, dedicated to this area in order to support their games.

This organic growth has meant we haven't developed our own set of standards and formal language, but borrow from the technical roles and apply them to the work that we support.

Since technical art is a very broad area, covering all areas of art generation from texture manipulation, to mesh transformation, VFX optimisation and animation simulation to give but a few examples, there's still quite a bit of ambiguity as to what it means to be a technical artist.

Part of my motivation for wanting to talk about Technical Art in a more formal way was a conversation I had not too long ago with a University lecturer who said that every company they'd spoken to had a different idea of what a technical artist is, so there wasn't any point to catering for it on their University course.

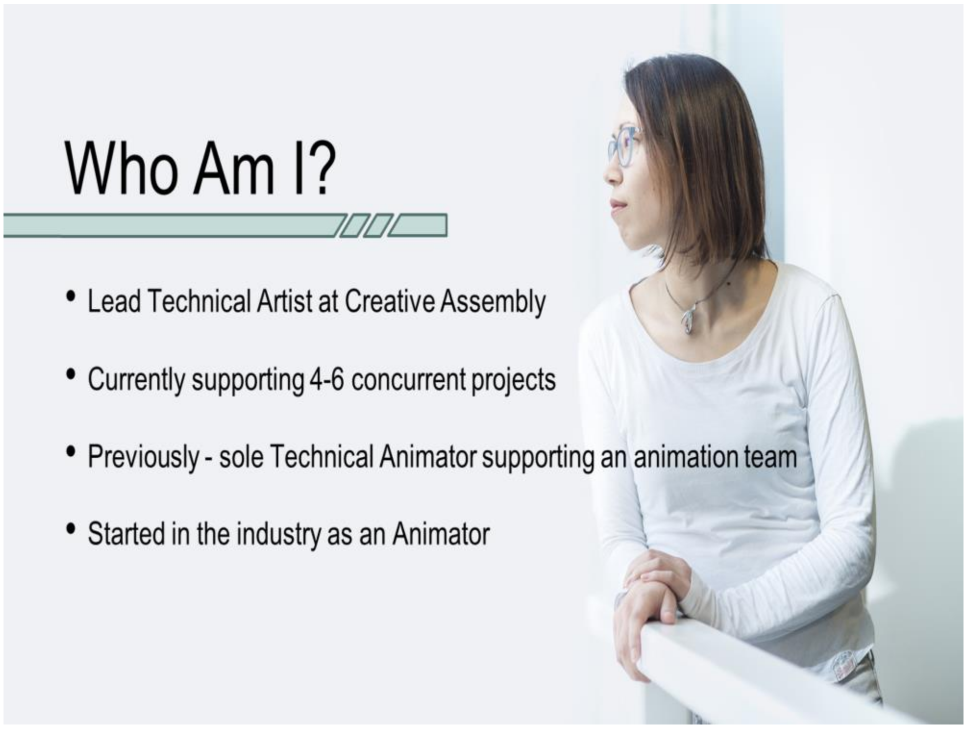
Now my initial reaction was, of course you can teach Technical Art, as a hiring manager I know exactly the kind of person I'm looking for. I can be flexible because every TA has a different background and different specialisation areas, but I know the core skills that they need.

And I know that you can teach this stuff, because I got taught it.

I was lucky when looking for University courses to find one at Bournemouth University in the UK that included teaching maths for computer graphics, programming and 3D art and animation.

That course was rebranded last year to Computer Animation Technical Arts, both recognising and clarifying that that's what they're teaching.

But then I realised, actually to teach somebody to be a TA, there is so much that they can learn that you'd need to dedicate an entire course to it, and that was the only reason I'd been able to learn it at University.



Who Am I?

- Lead Technical Artist at Creative Assembly
- Currently supporting 4-6 concurrent projects
- Previously - sole Technical Animator supporting an animation team
- Started in the industry as an Animator

I now work at Creative Assembly where I oversee the Technical Art team across all our Total War projects, which can be between 4 to 6 concurrent projects at any one time.

I’ve also worked at other studios as an animator, and also as a solo technical animator supporting an animation team which was working on 1 to 3 projects at a time. So I’ve experienced different ways of having to work and prioritise my work, and also been a lone TA on a team and part of dedicated Technical Art department

Creative Assembly is a big studio, so we pretty much just hire specialists. But what I’m looking for from a TA is someone who is more diverse.

They need to understand how artists think, how they interact with tools, what *they* want to achieve.

But they need to solve problems, they need to think like programmers to break a problem down, describe it to the code team, help come up with solutions and sometimes code the solution themselves.

They also need to design an experience. The entire pipeline of creating art needs to be user friendly and robust. We need to make solutions that last, but we also need to fix that urgent problem right now.

And suddenly I’m left with this overwhelming feeling of how do we get anything done or know we’ve done a good job?!

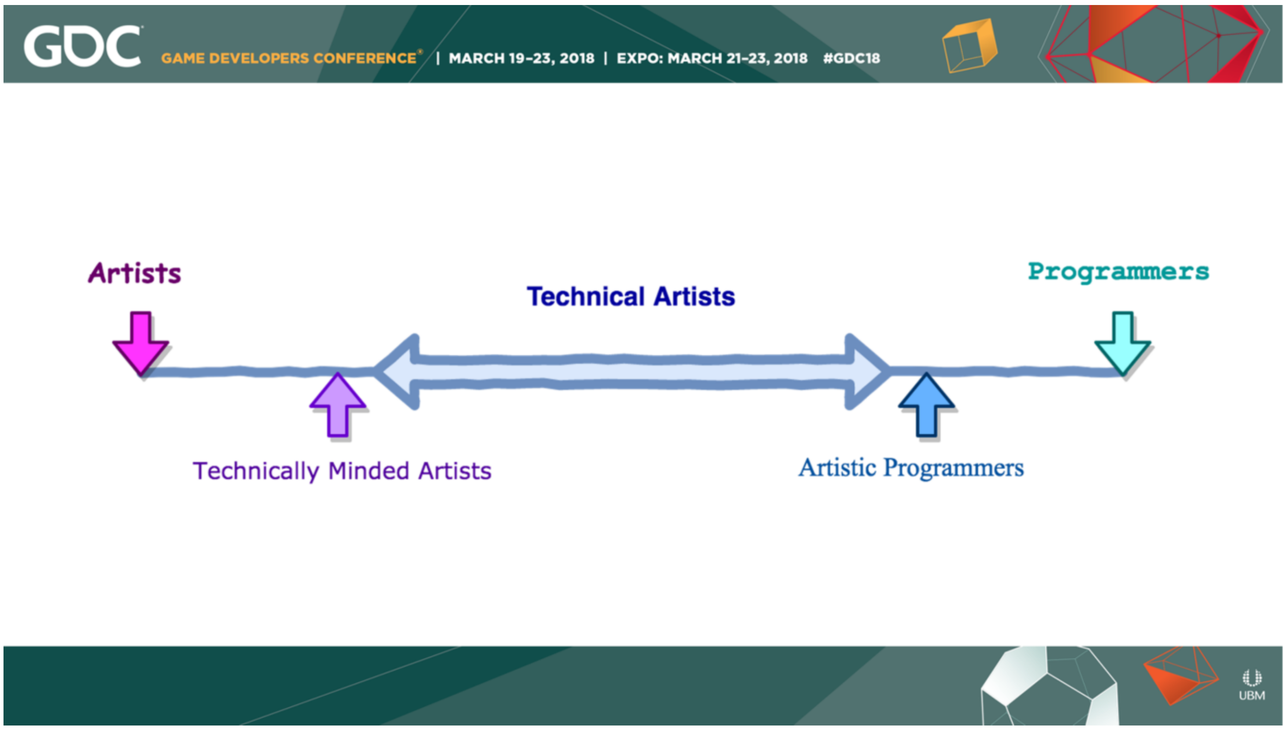
Where are we going?

But we do get things done. And what we do is incredibly valuable. There’s a lot that makes up the realm of Technical Art, but we don’t need to know everything at once.

So we’re going to look at some of the various methods, techniques and processes that formalise our work and that can be applied both when you’re that lone technical artist supporting a team in *everything* they want to get done and when you’re part of a big multi-project team that has to make sure your tools don’t crash on half the projects.

Most of the following slides could have an entire presentation dedicated to their topic, so the aim here is to cover different techniques and methods that can be used in your day to day work to make you think more critically about what you do ,with the goal of being more effective. They’re a starting point for expanding your technical art tool box and if any of them sound useful, I’d encourage you to go and find out more in depth information

And hopefully all of them will encourage us to develop a more formal vocabulary when discussing technical art, and help the position of technical artist mature in how all developers and aspiring developers understand it.


GDC GAME DEVELOPERS CONFERENCE | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18
Artists
Technical Artists
Programmers
Technically Minded Artists
Artistic Programmers
UBM

I'd like to make a distinction between technical artists and "technically minded artists", a term you're more likely to encounter in larger studios.

Technical Artists occupy this space between programming and art. We bridge the gap in understanding and we may be more artistic or more technical.

But we don't just write code and our job isn't to *make* art assets. Artists are our clients and many of us will have come from an art background, but our focus is solving problems, and we will never run out of problems to solve.

If you want to make art be an artist who thinks technically. Video games is a technical area so if you are technically minded you have a huge value to your projects.

I make the distinction because I've met several artists who have technical knowledge and found themselves as technical artists and realised that they were spending less and less time making the art that they love, because they then had the responsibility to support others.

Many game developers will categorise any technical task within the area of art as being the job of a Technical Artist but I believe we can make a relatively clear distinction and I think it's important to do so because

- Technical Art is a vast area in itself
- And I think we should be empowering artists who are technically minded and enforce in others to expect artists to appreciate the technical aspects of art in video games with the goal of Make better games all round

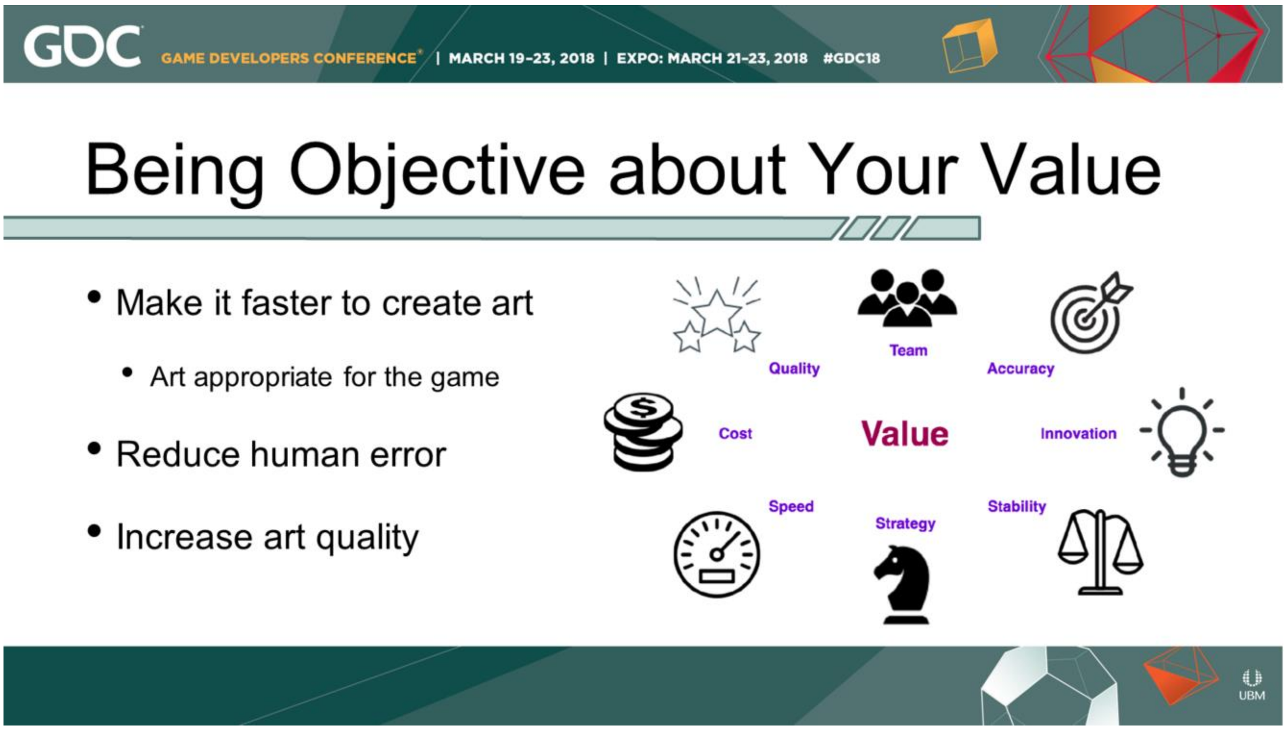
This isn't to say that as a Technical Artist you're completely detached from art production. We have to make art to prototype new solutions. Sometimes it's faster for you to do something than pass it down the chain to another artist. Depends on task and project scope but don't make it so you're the only person who can make that asset correctly.

Also in smaller companies as a technical artist you may make art, its small team, you are both the developer and the client.

Titles themselves aren't the important thing, it's the responsibilities that people hold, and titles are usually there to make it clearer to the team what those responsibilities are.

Understanding artists workflows and way of thinking is key. We're not programmers, we can write code but it's from the perspective of how do we make solutions that artists will use.

The developer-client relationship is so important. That's why TAs coming from an art background are so useful- we need to understand how the art is created. And we can never afford to lose that appreciation and understanding of art and the people who make it



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Being Objective about Your Value

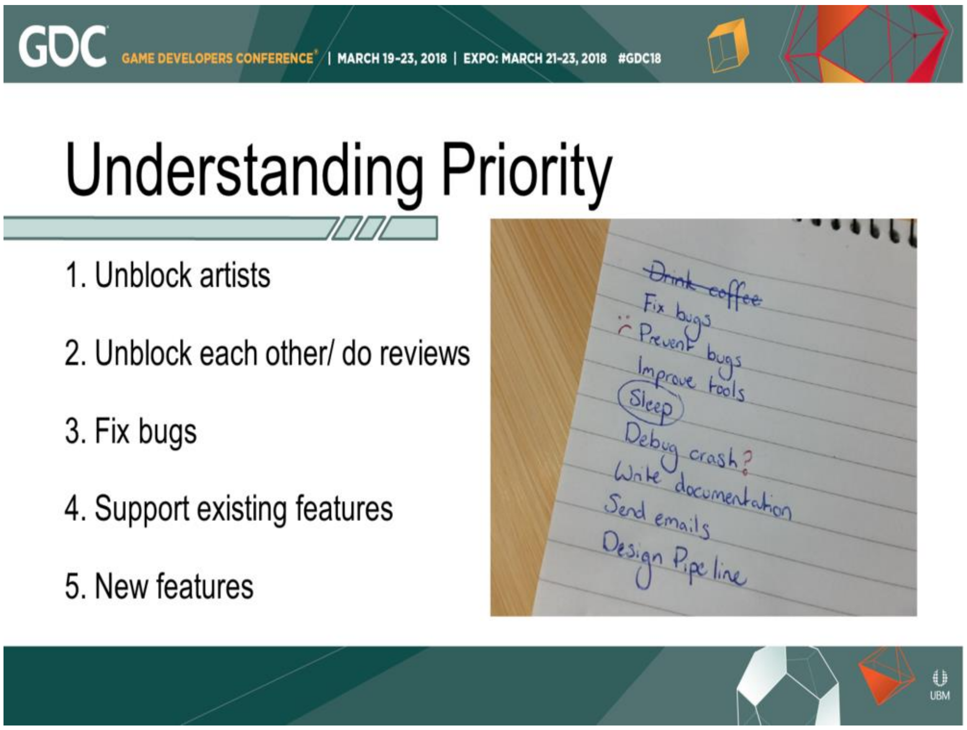
- Make it faster to create art
  - Art appropriate for the game
- Reduce human error
- Increase art quality

Quality
Team
Accuracy
Cost
Value
Innovation
Speed
Strategy
Stability
UBM

So what is our aim as technical artists?

- Make it faster to create art
  - And ensure that it's art that's appropriate for the game
- Reduce human error - through automating tasks we can remove repetitive or boring actions and save time, but we can also ensure that the end result is less likely to have bugs by making it more consistent in how we generate it and reduce the number of points of entry for mistakes by the user.
- Increase art quality - this is the big one It's the end goal because it's what the player is going to see. There's no point in having art that's correct, and functional and quick to produce if it doesn't look good and improve the game product.

So keep these things in mind as you're developing.



GDC GAME DEVELOPERS CONFERENCE | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Understanding Priority

1. Unblock artists
2. Unblock each other/ do reviews
3. Fix bugs
4. Support existing features
5. New features

~~Drink coffee~~
Fix bugs
Prevent bugs
Improve tools
Sleep
Debug crash?
Write documentation
Send emails
Design Pipe line

UBM

Because we're always finding problems, it's important to not get distracted by new tasks and that work gets completed to a standard that's usable and ideally that it will scale well and will last. Meeting all of those criteria takes time, so in the short term we have to make educated decisions on what to work on right now.

- Unblocking artists comes first. We're there to support them.

- These next two may be switched in priority, especially when you're at certain times of a project. Towards the end of a project you may need to focus on fixing bugs in the game.

Also if you don't work on a team with other technical artists there won't be others to unblock. However, you may need to unblock yourself by completing some work. Depending on what motivates you and what you enjoy most about the process, the finishing stage may not be the most exciting. But it's important to put the time into make sure that your tools and pipelines are as bug free as possible and they work for the end user.

Formally reviewing others' work *can* seem like a distraction from you working on your own tasks, but when working on a technical art team it's important to remember you *are* working as a team and that if any of the team's tasks gets completed you've progressed the system forwards. *Never* avoid reviewing each other work and remember this is a *collaborative* process. I'll go more into reviews later

- Finish work before you start something new. Until your task is complete and in the hands of the artists or in game, all the time you've sunk into it has limited value. It doesn't matter if you've done the difficult technical part, it needs to be completed and released into the wild before moving to something else. That can mean not getting distracted by exciting *new* challenging problems, but also supporting *existing* features of a tool and fixing bugs before you start on creating a new tool, even if the new tool would be very useful.

- We need to maintain strong belief from artists that the tools we make work and are reliable. If you have 10 tools that all have useful functionality but are all buggy and artists don't trust that they do what we tell them they do, they'll be less likely to use any of them, as opposed to 5 tools that are reliable and work as expected.

GDC GAME DEVELOPERS CONFERENCE | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Observing Artists

UBM

Something that feels very high priority but that we don't often give ourselves enough time to do is observing artists at work.

Because art is so broad, for many Technical Artists you wont be able to just focus on the character artists or the VFX artists and might find it difficult to allocate time *just* to watch what artists are doing.

However, it's a vital part of understanding them and what their processes are. If you can dedicate regular time to observing your artists, or if you have the capacity for one of your team to be responsible for a particular area, even if it's only for a time, then you will find out things you may otherwise miss.

Artists often downplay or ignore issues. I've had times when I've gone to fix a problem on an artist's machine and got an error message that they've then told me has been appearing for months, but they hadn't mentioned it to the TA team before.

Or they get used to inefficient workflows because they know the process and know that it works, so they don't actively notice that it's a time waste anymore because they trust that even with the inefficiency they get what they want.

You observing them at work adds a fresh pair of eyes on their interactions, a more critical eye to any processes and an opportunity to spot areas for optimisation that may be of more value than adding a new feature to a tool.

It also keeps you in tune with how artists are working so that you don't diverge too much from the artist way of thinking. We're still artists after all and if we forget that side of our role we lose a core part of what makes us effective at solving problems in this visual area.

If you think you're too busy to take time to observe artists, think about the cost of not doing it.

This is a task that we often don't get round to because other high priority tasks pop up and constantly push this down the list of things to do.

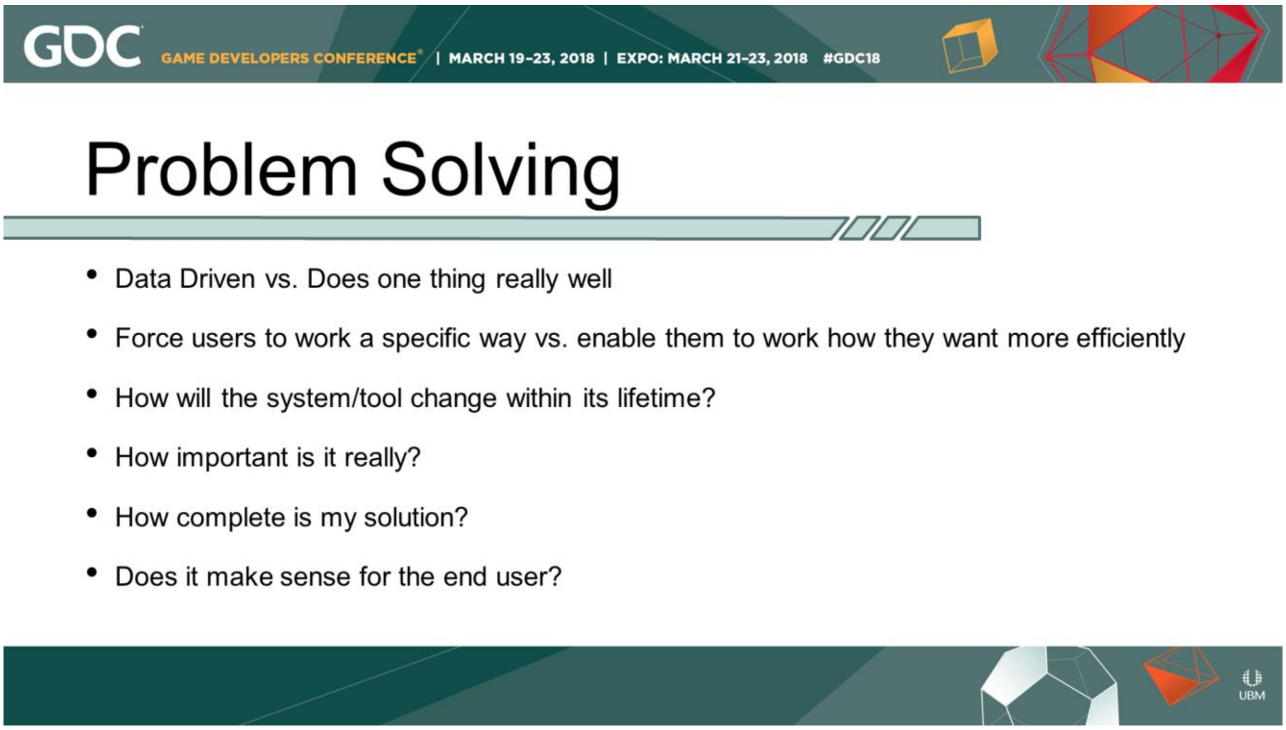
But what are you missing out on by not doing it.

What easy fixes are you missing spotting.

What big issues that lie on the horizon might you catch in the process of observing.

Or what tools already exist that an artist has forgotten about or haven't realised they could use in a different way for a new task in order to be more efficient and then have more time to spend on the creative part of their work.

If you never get round to sitting and observing artists, schedule time in and don't let it be taken up by other tasks. Put it in your calendar and don't let it move


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Problem Solving

- Data Driven vs. Does one thing really well
- Force users to work a specific way vs. enable them to work how they want more efficiently
- How will the system/tool change within its lifetime?
- How important is it really?
- How complete is my solution?
- Does it make sense for the end user?

UBM

Problem solving is at the heart of what we do and there are multiple ways of solving the same problem. Deciding what solution to go with is dependant on a number of factors. What is the required end result, what are the artists already familiar with, are there already ways of doing something similar, is efficiency or accuracy a priority, and many more factors besides. It’s really dependant on the task and place of work.

You don’t have to overthink a problem, or spend a disproportionate amount of time deliberating, but whether you’re an individual or part of a team, giving a problem a bit of thought before diving straight in can give you some perspective and help you measure the efficacy of different solutions.

There are ways of evaluating solutions and deciding which ones are *more* appropriate for a particular instance.

- Do you want to create a data driven solution or should the solution do just one thing **really** well

A data driven solution may be more scalable, but it's also more likely to accept a range of data and needs to handle them all. This gives more overhead and the problem may only ever require one or two data types to be passed.

- Forcing users to work a particular way vs. tools that enable them to work the way they want but more efficiently

There's an excitement to writing a new interface for Maya that's specific to your game engine and workflow and hides things that the asset creators don't currently need.

However, you need to consider how the system will evolve.

What happens in the future? Is it easy for someone to understand the changes.

What happens if the software is upgraded or downgraded?

What happens if in the future there is no technical art team to support it? Can artists still use the software to get assets in game?

Are they still able to access the default functionality without having to hack the system or disable systems required to get things into game correctly.

Early resistance to changes eventually goes away, but you have to get artists over the hump - how long will it take? is that time well spent?

- How will the system or tool change within its lifetime?

Robust systems are great, but how much do you really know about the future?

How will you change during the lifetime of a tool?

You'll learn new things. You'll find different ways of solving the problem. You'll become a better coder.

New technologies may be developed that make your tool irrelevant or replace certain functionality.

How will the requirements for the tool change. Will you change the types of game you make

Again is it worth investing the time

- What is the importance of the problem/solution

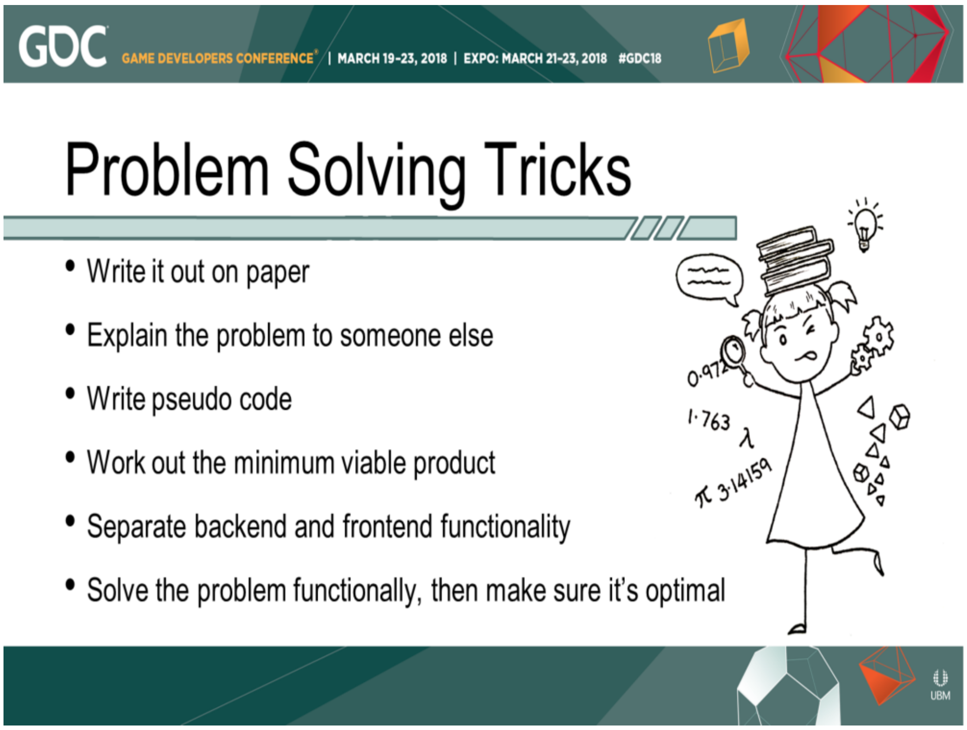
How much time does it really save? Is that a valuable use of time compared to working on something else?

- How complete is the solution? Some problems have 0 use if they are only 90% complete. How confident are you in the solution actually meeting the end user's requirements and you having a thorough understanding of what you intend to implement

- Does it make sense for the end user?

Does it result in what they need

Is it user friendly and does it make the artist confident in using it and in the support you're giving them?


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Problem Solving Tricks

- Write it out on paper
- Explain the problem to someone else
- Write pseudo code
- Work out the minimum viable product
- Separate backend and frontend functionality
- Solve the problem functionally, then make sure it's optimal

0·972
1·763
λ
π 3·14159
UBM

So some tips when trying to come up with solutions

- Write it out on paper - regardless of whether the end result will be an art asset, a piece of code, part of a rig, some performance statistics - if it's a new problem take it back to basics. Write out the problem on paper. Explore your options first before you dive straight in. Make sure you understand what your end product needs to do and what the process will be of getting there.

- Can you explain it to someone else - do you understand the problem and your proposed solution well enough to explain it to another human being. If you can't are you sure you have enough information to know this is the right solution? Do you need to do a review of various possible solutions with another member of the team, or with the client in order to make sure that it sounds like it will actually solve the issue.

- Before you start writing the code, try write pseudo code. This is a half way step between the explaining the problem on paper, and actually starting to write the code for a tool. It allows you to break down the problem and think of it in terms of what the functions need to do, without getting caught up in the actual code itself. You can focus on structure and flow of data before trying to get any working code. If you write the pseudo code as comments you can then write the actual code between the lines which can remain or be rewritten in your code to document the processes.

- work out what the minimum viable product is

Break down the problem - this will be useful later on when you come to reviewing changes - and work out what the simplest version of the tool is that an artist could start using.

it May not have all the features but enough for artists to start using it - this then becomes part of the testing process and you can get the product to the end user early to ensure you are not on the wrong track and that you're both still confident it will meet the requirements.

If any changes need to be made it is then easier to add the additional functionality to a product that you're confident meets needs and is correct

- Separate interface from backend functionality and keep data separate from code.

A good way of developing and keeping code clean and clear to read is separating any area that can work independently. In class based languages this can happen naturally through clearly defining what functions should belong to which class. We can also do it in less structured languages such as Maxscript, by separating code into different files.

By separating the backend functionality we can allow it to be used by multiple interfaces, or even access it directly for batch operations.

Keeping the data separate from the code ensures you’re not littering the code with hardcoded values that you then have to search for in order to update behaviour.

It also allows us to develop project agnostic solutions where the tool is driven by the data. By having the values passed to the code and working on fewer assumed values, we can make our tools flexible and easy to expand to deal with different situations simultaneously.

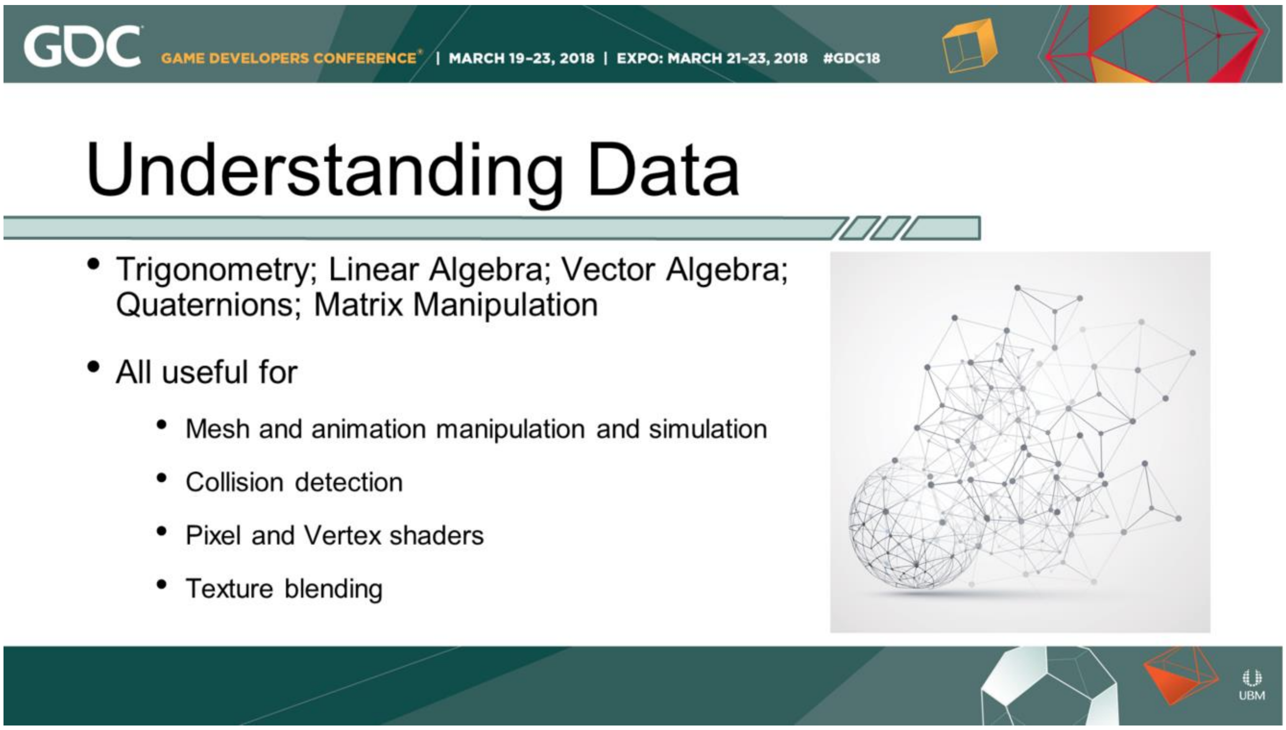
- Solve the problem functionally then do a second optimisation pass.

This isn’t always necessary, however, there’s value to focusing on function first and then doing a second pass to improve performance.

Firstly you can concentrate on getting the solution to do the correct thing, rather than continually try and optimise, realise a bit of code you optimised is no longer needed, or that you’ve accidentally changed the behaviour by refactoring the code as you write it.

Usually people stop after this first stage of implementing the correct functionality because the initial objective has been fulfilled. But you’ll probably have to come back later, so if you do a second pass to rewrite or remove any redundancies then you’ll have a an easier time later on and make it easier for whoever is reviewing your changes.

So if your update is to an asset file, check there are no redundant empty layers, materials, or test data. If it’s a rig update, make sure you don’t have multiple constraints that can be combined, or empty null groups. If it’s code make sure you don’t have logic loops that can be combined, or are calling functions on individual elements instead of an entire array at once if the code allows for it. If necessary rewrite the code changes to be optimal, but first make sure to backup the working version in case anything goes wrong.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Understanding Data

- Trigonometry; Linear Algebra; Vector Algebra; Quaternions; Matrix Manipulation
- All useful for
    - Mesh and animation manipulation and simulation
    - Collision detection
    - Pixel and Vertex shaders
    - Texture blending

UBM

One of the most powerful tools to us is understanding that 3D art is just data and maths.

Our ability as Technical Artists to view and criticise art through both its aesthetic and data forms makes us extremely useful.

We can work on solutions to problems that may rely on mentally breaking down the art into its data forms and expanding how that data is manipulated in order to reach the visual goal that an artist has in mind, all while being in conversation with them and translating those steps into language that they can understand and are confident that what you are telling them will give them what they're asking for.

The level of understanding required will depend on the area in which you focus. It's not a necessity to have a strong mathematical knowledge as a TA, however understanding the maths behind computer graphics makes you a very powerful problem solver and makes it easier to see how your existing knowledge can be applied to data in different ways.

Trigonometry -sine, cosine, tangent

useful for cloth, procedural movement

Linear Algebra

- texture blending - multiply; overlay hard light etc.

Vector algebra

- collision detection - what side of a plane is a point on - useful in animation simulations

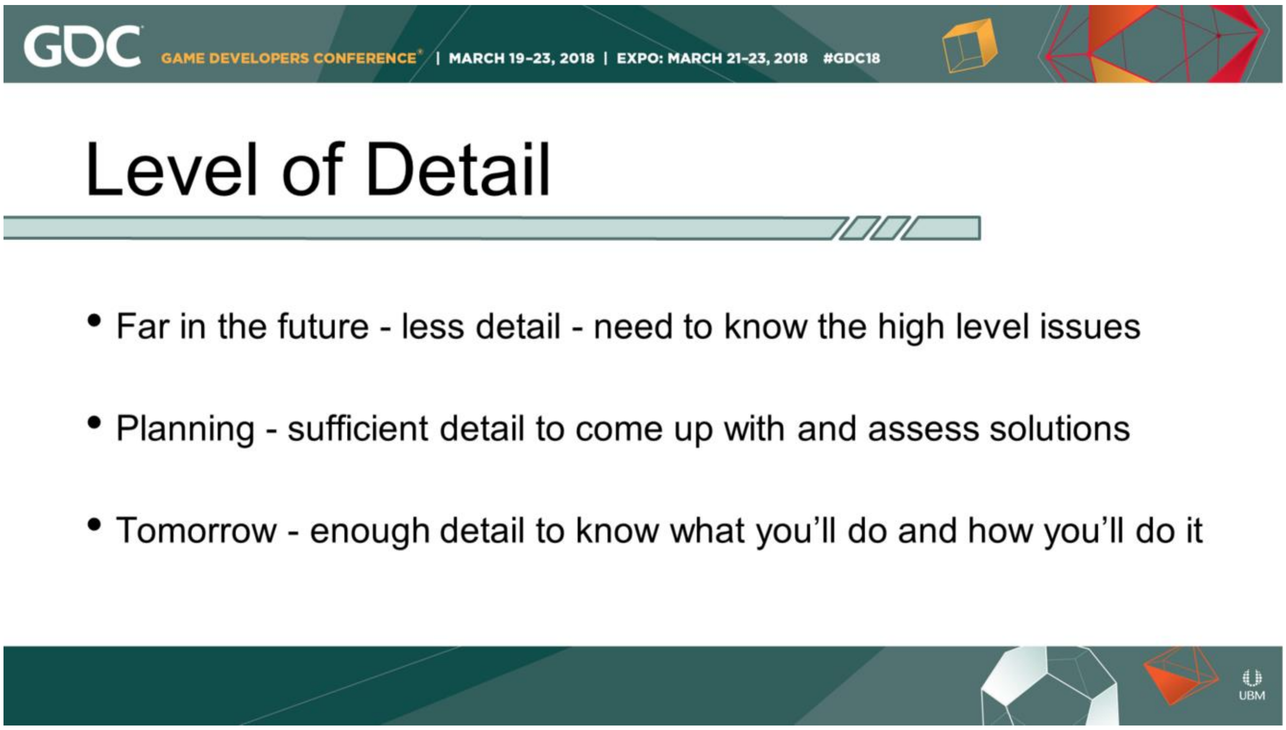
Rotation

- Quaternions
- Useful for hue shifting

Matrix manipulation

- Animation
- Mesh manipulation
- Applying poses - working out the relative position of an object to any other object and transforming it round that point
- shaders - vertex animation

By understanding the underlying maths to how that data is constructed and can be manipulated, you can be much more expansive in your problem solving. With technology pretty much anything is possible, it just takes time to do.


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Level of Detail

- Far in the future - less detail - need to know the high level issues
- Planning - sufficient detail to come up with and assess solutions
- Tomorrow - enough detail to know what you'll do and how you'll do it

UBM

Having the right level of detail when solving a problem is useful.

Do you have enough information to make a decision?

When does the task need to be completed?

But you also don't want to get caught up in fine detail for issues you're going to tackle way off in the future.

Understanding level of detail is especially important if you are planning for the long term. If you're just focused on one project, especially if it's a short project, then it is often easier to be ruthless with requests and decide if something is too large in scope to be achieved.

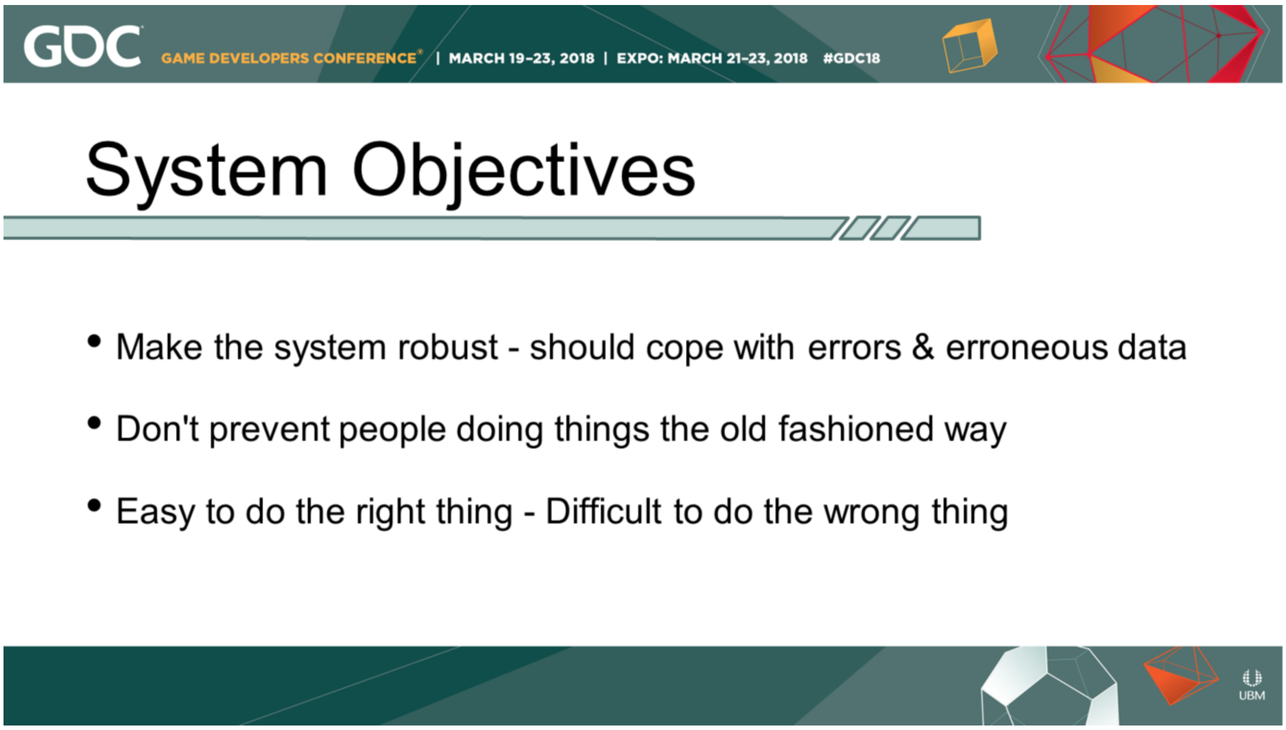
If you're supporting an ongoing pipeline you need to know details to work out how severe an issue it is. Does it need fixing now? Or can it wait. Is it a long term goal?

Small known issues don’t require a lot of planning or detail as we can derive a solution during the task itself. The larger a problem, the more likely there will be information that is unknown or decisions that need to be made on how to solve it.

For issues that can be dealt with in the future you don’t need to know as many details - you need to know the high level issues in order to give a reasonable estimate as to the amount of time it will take, whether it requires collaboration with other departments, or additional resources. However things may change between now and the time of implementation, so you need to give due diligence to the problem, but don’t spend too much time on details that may change.

During the planning stage you need sufficient detail to come up with and assess solutions. At this stage you can work out a more accurate idea of the time it will take to implement, based on the solution you want to follow through with and may break the problem down into discreet tasks. If tasks need to be completed in a particular order, especially when involving other teams, make sure that time is allocated accordingly.

Once it’s time to implement the solution you need enough detail to know what you are going to do and how you are going to do it. You need to confirm that the initial problem is still the same and that the proposed solution is sufficient to support the requirements.



System Objectives

- Make the system robust - should cope with errors & erroneous data
- Don't prevent people doing things the old fashioned way
- Easy to do the right thing - Difficult to do the wrong thing

There are two ways of protecting your job security.

1) Write a system only you can understand and fix, so you're kept on to maintain it.
2) Create good systems and continually improve so you can always move to a better job.

Now if you're even contemplating the first one I'd ask are you really passionate about what you do? And do you feel secure in your abilities?

As Technical Artists we often sit near the head of technological advancement, evaluating and integrating new software that if you're not adaptable you could find yourself skilled out of the market in a few years.

So it might seem redundant to point out the idea that it's bad to write systems that require a lot of maintenance. But it's worth thinking about it on a higher level of

How does the system hold up if you're not there?

Some developers like to be the hero, coming in to save the day when people are stuck, or there’s a problem. Or they want to feel like if they go away their co-workers will notice they’re absence because things don’t run as smoothly.

People often don’t remember when a system just worked, but they do remember the person who came in and fixed the system when it was broken and enabled everyone to achieve what they’re aiming for.

But for your own sanity and growth of ability it’s much better to write maintainable code and systems that don’t rely on you. It will give you more time to spend solving interesting, challenging problems, rather than just maintaining a system. There are plenty of opportunities for being the hero as a Technical Artist just by doing the required job

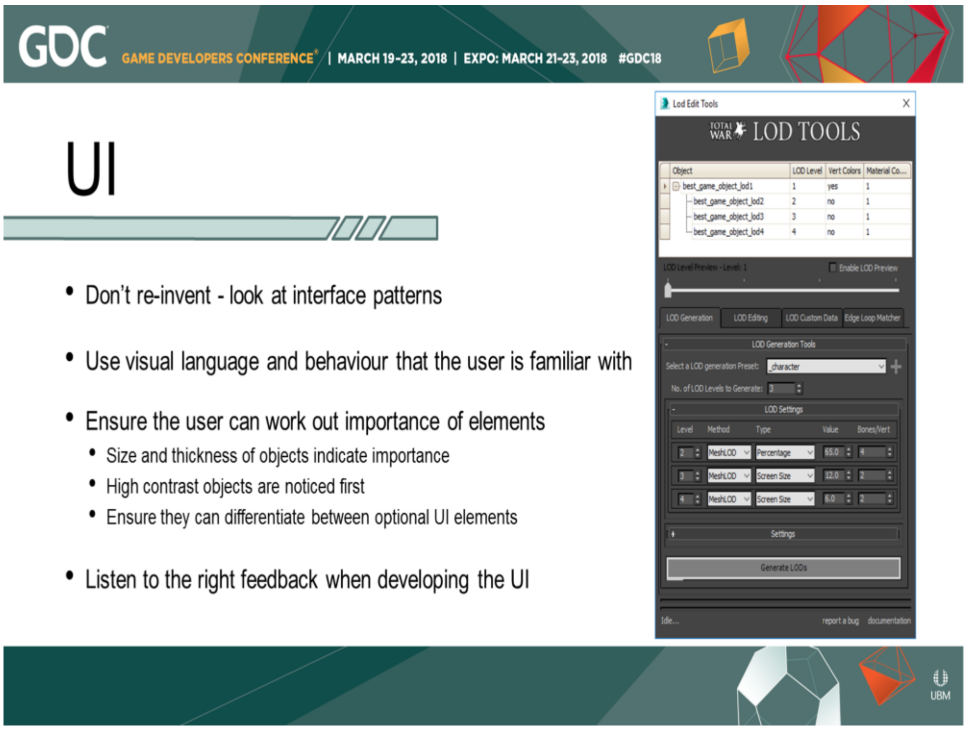
It’s also worth thinking - What would happen if all the tech artists left?

Are the systems robust? Do they handle errors and erroneous input. Is it clear to the user what to do in order to progress if they don’t do the right thing.

Sometimes it’s easer to override a software’s default behaviour to get people to do what they need to to get something correct for the game, but does this prevent the user from using standard behaviour? Overriding default behaviour in software can make it harder for users to debug problems themselves as they can’t do a search online expecting the same results.

If the system fails can the user still get things into game a more manual way.

As a rule of thumb make it easy to do the right thing and difficult, but not impossible, to do the wrong thing. Artists will naturally do the thing that they are most comfortable with, and if it’s easy to do then they are unlikely to be resistant towards it. If you don’t want them to do something, make it difficult. This’ll deter people from going out of their way to do something that you don’t want to support or wont be supported in the game anyway, but it might be necessary in the future, so if you don’t make it impossible there’s still room for the most adventurous artist to use a feature for testing purposes before the future TAs make that functionality easier if necessary


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

UI

- Don't re-invent - look at interface patterns
- Use visual language and behaviour that the user is familiar with
- Ensure the user can work out importance of elements
  - Size and thickness of objects indicate importance
  - High contrast objects are noticed first
  - Ensure they can differentiate between optional UI elements
- Listen to the right feedback when developing the UI

Lod Edit Tools

TOTAL WAR LOD TOOLS

Object	LOD Level	Vert Colors	Material Co...
best_game_object_lod1	1	yes	1
best_game_object_lod2	2	no	1
best_game_object_lod3	3	no	1
best_game_object_lod4	4	no	1

LOD Level Preview - Level: 1   Enable LOD Preview

LOD Generation | LOD Editing | LOD Custom Data | Edge Loop Matcher

LOD Generation Tools

Select a LOD generation Preset: _character

No. of LOD Levels to Generate: 3

LOD Settings

Level	Method	Type	Value	Bones/Vert
2	MeshLOD	Percentage	65.0	4
3	MeshLOD	Screen Size	12.0	2
4	MeshLOD	Screen Size	6.0	2

Settings

Generate LODs

Idle... report a bug documentation

UBM

UI is an interesting area, because it's often been overlooked, both within technical art and in games development in general, and yet it covers an entire job role of information. In fact a huge part of technical art is about user experience - how effective are your tools and pipelines if they're not friendly to use.

So with tips for User Interfaces let's start with

- Don't re-invent.
  - Plenty of research has already gone into UX across the technology sector in how users interact with interfaces and devices, how long does a user looks at certain buttons, what's their pattern of eye movement in looking at different elements on a screen.

Take advantage of this research. We don't need to rediscover it for ourselves.

  - There are existing ways of doing things that users expect. File browsing - how does Windows do it. Right click menus. Sometimes there are even libraries to do it for you. Utilise these.

- Use visual language that the user is already familiar with. Close button top right

OK and cancel button in the same order.

Consistent casing of labels

- Does it match behaviour with the rest of the tool and with the other tools you've developed
- Does it match current software behaviour
- Does it match common software behaviour (e.g. viewport tumbling, hotkeys)
- Does it match OS behaviour
- Any behaviour that the user can understand or expect from use of other software means a reduced amount of time to learn how the tool works, less time wasted switching between using different tools and more trust in your tools through comfort of use.

Also if you have a proprietary engine, make sure the programmers maintaining those tools are in sync and are given information on expected behaviour and UI layout.

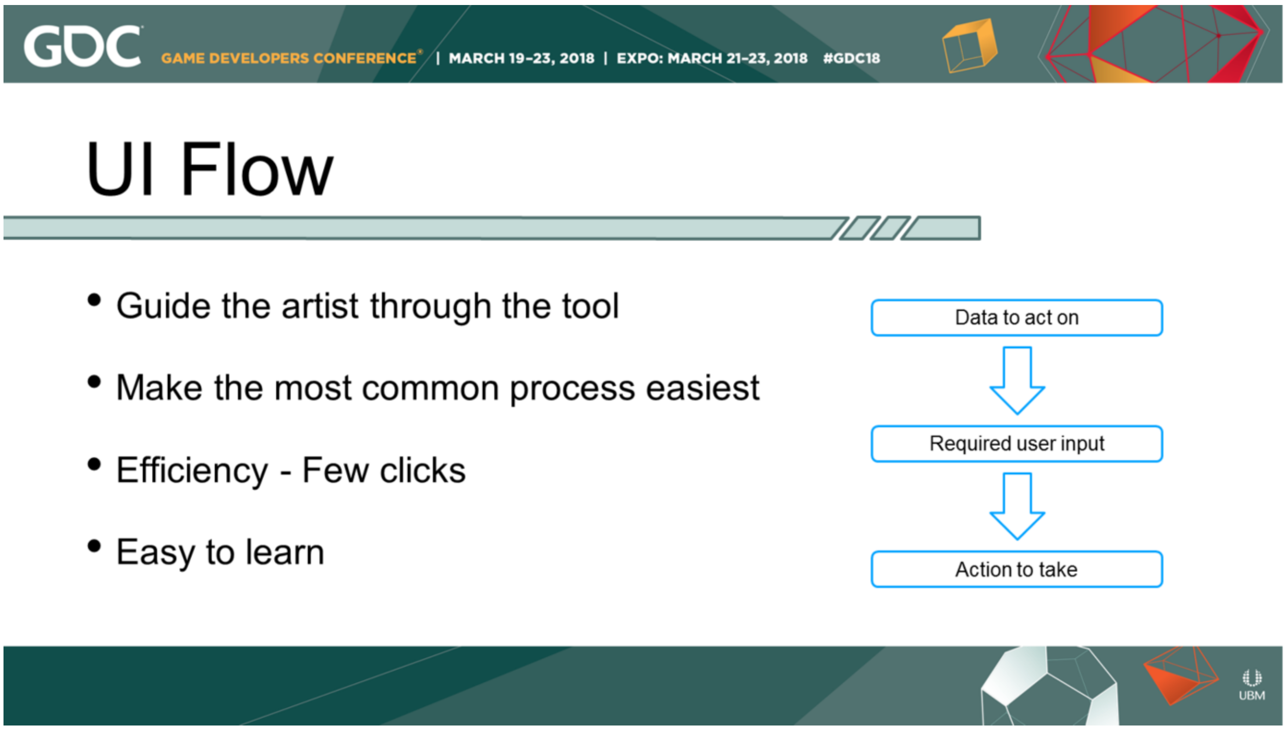
- Ensure the user can work out importance of elements

The Size of buttons and thickness indicate importance

people Notice high contrast objects first

make it clear if an input is optional or the user has to interact with it in order for the tool to function correctly - hide under advanced options

- Focus on the right users - if there is more than one user group that may use a tool in different ways make sure you are optimising for your core audience for who it will save the most amount of time. The main thing is to not get distracted with conflicting requests to have the tool laid out a certain way. Use presets or split the tool into different interfaces that both use the same backend, but are loaded separately if it makes sense.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

UI Flow

- Guide the artist through the tool
- Make the most common process easiest
- Efficiency - Few clicks
- Easy to learn

Data to act on
Required user input
Action to take

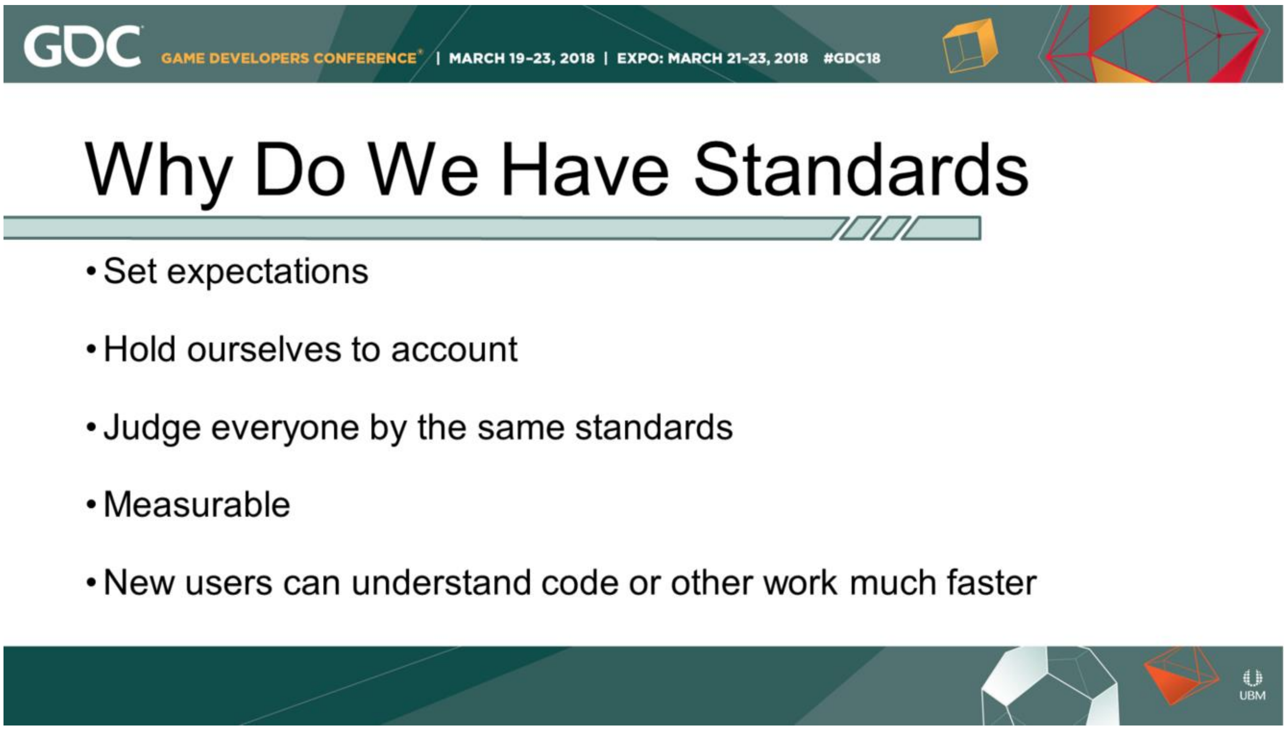
In order for users to know how to interact with the UI there is a flow to how the elements are laid out

- We want to Guide the artist through the tool. They should go through the UI from top to bottom as though they are following steps to reach the end goal of the button that actually runs the main functionality. Elements that work together should be grouped together and if certain UI elements rely on the user making a decision, the UI representing that decision should appear higher in the UI

- If your tool can accommodate different options have sensible default values so that a user can get an expected result as soon as possible. If certain options are rarely used, or are only required for advanced operations hide them under an advanced options rollout, or other way of hiding them from new users. This will keep the UI uncluttered and less scary.

- Keep the number of UI elements to a minimum so that artists can quickly work out what the tool expects. Add default values if it makes sense to, in order to reduce the number of clicks. Make it fast for users to get an expected result, they can go back later to try out different options. The fewer clicks and conscious decisions that a user has to make, the more efficient it will be for an artist to use and more comfortable they will feel using it.

- Going back to all the points on the previous slide about expected behaviour - your tool should be easy for a user to understand with only the knowledge of their job and existing tools. There’s is a level of technical knowledge expected to create game art, such as knowing the names for certain operations and features in asset creation software. But you shouldn’t expect your user to have to read documentation or decipher button labels to use a tool if it is to support something they’ve requested, or is intrinsic to them performing their job. By all means have documentation to support the tool and allow new users to find out unexpected uses or advances features of the tools, but they should be able to quickly get the tool to do something useful and not feel stupid or scared of using the tool.


Why Do We Have Standards

- Set expectations
- Hold ourselves to account
- Judge everyone by the same standards
- Measurable
- New users can understand code or other work much faster

Having standards for our work is important

- They set expectations
    - Does everyone on the team have the same goal
    - What are the high level objectives for what we develop
    - Unifying these things makes it easier for people to review each other's work

- They allow us to hold ourselves to account
    - Do you know what you should be delivering
    - Have you successfully delivered work to the agreed standards

- Everyone is judged by these same

standards

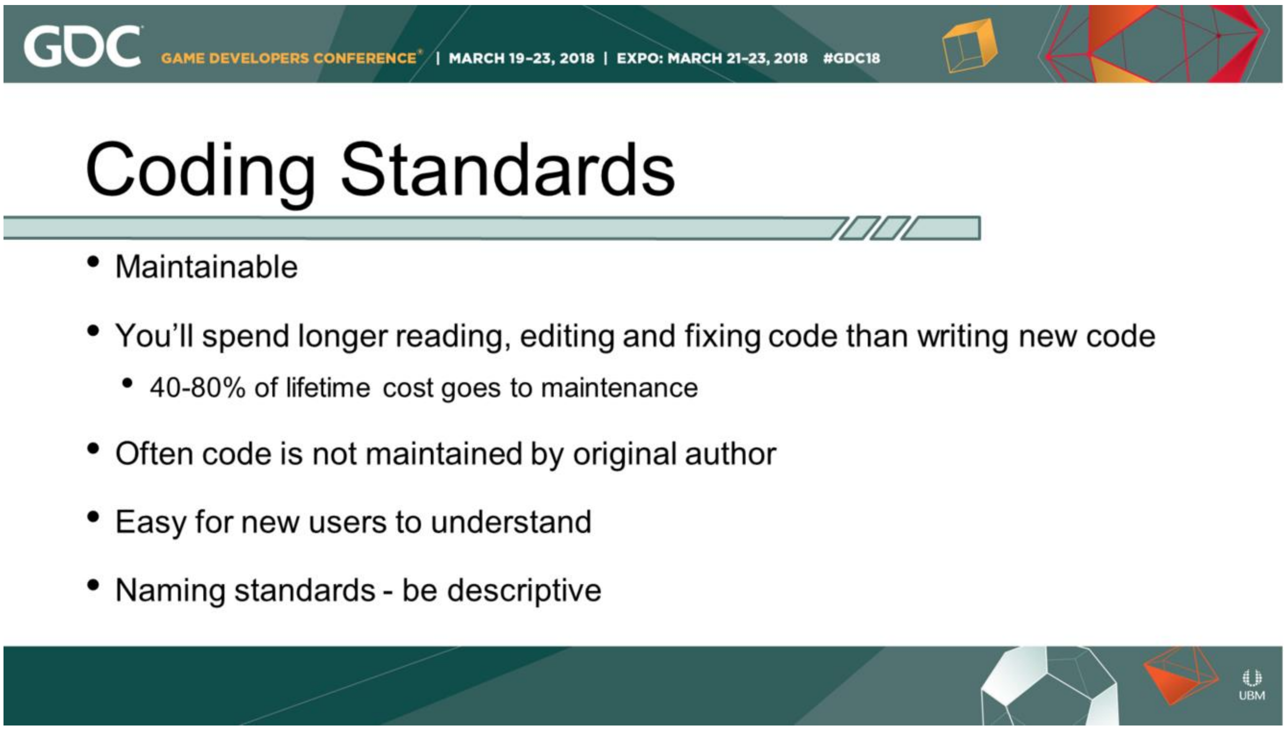
- If something new needs changing it's easier to discuss what the change should be if there's predefined set of standards to base that off of. Conversations can be less emotional, or less personal because it's clear to begin with what the standards are.

- They make it easy to measure the successfulness or completeness of a change
    - It's easier for someone to decide if something should pass review
    - And they allow us to make scripts to automate testing based on predetermined criteria, such as performance speed or expected output

- New users can understand existing code or other work much faster
    - They can work out where to look for specific things in a file or file structure based on expectations from standardisation

If you're the only TA working on a project, these standards may be the high level goals you want to achieve and allow you to be critical of what you can and can't do in the scope.

It's still important to have a set of standards when working alone as you will have to come back to your work at a later date, so leave it in good condition with things named correctly, and use your standards to help you work more predictably and measurably to achieve consistent results.


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Coding Standards

- Maintainable
- You'll spend longer reading, editing and fixing code than writing new code
  - 40-80% of lifetime cost goes to maintenance
- Often code is not maintained by original author
- Easy for new users to understand
- Naming standards - be descriptive

UBM

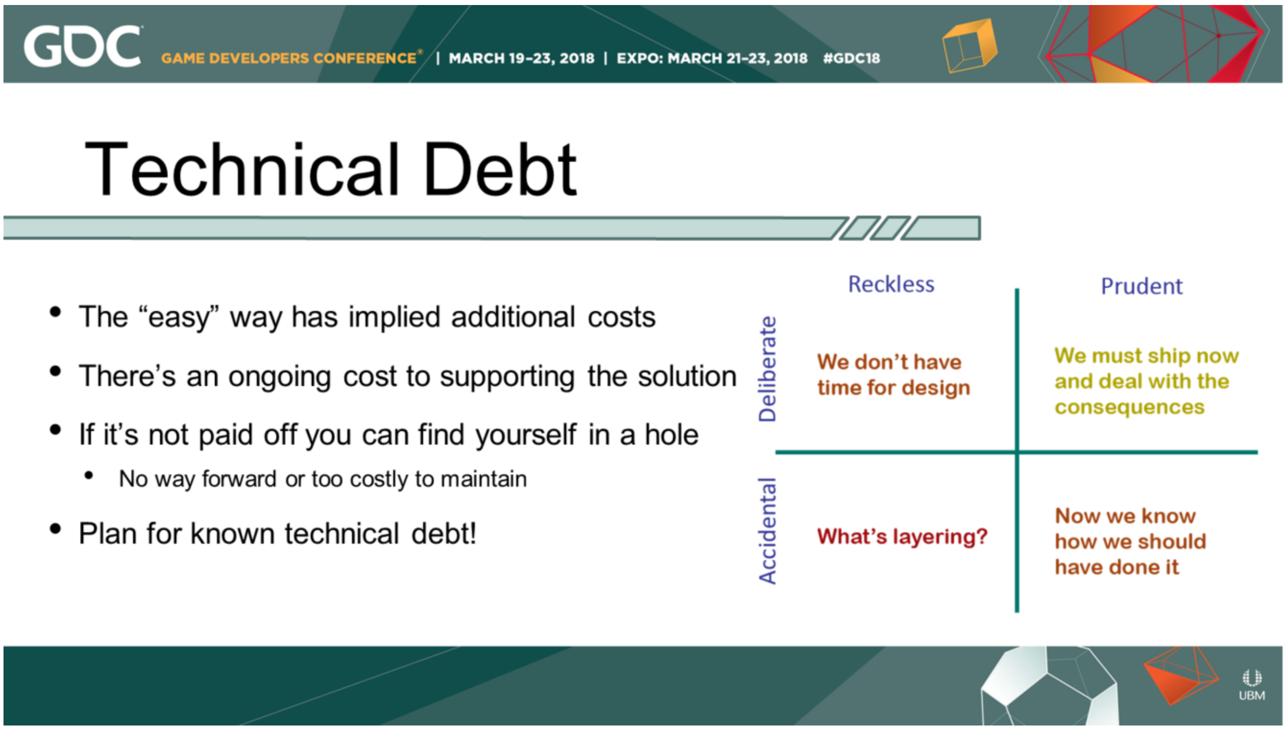
There are also specific standards that apply to code that you will write

- These help keep the code maintainable
  - You'll spend longer reading, editing and fixing code than you do writing new code
  - 40-80% of lifetime cost goes to maintenance, so make it easy to read and easy to debug
  - Often is not maintained by original author - even if it is you who will edit it, you will forget what things do over time

- again It makes it easier for new users to understand, so there's a shorter time between someone being introduced to a code base and them being an effective user of it

- Ensure code is easy to read by having naming standards -

be descriptive - we’re people, not robots so it’s faster to work out what a variable is used for with a descriptive name. You can assume what a function does and what its expected return values might be from its name. And it also helps understand what the code *should* be doing, for cases where there may be bugs and the current behaviour is not what is expected.



Technical Debt

- The "easy" way has implied additional costs
- There's an ongoing cost to supporting the solution
- If it's not paid off you can find yourself in a hole
  - No way forward or too costly to maintain
- Plan for known technical debt!

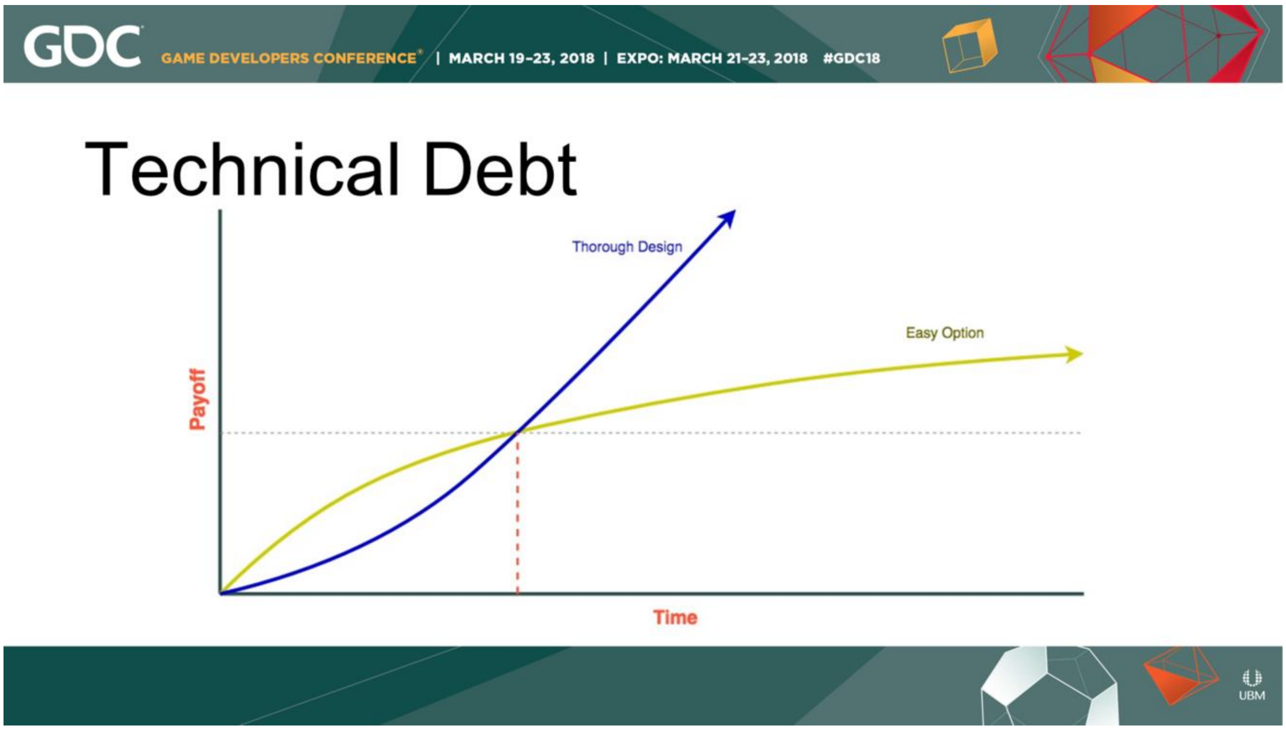
	Reckless	Prudent
Deliberate	We don't have time for design	We must ship now and deal with the consequences
Accidental	What's layering?	Now we know how we should have done it

Technical debt refers to the implied cost of additional work required to support the decision to go with an easy solution, rather than with the "correct" robust but more expensive solution to start with

This may be the cost of support to maintain the code; to get existing features to work with new data or projects; or to add new features.

It's not just the cost of what you were not able to implement at the time, but what you will end up spending in terms of time and resources on the problem in the long run as building on the easy solution is harder to implement and takes longer.

Technical Debt is unsecured short-term debt. There's no collateral if it doesn't get paid off so you're stuck in a hole.

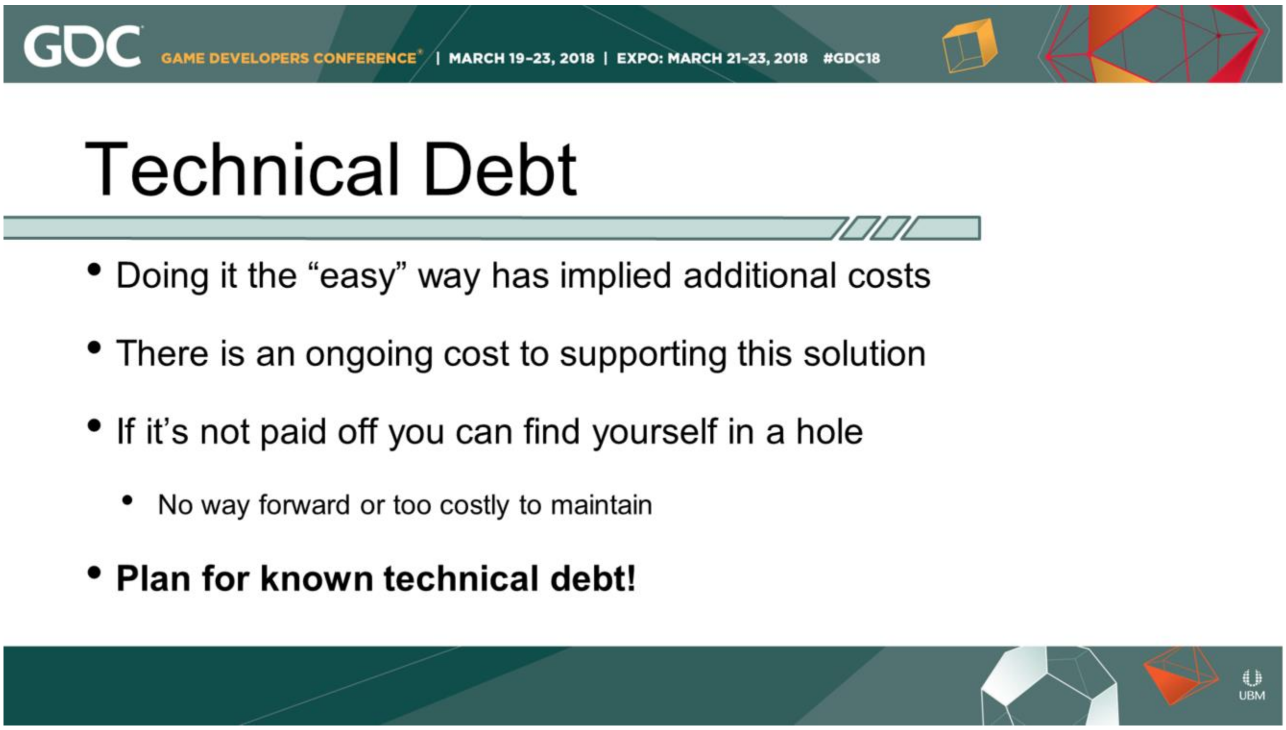


Technical Debt

It’s still important to be objective about when’s the right time is to get things done.

Sometimes a thorough solution is not necessary, or we are unable to prove that it will be required past a certain point. Finishing a project is a priority, so that the company can pay all its employees, so sometimes we just don’t have the resources to build the robust solution. And if we can’t afford to make the next project then it’s a waste investing extra resources into a “correct” solution anyway

On the flip side of that long term success can’t be constantly put off for short term gains, otherwise we end up with this technical debt that we can’t pay off and realise that we would have saved a disproportionate amount of time if we’d done it right to begin with.

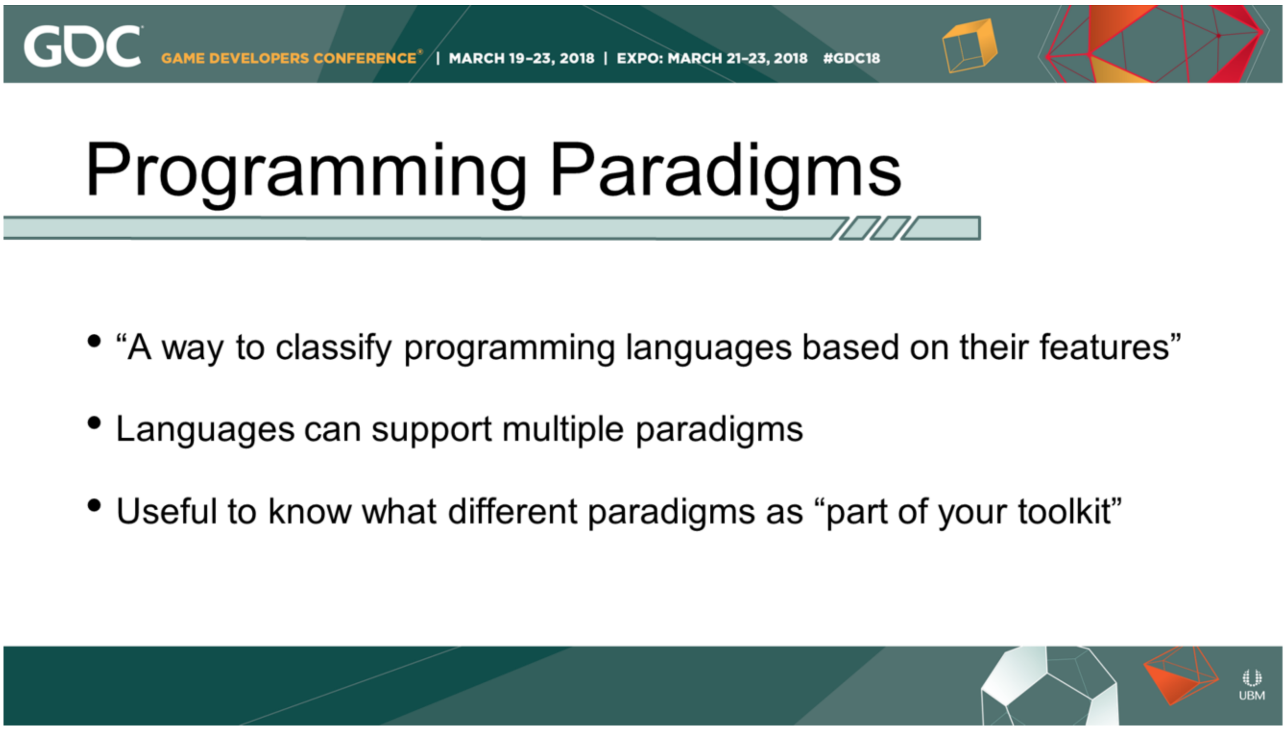


Technical Debt

- Doing it the “easy” way has implied additional costs
- There is an ongoing cost to supporting this solution
- If it’s not paid off you can find yourself in a hole
  - No way forward or too costly to maintain
- **Plan for known technical debt!**

It’s easy to know in hindsight and you’re not always the person who can make the final call.

Importantly if you need to accept technical debt, know about it and schedule for it - make informed decisions based on the fact that you know you will have to pay the cost at a later date.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Programming Paradigms

- “A way to classify programming languages based on their features”
- Languages can support multiple paradigms
- Useful to know what different paradigms as “part of your toolkit”

UBM

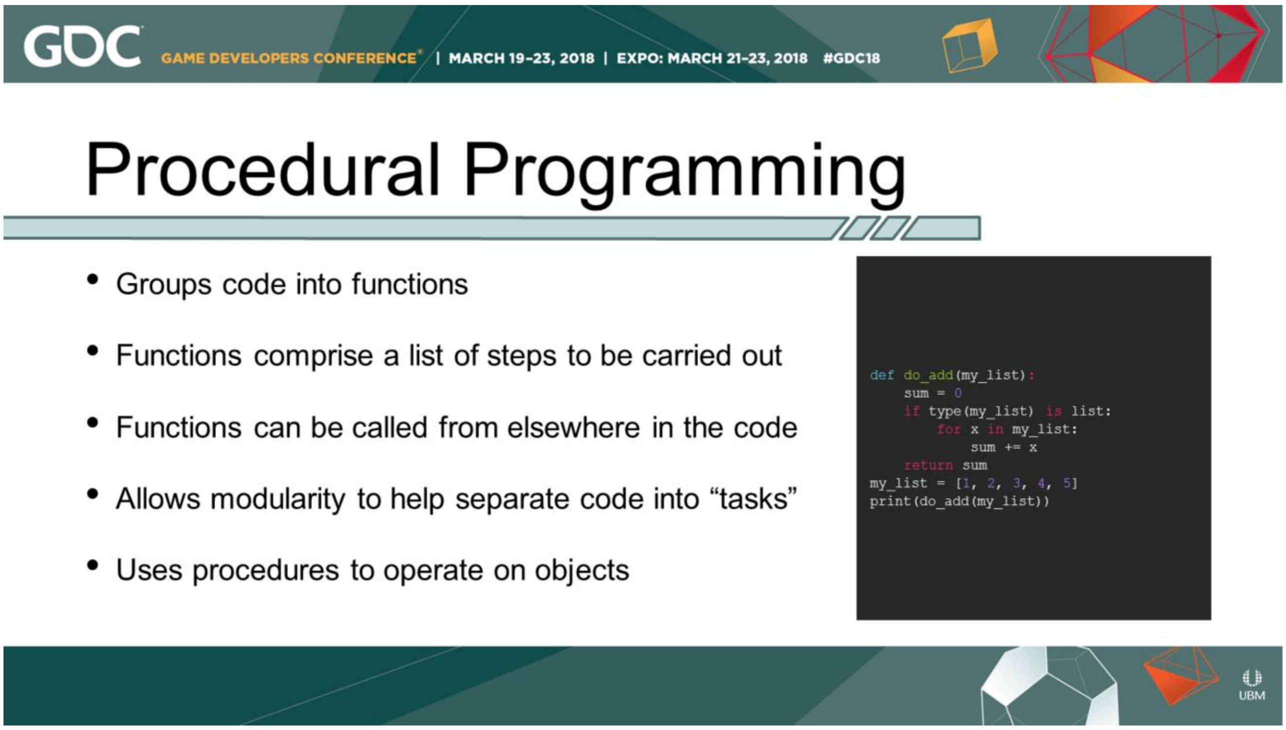
As technical artists we won't often need to use or explore low-level concepts of programming. Most of us will be working with a limited number of languages and will use a fraction of what they’re capable of in order to get what we need to work. In a way we’re quite lucky because if we do need to use non-software specific languages there’s a wealth of programmers out there who’ve filled StackOverflow with all our language specific problems

What is useful for technical artists is understanding fundamental principles that help us to think about how we transfer a theoretical problem into code, how we structure our code, how to separate syntax from general programming practices and implementations to make us adaptable to any similar language.

Paradigms are not mystical, or an aspiration of which we seek to use more complex paradigms. They are a tool.

There are many programming paradigms, but I’ll cover three that you're likely to encounter. Understand what they do in order to better understand a particular language and the benefits of structuring your code a particular way, or taking advantage of a particular paradigm, but don’t get hung up on enforcing something that sounds good, if it makes the overall code less clear. The end result is what we want to focus on, the code is the tool to get there.

Weigh up the pros and cons of using a language based on the task it needs to achieve, but also on who will be maintaining the code. Are they familiar with the language, is there a cost to training, is it easy to hire for - remember we’re not programmers, it is only part of our role.


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Procedural Programming

- Groups code into functions
- Functions comprise a list of steps to be carried out
- Functions can be called from elsewhere in the code
- Allows modularity to help separate code into "tasks"
- Uses procedures to operate on objects

```
def do_add(my_list):
    sum = 0
    if type(my_list) is list:
        for x in my_list:
            sum += x
    return sum
my_list = [1, 2, 3, 4, 5]
print(do_add(my_list))
```

UBM

This is probably the first paradigm that we started using when we were learning how to code as technical artists .

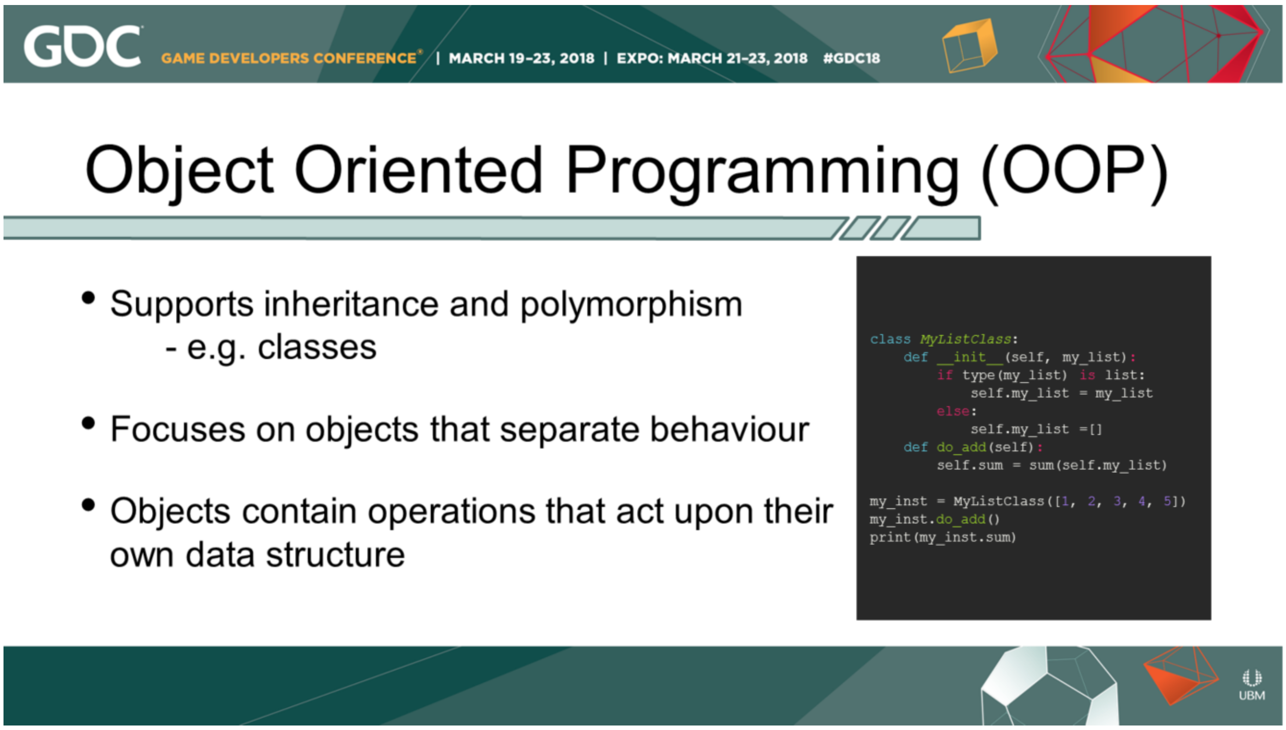
After the initial wonderment of just writing lines of code that do things in a scripting language, and realising we have all this power at our fingertips, we start grouping our code into functions that do specific things.

These functions describe the steps we want to be carried out, and then we can call the functions from elsewhere in the code.

Procedural programming allows us to be more structured than just writing everything we want the code to do in a long list. We can start re-using code, but everything is still relatively simple and straight forward

Mel in Maya is a procedural language.

n.b. procedural is a type of imperative programming



Object Oriented Programming (OOP)

- Supports inheritance and polymorphism
  - e.g. classes
- Focuses on objects that separate behaviour
- Objects contain operations that act upon their own data structure

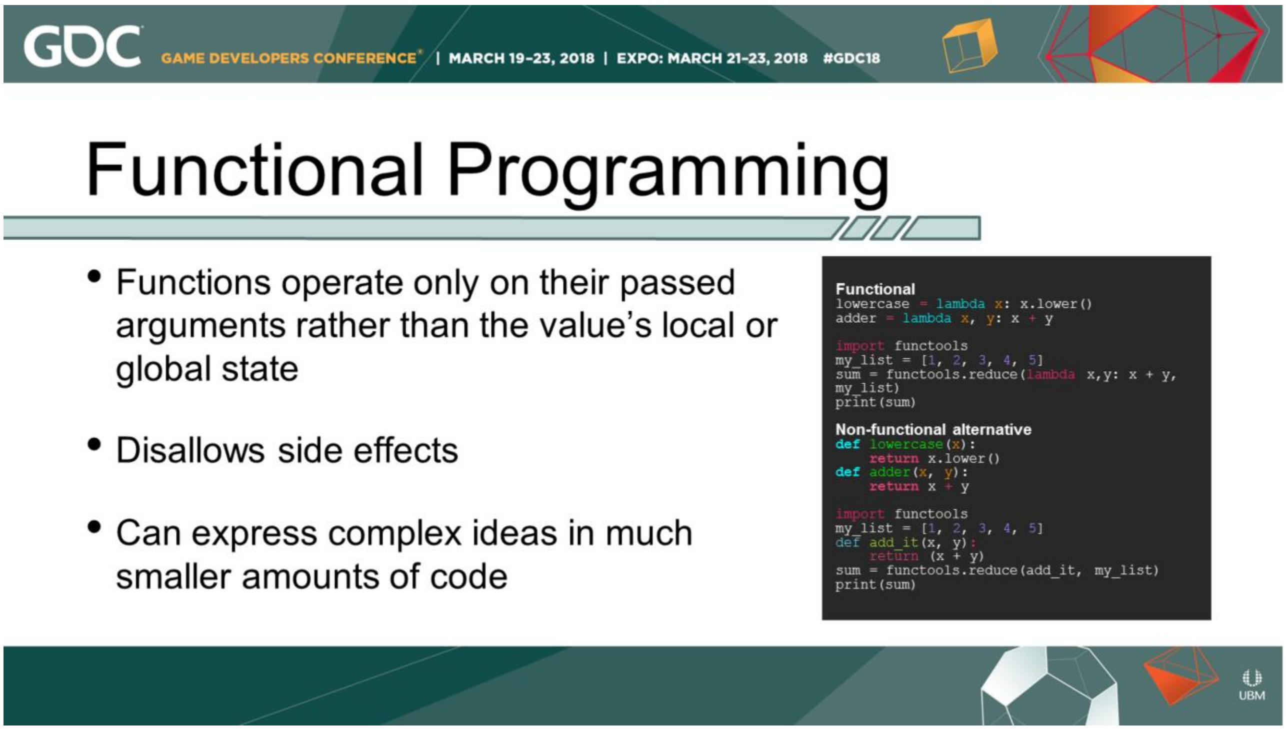
```python
class MyListClass:
    def __init__(self, my_list):
        if type(my_list) is list:
            self.my_list = my_list
        else:
            self.my_list =[]
    def do_add(self):
        self.sum = sum(self.my_list)

my_inst = MyListClass([1, 2, 3, 4, 5])
my_inst.do_add()
print(my_inst.sum)
```

Object Oriented Programming is probably the most well known paradigm by technical artists. Once you realise the limitations of procedural programming the re-usability of object orientation is very appealing.

It allows us to separate code into objects. This helps us structure the code and objects can have both data and methods belonging to them. This allows objects to call procedures on themselves.

It also supports inheritance and polymorphism, so methods can be inherited from parent classes but can handle different data types being passed to them and can handle the data differently by overriding the methods

python, C# & C++ support object oriented programming


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Functional Programming

- Functions operate only on their passed arguments rather than the value's local or global state
- Disallows side effects
- Can express complex ideas in much smaller amounts of code

Functional
```
lowercase = lambda x: x.lower()
adder = lambda x, y: x + y

import functools
my_list = [1, 2, 3, 4, 5]
sum = functools.reduce(lambda x,y: x + y,
my_list)
print(sum)
```

Non-functional alternative
```
def lowercase(x):
    return x.lower()
def adder(x, y):
    return x + y

import functools
my_list = [1, 2, 3, 4, 5]
def add_it(x, y):
    return (x + y)
sum = functools.reduce(add_it, my_list)
print(sum)
```

UBM

Functional programming is not widely used in Technical Art. However, there is some implementation of it in python so you may come across it or want to investigate it further

Most common use that you're likely to have seen is the use of the lambda expression.

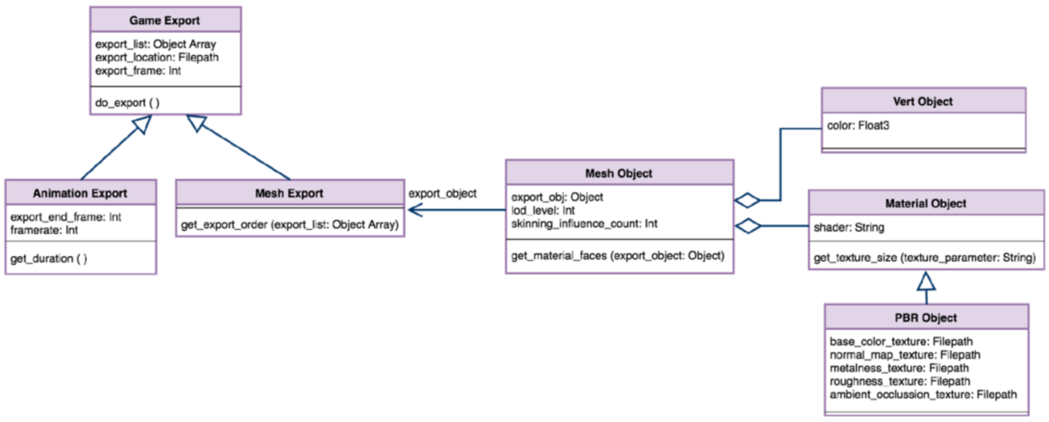
Functional programming allows us to express complex ideas in much smaller amount of code, which can be quite appealing. It makes code easier to debug and test as functions are generally small and specific.

It also disallows side effects - the output of an expression relies only on the arguments it's passed, rather than local or global state of values, which can make it easy to debug

However, writing code in a such a succinct way can make it less readable so this slide can be seen as a warning incase you come across it, to remember to keep readability a priority. Not just for yourself, but for other members of the team who may not be as familiar with a code base or way of writing code.

n.b. functional is a type of declarative programming

Unified Modelling Language (UML)


Game Export
export_list: Object Array
export_location: Filepath
export_frame: Int
do_export ( )
Animation Export
export_end_frame: Int
framerate: Int
get_duration ( )
Mesh Export
get_export_order (export_list: Object Array)
export_object
Mesh Object
export_obj: Object
lod_level: Int
skinning_influence_count: Int
get_material_faces (export_object: Object)
Vert Object
color: Float3
Material Object
shader: String
get_texture_size (texture_parameter: String)
PBR Object
base_color_texture: Filepath
normal_map_texture: Filepath
metalness_texture: Filepath
roughness_texture: Filepath
ambient_occlussion_texture: Filepath

UML is a standardised graphical language for visualising a system.

It covers different types of diagrams representing Structure and Behaviour and allows you to visualize the system in a diagram, representing

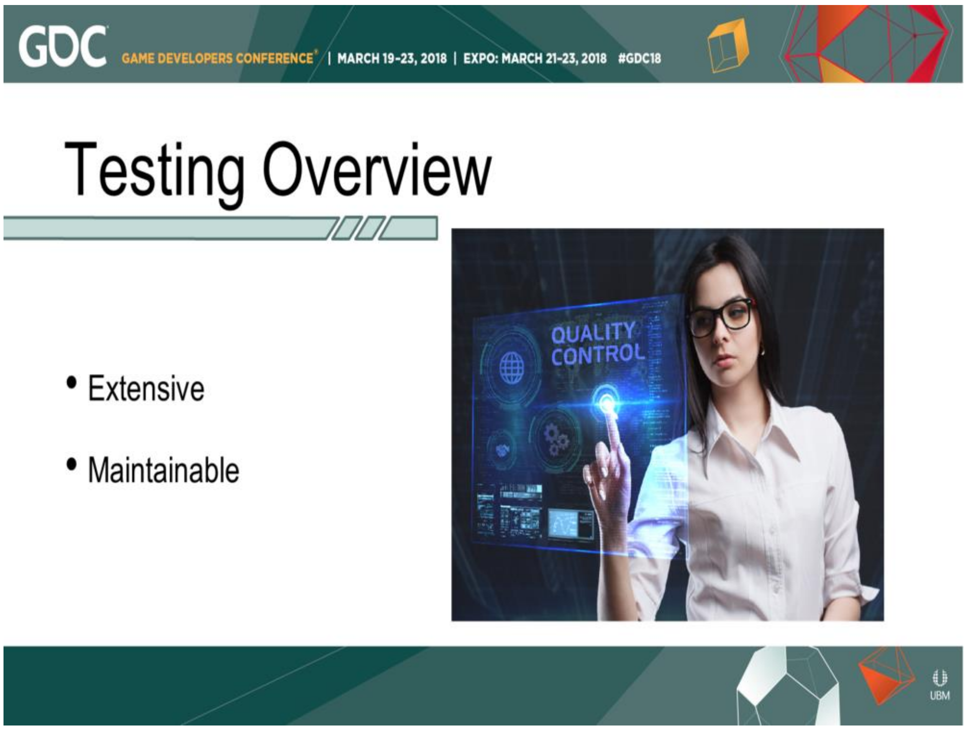
- any actions and activities
- individual components, such as classes attributes and methods
- and how they can interact with each other
- how the system will run and the flow of data
- Relationships between entities
- and the

external user interface.

Visualising the system in this way helps you to understand how the data breaks down and the relationship between different data models - you can visualise how you’re planning to design the system and work out the software flow you want to achieve before you start writing any actual code

I love UML because it’s a formal way of getting yourself to fully understand the problem in enough detail before jumping into the code, and a chance to do something specific that informs the code but isn’t writing code.

You can use computer programs to create UML diagrams or draw it on paper.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Testing Overview

- Extensive
- Maintainable

UBM

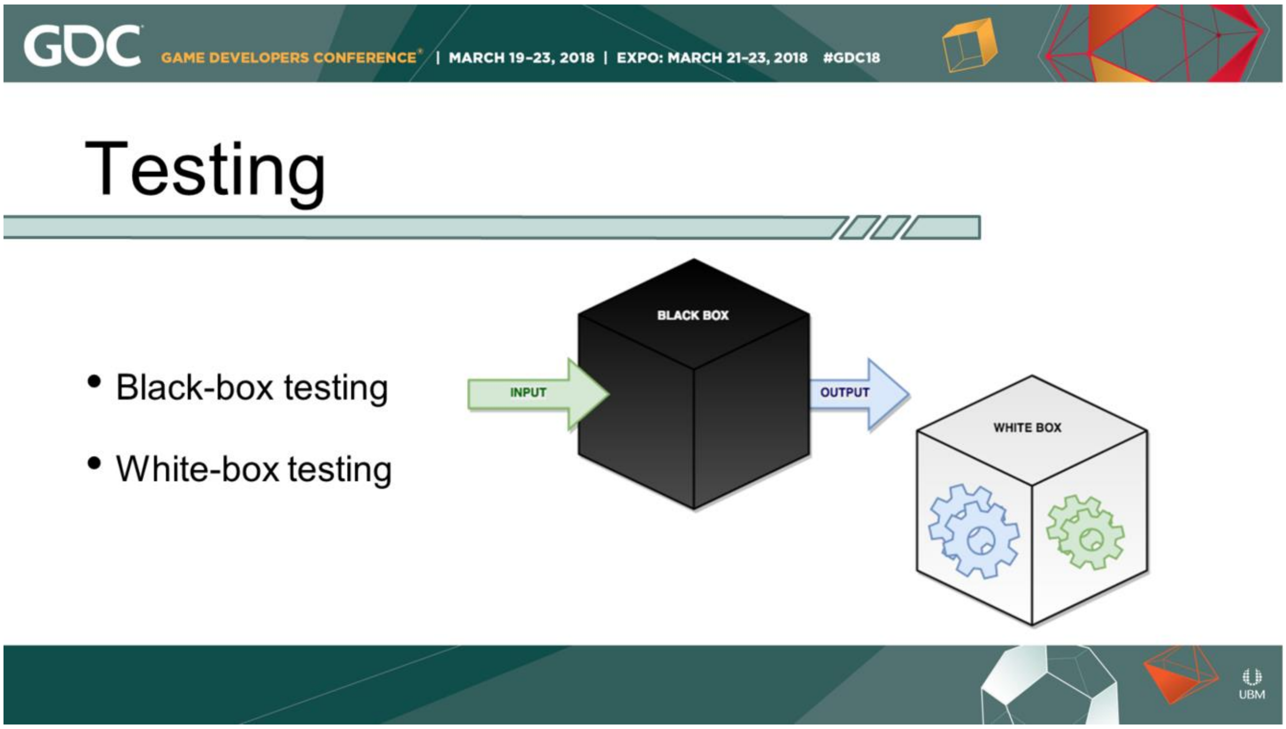
Testing is a key part of ensuring we are providing useful solutions that meet the requirements of the end user. Also that our solutions are reliable and that artists can trust in what we provide them.

Tests should be

1. Extensive - cover that it behaves as expected with a wide range of expected data; fails gracefully with unexpected data. Is performant with different sizes of data.
2. Maintainable - we’re not spending a disproportionate amount of time testing; we can reliably cover the testing in a reasonable amount of time; the tests can cover our needs if the requirements change.

Help yourself by having a test plan. This may be a checklist of things to test or a list of files commonly used with a particular tool. This is especially important if your tools support multiple projects.

Make sure this is accessible to all members of the team.


GDC GAME DEVELOPERS CONFERENCE | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18
BLACK BOX
INPUT
OUTPUT
WHITE BOX
UBM

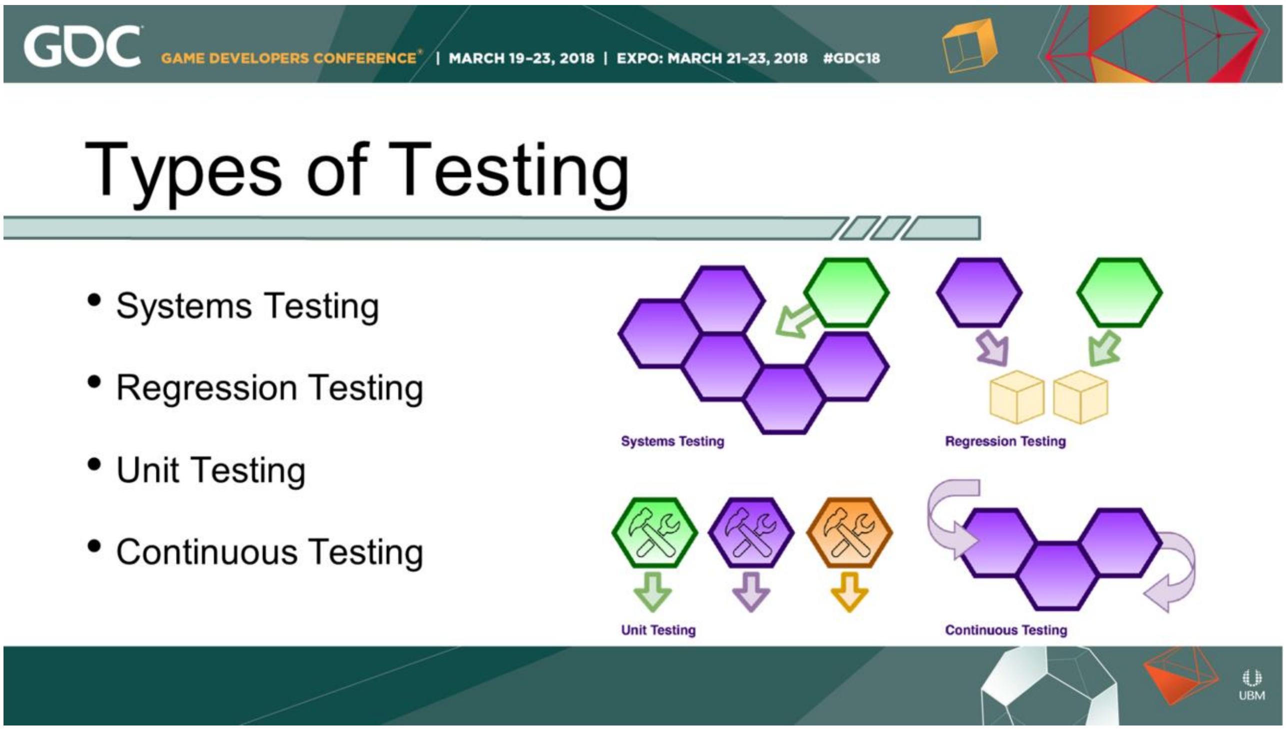
Testing

- Black-box testing
- White-box testing

As technical artists we often need to test both functionality and the implementation, which is often code. But testing shouldn’t be restricted to just code solutions. Anything that you are responsible for providing to artists should have an appropriate level of scrutiny applied to it before it reaches their hands, to ensure that it fulfils requirements and engenders trust between you.

Black-box testing tests the behaviours, or functionality without looking into the inner workings. Because you don’t need a technical understanding of how the problem was solved or its inner workings, this can be done by any end user or team member who knows what the expected output should be with known inputs.

White box testing scrutinises the inner workings of the solution. You can test the control flow and data flow of the software as well as statement and decision coverage. The intension is to find and prevent hidden errors later on. You want your code to be an error free environment, so white box tests allow you to determine whether particular functions or lines of code are passing back the correct data.


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Types of Testing

- Systems Testing
- Regression Testing
- Unit Testing
- Continuous Testing

Systems Testing
Regression Testing
Unit Testing
Continuous Testing
UBM

These testing methods see practical application in the following types of testing and many others. There are plenty of other categories of testing, but I'll give a few details on these four as they cover a lot of the main issues that we notice as TAs and that we may want to focus on or be aware of when writing test plans.

Systems Testing

Tests the whole system - including design, behaviour and expectations of the client.

You can look at

- usability
- interface
- performance
- compatibility
- load testing and scalability

Regression Testing - falls within systems testing

It focuses on finding errors after, usually a significant, code change has been made

Do we still get the expected outcome after changes have been made - do areas that should not be affected by the change have an identical outcome after the change.

We find this out by comparing the result pre change and post change. This could be comparing a list of data outputs or comparing a before and after image.

During this testing we want to find if we have lost any behaviour or old bugs have come back - this can especially be the case if you have multiple people working on a code base and changes are being worked on concurrently.

Unit testing is a method by which you test individual units of code to determine if they meet the design and behaviour expectations and are fit for use.

A unit could be an entire class, or it could be a single method. But each test is written to run independently

They are appealing as they catch more bugs during development of code, leading to a robust solution

However, success is based on thoroughness and creating extensive unit tests takes time, which may not be maintainable.

Especially in a Technical Art environment where we need to be very reactive

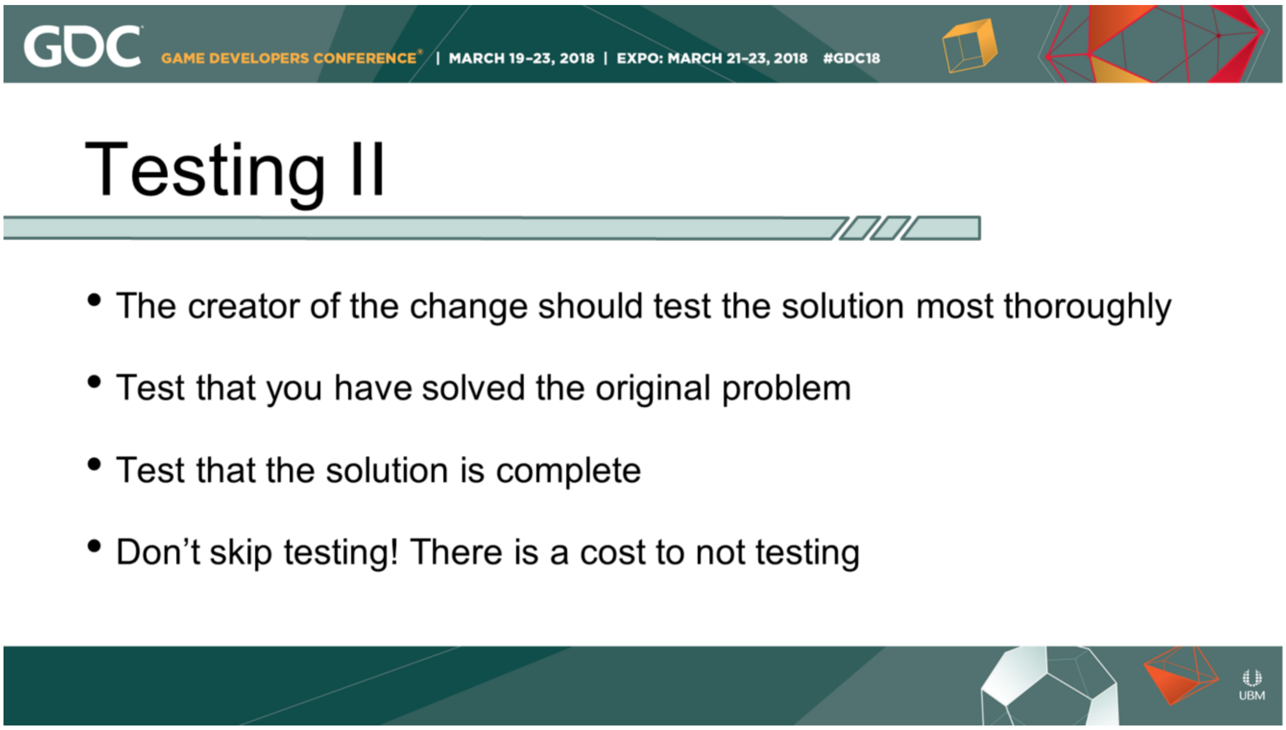
It can also lead to lower satisfaction for you - as you will spend a lot of time writing tests

However, using them can be applicable to some areas - it makes more sense to write unit tests that test particular python libraries that you know will form a core base for a lot of tools and have a long lifetime. The unit tests can be run any time that particular code changes to ensure the behaviour is expected, and because unit tests can be run independently any additional more volatile code can have a different level of testing that is easier to maintain.

Incorporating unit testing changes the entire way you develop, as you write tests before you even write the problem solving code, so if you’re interested in implementing it speak to programmers about why they do or don’t use this, and whether they think it’s suitable for a particular system or area

Continuous Testing is the process of using automated tests to give fast and continuous feedback on issues. This makes it easier to assess the riskiness of submitting a solution and provides higher quality submissions through finding bugs much closer to when they first occur.

Like with unit testing there are issues with the usefulness of implementing this, due to having to maintain this testing method in order for it to effective. Continuous testing utilises unit tests to perform operations, so is only as useful as the thoroughness of the testing code written



Testing II

- The creator of the change should test the solution most thoroughly
- Test that you have solved the original problem
- Test that the solution is complete
- Don’t skip testing! There is a cost to not testing

So taking it back to a higher level overview

The onus is on the author to do the majority of testing, since they will have a much faster turn around time of finding and fixing issues. So don’t avoid thoroughly testing your own work because you believe issues will get picked up in the testing process. More time is wasted passing changes back and forth than would be running thorough testing. By the time a problem reaches the end user they may be unsure if it is a bug they’re encountering or if they’re using the solution incorrectly.

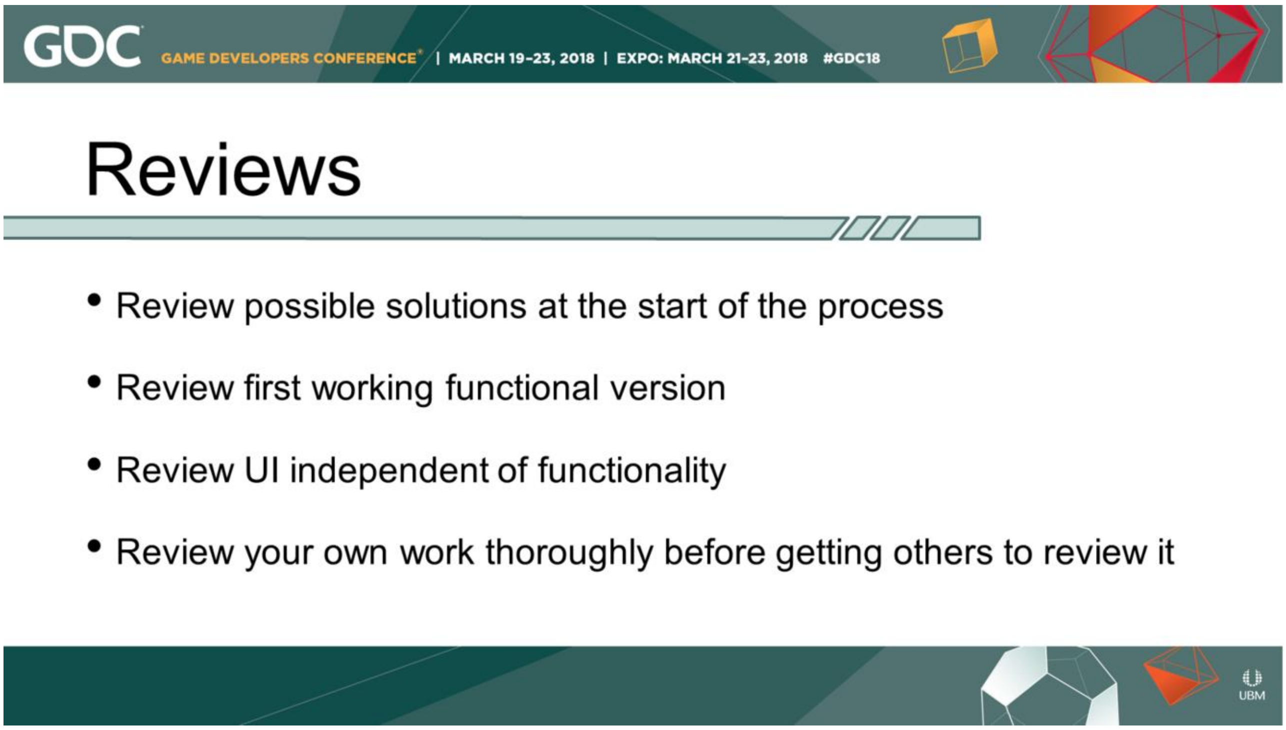
Have you solved the original problem. Are you testing the change within the same environment as the user?

Is the solution complete? Have you tested it on a range of expected and unexpected date to ensure it behaves correctly and handles exceptions.

And if you think you don’t have time to do thorough testing think of the cost of not testing

The closer to the source of fixing a problem a bug is found, the faster it will be to fix and the less impact it will have on productivity. The earlier issues are found, the more likely you will remember how the code works and where the likely cause of error is.

There is a cost to not testing, so I will reiterate make your testing extensive and maintainable.


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Reviews

- Review possible solutions at the start of the process
- Review first working functional version
- Review UI independent of functionality
- Review your own work thoroughly before getting others to review it

UBM

The review process is important because it helps you determine whether you're still solving the original problem, and that it's the right solution for the end users.

Reviews can happen at any time in the development process and its important to not just leave it until you've finished creating the entire solution to review what you're working on, as especially when creating a large tool or developing a complex solution, you are at risk at any point of diverging or getting off track.

Reviews are not just limited to reviewing the code. You should review any problem to make sure you are not making assumptions and are solving the actual problem.

Each problem will demand a different level scrutiny in reviewing the solutions. There's no point in over analysing a task where your objective is to turn 10 button clicks into one.

Review possible solutions at the start of the process.

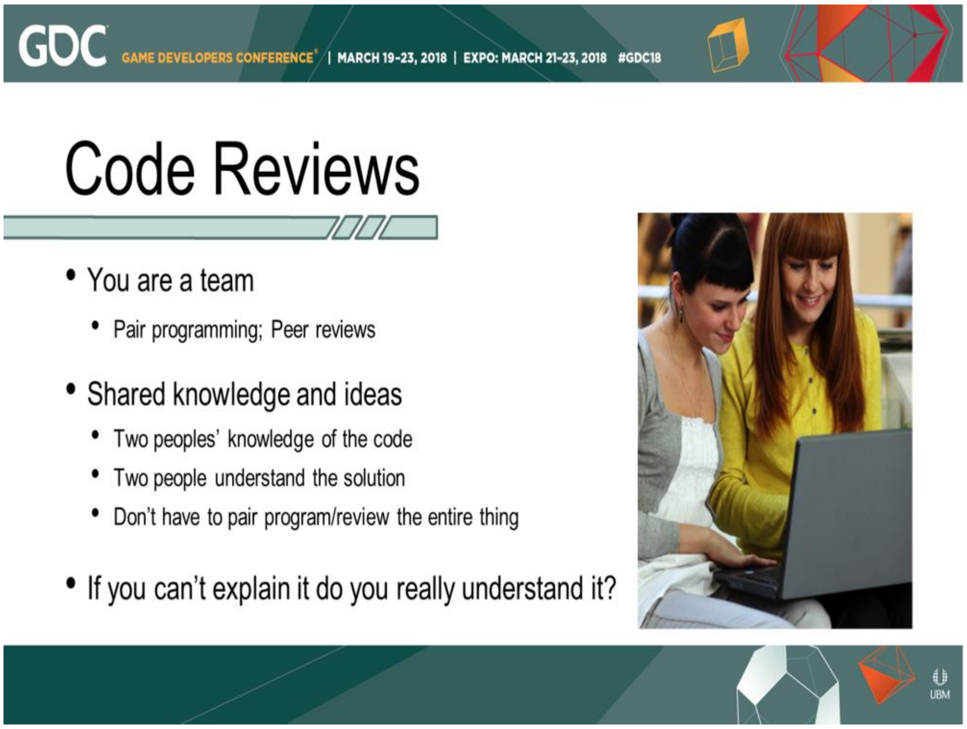
Test early - get your first working version to artists as quickly as possible and Review any interfaces independently from the functionality to ensure

- it solves the original problem
- Is user friendly
- and works within the rest of their pipeline

Review your own work before putting it up for review - which was covered by the slides on testing

Learn to be self critical

One of the simplest things you can do whether you are a lone TA or in a team - diff. your changes before submitting or putting up for a review. Use this to check for unnecessary debug output, bad names, inconsistencies or any other unnecessary changes.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Code Reviews

- You are a team
  - Pair programming; Peer reviews
- Shared knowledge and ideas
  - Two peoples' knowledge of the code
  - Two people understand the solution
  - Don't have to pair program/review the entire thing
- If you can't explain it do you really understand it?

UBM

For code changes we can have a very standardised process and code reviews are something that many programmers have done for a long time, so we can learn from what works from them. You may already have a formal code review process in place where you work that you can incorporate or learn from.

However, even among programmers there are still these false ideas such as

- It often being seen as a speed bump to getting "dev complete"
- Only done by senior members

It's important to realise that the author and the reviewer are a team.

It's not a process of finding flaws in the author - it's a process of developing a solution of higher quality than one could achieve alone

Code reviews are there to spot bugs - but that’s not the most important thing

Reviews help ensure code quality and maintainability

They also allow other people understand the code changes

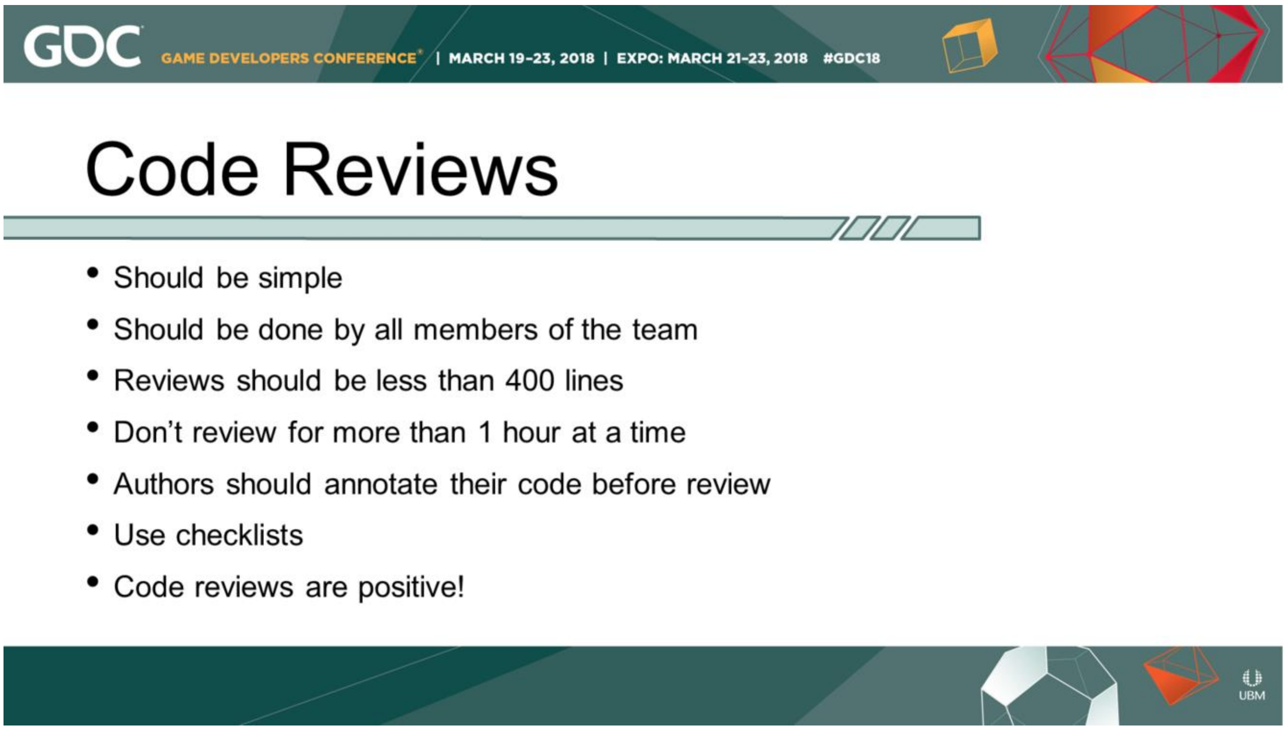
Github and Perforce have their own inbuilt tools for doing in code reviews - they allow you to leave comments on the code, see other peoples’ comments and see previous versions of the requested changes

There are other independent review softwares you can use

But don’t forget the power of human interaction. During the process of writing the code you can pair program. Write the code with another member of the team so you can talk through how you intend to implement the solution, discuss options, and share knowledge of the existing code base and the language being used. You can pair program only a portion of the development too.

During the review process you can also do face to face peer reviews. This gives you an opportunity to discuss the changes in person, rather than just have someone read your code alone. And when you have to explain what your code does and why, it enforces that you understand it and may pick up on small mistakes by explaining it out loud.

http://davidbolton.net/blog/2014/06/06/code-reviewing/


GAME DEVELOPERS CONFERENCE | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Code Reviews

- Should be simple
- Should be done by all members of the team
- Reviews should be less than 400 lines
- Don't review for more than 1 hour at a time
- Authors should annotate their code before review
- Use checklists
- Code reviews are positive!

- Code reviews should be simple
- They should be performed by everybody
- Review less than 400 lines at a time
  - because your brain can only process so much information effectively in one go
- Take your time and give due diligence to the review
- But don't review for more than an hour at a time
  - otherwise the quality of the review will decreases as your concentration drops
- Authors should annotate code before review. This again enforces with themselves that they understand their own changes, and makes it easier for the reviewer to know the intended functionality and pick up on any code that may not work as intended
- Use checklists
  - Some mistakes are commonly made over and over
  - There will also be specific files or situations that are

likely to be affected by any changes so ensuring you are going through your review process in a logical and thoughtful manner helps ensure coverage of common issues and you can be more confident in the quality of the code.

- Code Reviews are positive!
    - They're a blame free environment and we should embrace the positive results they yield

In conclusion - Everything is about Balance

You will never be able to achieve everything you want to. Technology is always changing and there's always new things to learn.

But you can

- Find balance between process and end results
- and find your personal balance between art and programming

We've covered various ways of scrutinising our processes, of being more critical of how we develop and looking at ways of future proofing our work and deciding if something is worth doing to begin with. These processes shouldn't feel like an unnecessary blocker to your work by introducing more formality, but should be applied to your unique development team appropriately, in order to make it easier to deliver reliable solutions and content. This in turn should free you up from unnecessary bug fixing, helping you and your team grow together as critical thinkers and enjoy the exciting part of problem solving and improving visual quality.

Being more formal in how you approach problem solving and implementing solutions will make you more consistent, make your results more measurable and the results more predictable.

The key is applying these processes *appropriately* to make them effective. There's no point going with a test driven approach if your priority is getting results into game quickly in order to ship a project. But if you want to create pipelines that stand the test of time and can support multiple projects you may need to design your tools with a data driven solution in mind from the start.

Be critical of whether a task is worth doing or are there more effective things to be spending your time on right now.

We're game developers and shipping great games is our priority and to do that effectively we support artists to create amazing art.

If we lose sight of these goals it doesn't matter how great the tools are that we create, or how scalable and future proof the systems we implement. Objectively our value is in making it fast and reliable to create high quality art that supports the creation of an awesome game experience.