



GDC EDUCATION
SUMMIT

Teaching Designers to Code

Margaret Moser
Assistant Professor, USC Games



GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16



Hello, I'm Margaret Moser. I'm going to talk about my experiences teaching design students to code.



1. Motivation
2. Principles
3. Tools
4. Conclusions



Here's what I'll talk about – first I'll talk about my goals. Then we'll look at some research. Then I'll show some specific tools I think are helpful for beginning programmers, and discuss why.



GDC EDUCATION
SUMMIT

Motivation



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16



GDC EDUCATION
SUMMIT

Programming is the last
mile of game design.

Jonathan Blow

Game designers must
learn to program.

Chris Hecker

Everybody in this country
should learn how to
program a computer.

Steve Jobs



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

There was this controversy, from a couple of years ago. Do designers need to code? It's an important question.



But that's not really my motivation. For me, it's personal.



The Dark Ages: CS Curriculum

Semester	Content
Semester 1	Scheme (a Lisp variant) functional programming (handout for Unix shell)
Semester 2	algorithms data structures (handouts for C++, emacs, gcc)

So this was the introductory CS curriculum when I started college. Where I have “handout”, I mean that they literally gave us a paper handout. Sometimes they also went over it briefly in class.

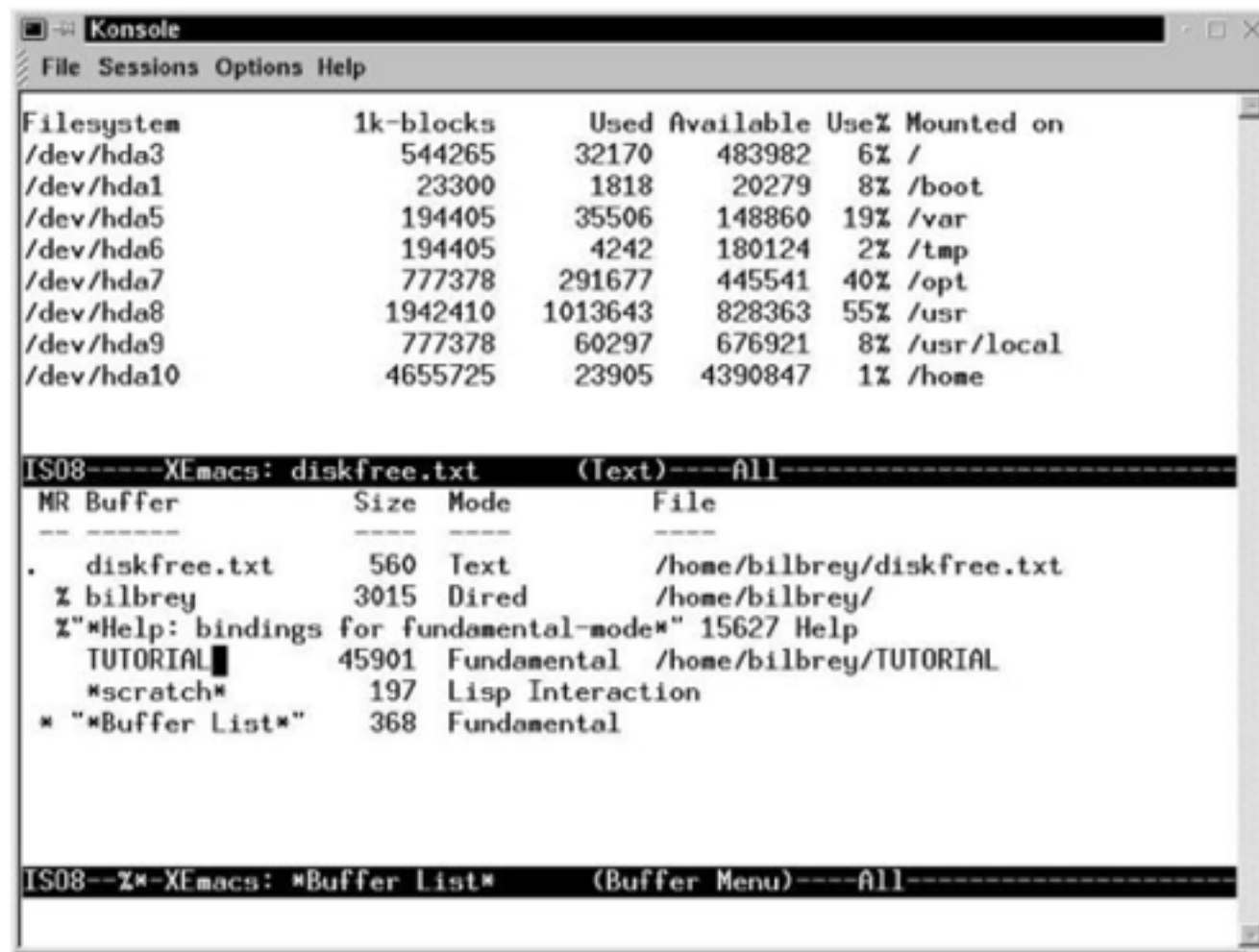
They gave us powerful tools though!



Computers hewn from



the latest stone



The screenshot shows a terminal window titled 'Konsole' with a menu bar containing 'File', 'Sessions', 'Options', and 'Help'. The main content is divided into two sections. The top section displays disk usage for various filesystems, and the bottom section shows the Emacs buffer list.

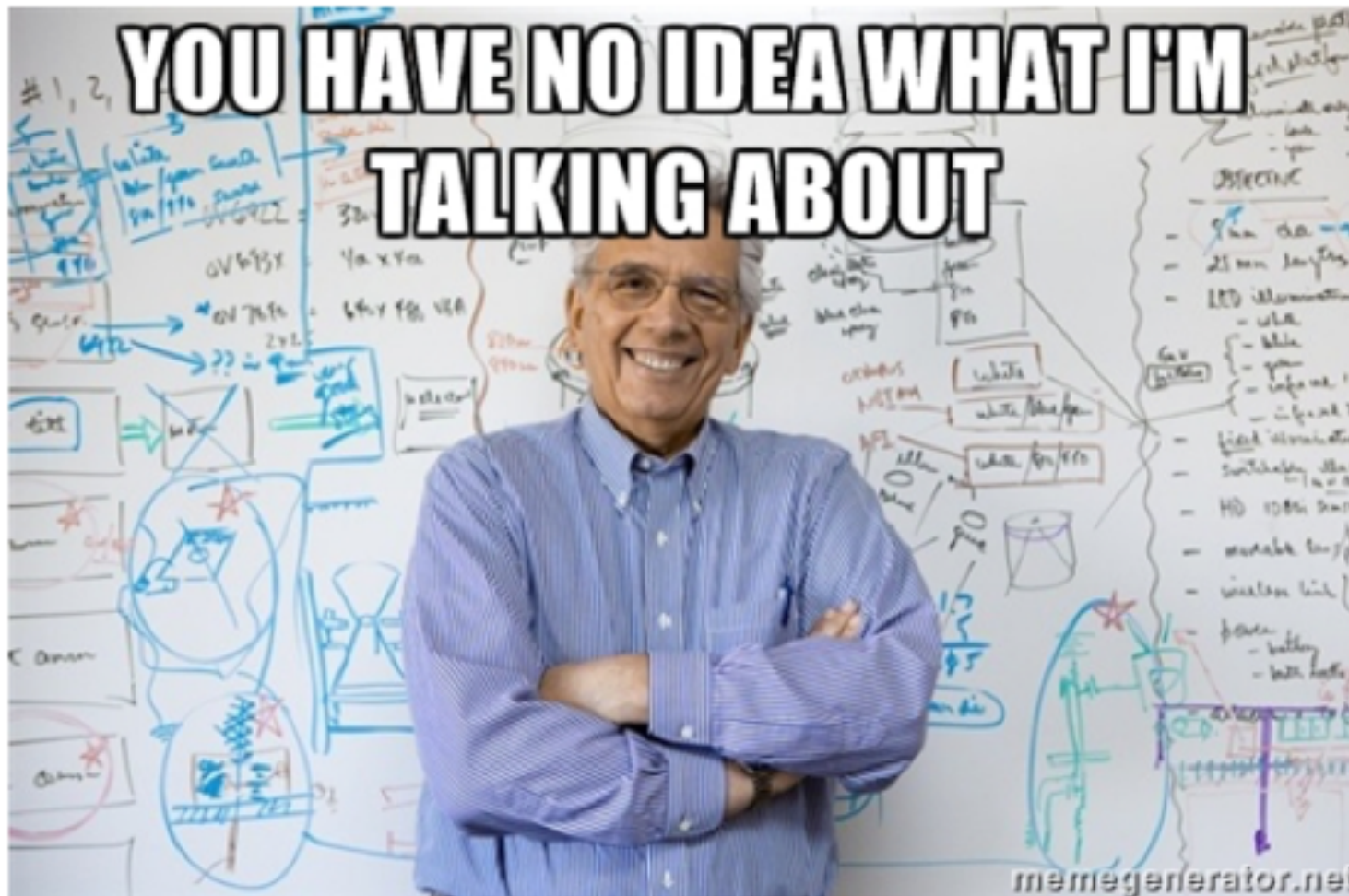
Filesystem	1k-blocks	Used	Available	Use%	Mounted on
/dev/hda3	544265	32170	483982	6%	/
/dev/hda1	23300	1818	20279	8%	/boot
/dev/hda5	194405	35506	148860	19%	/var
/dev/hda6	194405	4242	180124	2%	/tmp
/dev/hda7	777378	291677	445541	40%	/opt
/dev/hda8	1942410	1013643	828363	55%	/usr
/dev/hda9	777378	60297	676921	8%	/usr/local
/dev/hda10	4655725	23905	4390847	1%	/home

MR Buffer	Size	Mode	File
. diskfree.txt	560	Text	/home/bilbrey/diskfree.txt
% bilbrey	3015	Dired	/home/bilbrey/
% *Help: bindings for fundamental-mode*	15627	Help	
TUTORIAL	45901	Fundamental	/home/bilbrey/TUTORIAL
scratch	197	Lisp Interaction	
* *Buffer List*	368	Fundamental	

And emacs. Emacs had autocomplete... if you typed a bracket or paren it would provide the other one!

This screen shot shows menus, but we didn't have menus. So to use it, you had to know a lot of arcane key combinations – like even to save a file. I mean buffer.

I did okay learning the Unix command line and emacs and I even enjoyed functional programming. But then



We got to data structures and algorithms in C++, and I couldn't seem to keep up. Nobody thought that class was easy, but somehow they were getting traction and I wasn't. I didn't get it – I got the highest score in AP Computer Science; the only prerequisite was the Scheme class where I'd gotten an A.


```

from /home/kaiser/develop/testclang/main.cpp:3:
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable_policy.h: In member function 'size_t st
ed hash, false>::M_hash_code(const Key&) const [with Key = A, Value = std::pair<const A, int>, ExtractKey = std::Select
shing]':
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable:980: instantiated from 'std::pair<typ
e hash code, __constant_iterators, __unique_keys>::iterator, bool> std::tr1::Hashtable<Key, Value, Allocator, ExtractKey
st Value&, std::tr1::true_type) [with Key = A, Value = std::pair<const A, int>, Allocator = std::allocator<std::pair<cons
_H2 = std::tr1::__detail::Mod_range_hashing, Hash = std::tr1::__detail::Default_ranged_hash, RehashPolicy = std::tr1::__d
ys = true]':
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable:420: instantiated from 'typename __gr
erators, __cache_hash_code>, bool>, std::tr1::__detail::Hashtable_iterator<Value, __constant_iterators, __cache_hash_code>
cache_hash_code, __constant_iterators, __unique_keys>::insert(const Value&) [with Key = A, Value = std::pair<const A, int>
, Equal = std::equal_to<A>, H1 = AHasher, H2 = std::tr1::__detail::Mod_range_hashing, Hash = std::tr1::__detail::Defaul
constant_iterators = false, bool __unique_keys = true]':
/home/kaiser/develop/testclang/main.cpp:76: instantiated from here
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable_policy.h:754: error: passing 'const AHa
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable_policy.h: In member function 'size_t st
ed hash, false>::M_bucket_index(const std::tr1::__detail::Hash_node<Value, false>*, size_t) const [with Key = A, Value =
l = AHasher, H2 = std::tr1::__detail::Mod_range_hashing]':
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable:1242: instantiated from 'void std::tr
__constant_iterators, __unique_keys>::M_rehash(typename Allocator::size_type) [with Key = A, Value = std::pair<const A, i
>, Equal = std::equal_to<A>, H1 = AHasher, H2 = std::tr1::__detail::Mod_range_hashing, Hash = std::tr1::__detail::Def
__constant_iterators = false, bool __unique_keys = true]':
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable:950: instantiated from 'std::tr1::__d
Allocator, ExtractKey, Equal, H1, H2, Hash, RehashPolicy, __cache_hash_code, __constant_iterators, __unique_keys>::M_i
or, ExtractKey, Equal, H1, H2, Hash, RehashPolicy, __cache_hash_code, __constant_iterators, __unique_keys>::Hash_code
ExtractKey = std::Select1st<std::pair<const A, int>, Equal = std::equal_to<A>, H1 = AHasher, H2 = std::tr1::__detail::
rehash_policy, bool __cache_hash_code = false, bool __constant_iterators = false, bool __unique_keys = true]':
/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../include/c++/4.4.6/tr1_impl/hashtable:985: instantiated from 'std::pair<typ
e hash code, __constant_iterators, __unique_keys>::iterator, bool> std::tr1::Hashtable<Key, Value, Allocator, ExtractKey
st Value&, std::tr1::true_type) [with Key = A, Value = std::pair<const A, int>, Allocator = std::allocator<std::pair<cons
_H2 = std::tr1::__detail::Mod_range_hashing, Hash = std::tr1::__detail::Default_ranged_hash, RehashPolicy = std::tr1::__d
ys = true]':

```

I remember working very late nights and spending a lot of time trying to decipher compiler error messages. I talked to the TA, who was a CS grad student. I would come in with code that was seriously broken, and he would say things like “this loop is inefficient.”

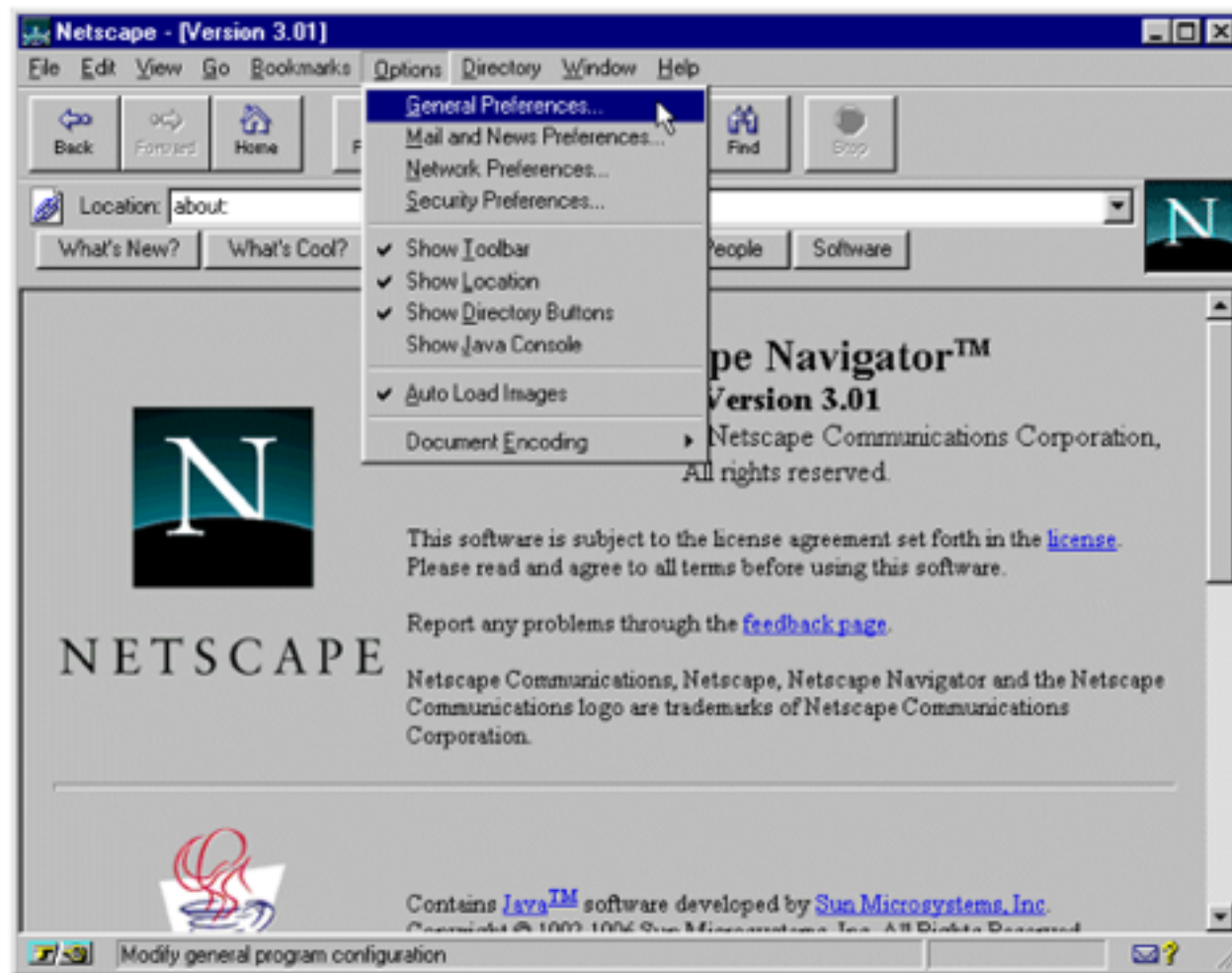
I was so frustrated by this experience that I ended up bailing



and becoming a German



major



But I still wanted to make stuff, and there was this new thing called the Web. I taught myself a bit of JavaScript.

In my first job (as a tech writer) I got a chance to make web-based training. I kept going as a web developer, and I kept teaching myself.



I learned from coworkers' code. I picked up some Java. I learned about encapsulation and design patterns. I spent more time with the Java Date object than with search algorithms, but I could reliably solve problems and write clean, modular, readable code. Eventually I had a business card like this.

But by then I wanted to make my own stuff. In particular I got interested in making games.



So I paused and got an MFA (and taught myself Unity). Then I worked at various small companies. And then

USC Games

miraculously I got a teaching job at one of the top programs. NOW I COULD



RIGHT THE WRONGS111!!!

USC Viterbi
School of Engineering
Department of Computer Science
GamePipe Laboratory

+

USC School
of Cinematic Arts
Interactive Media & Games Division

But what was the job actually?

USC Games is actually four degrees (and a few minors) in two different schools. Our engineering school offers degrees in computer science with a concentration in games.

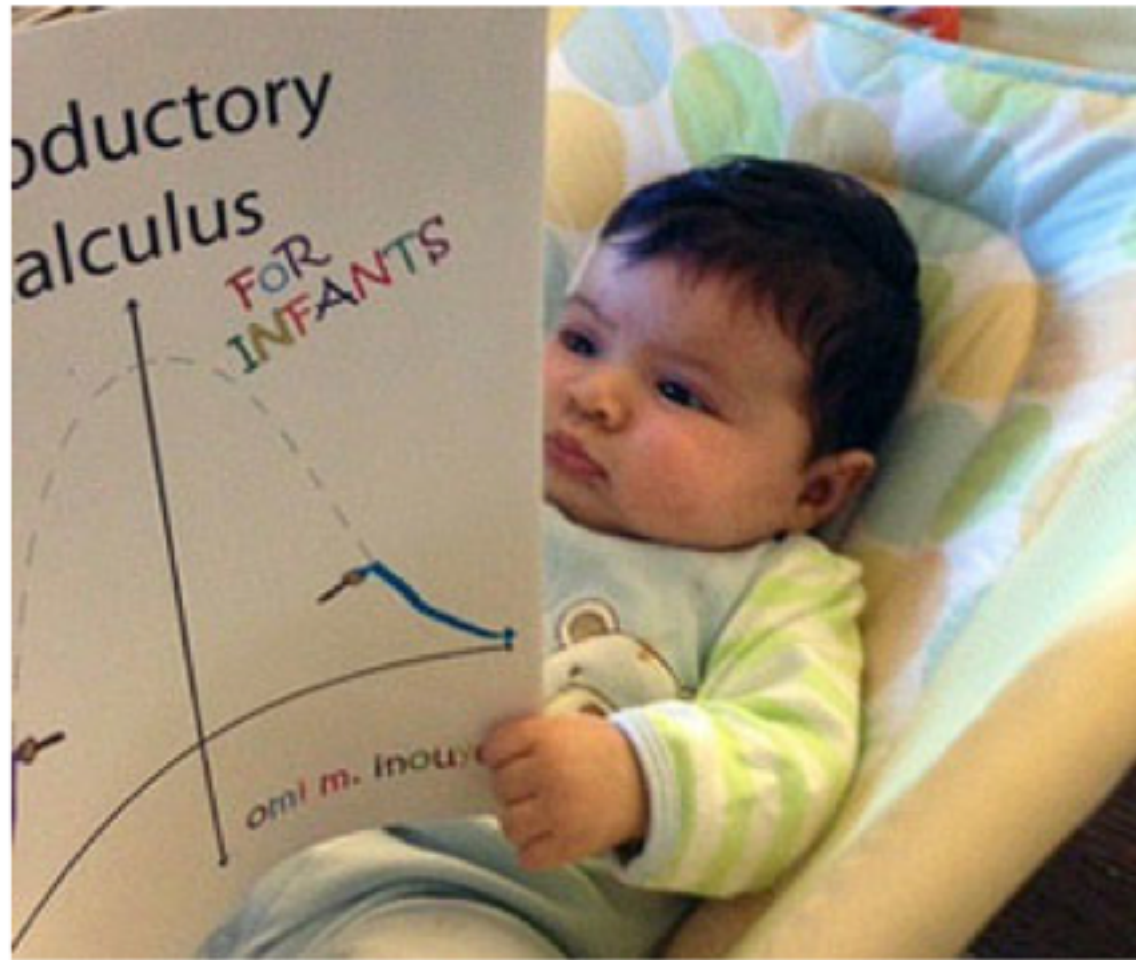
Interactive Media & Games lives in the film school and focuses on design and production skills. That's where I work.



So our philosophy is that the designers in our program should in fact learn to code. And that's the primary teaching challenge I came into: teach Unity, including C# scripting, to design students. Just enough so they can prototype their ideas.

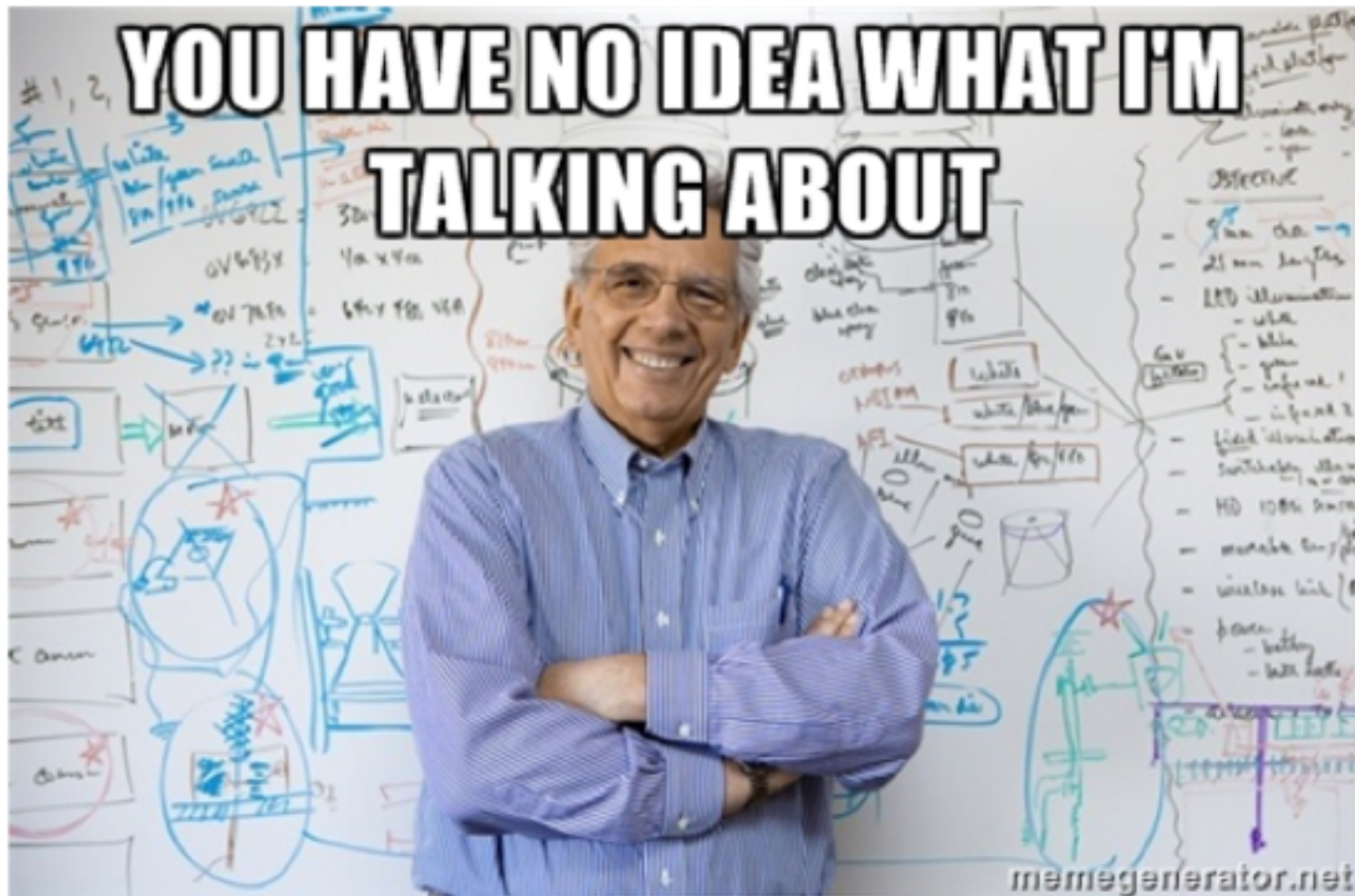


So for them, coding is a means to an end. They don't need the most powerful toolset or the cleanest code. Those things will come with time, if they want them. But as Mitu and others have noted here at the Summit, putting all that stuff in my intro class would just frustrate them.



They come in with a huge range of experience with code, but it's an intro course and I start at the beginning.

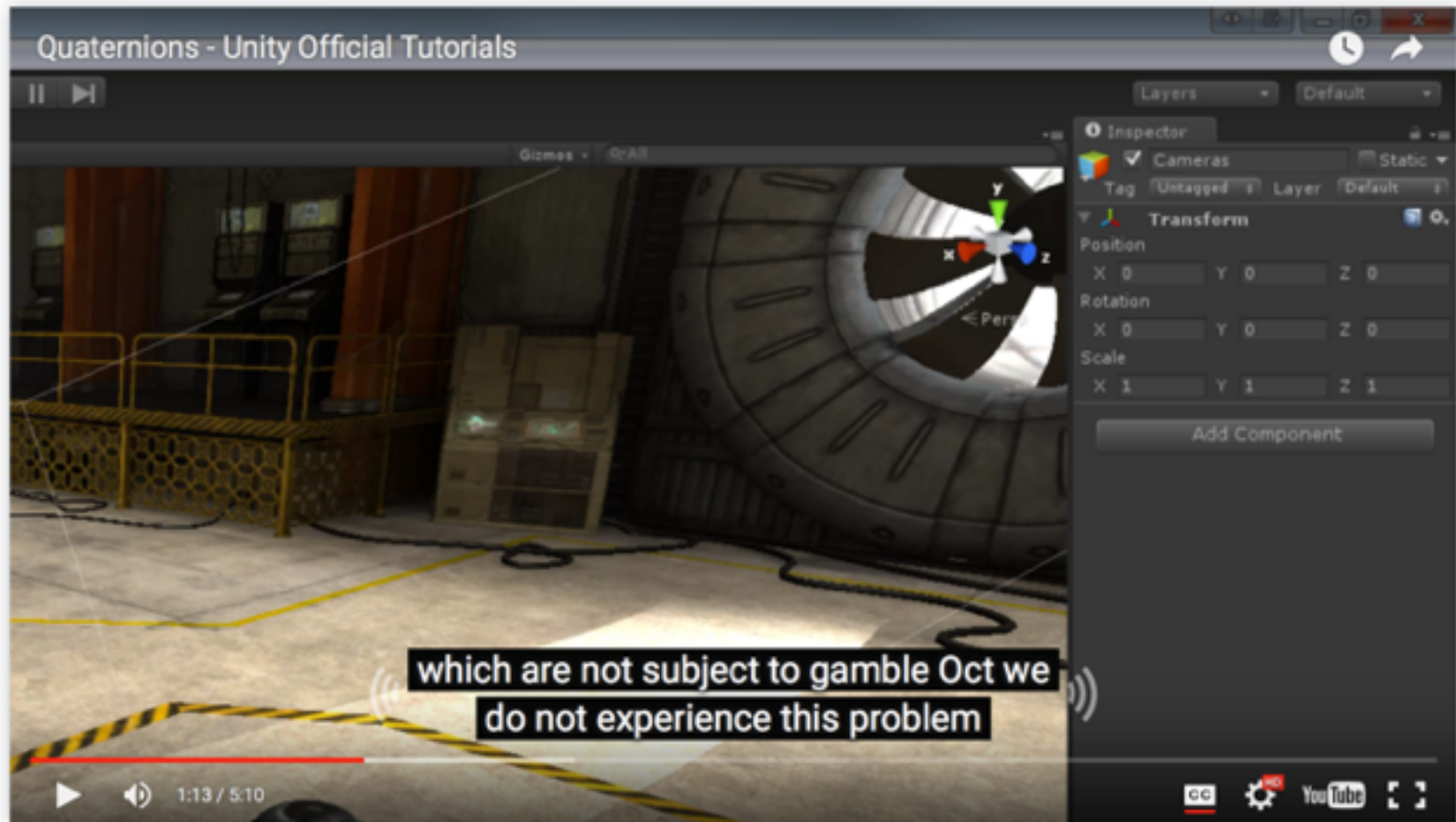
So. I had a solid syllabus from Jeremy Gibson but I didn't have lesson plans. I looked around; there weren't really any direct equivalents of this course. I looked over to Computer Science...



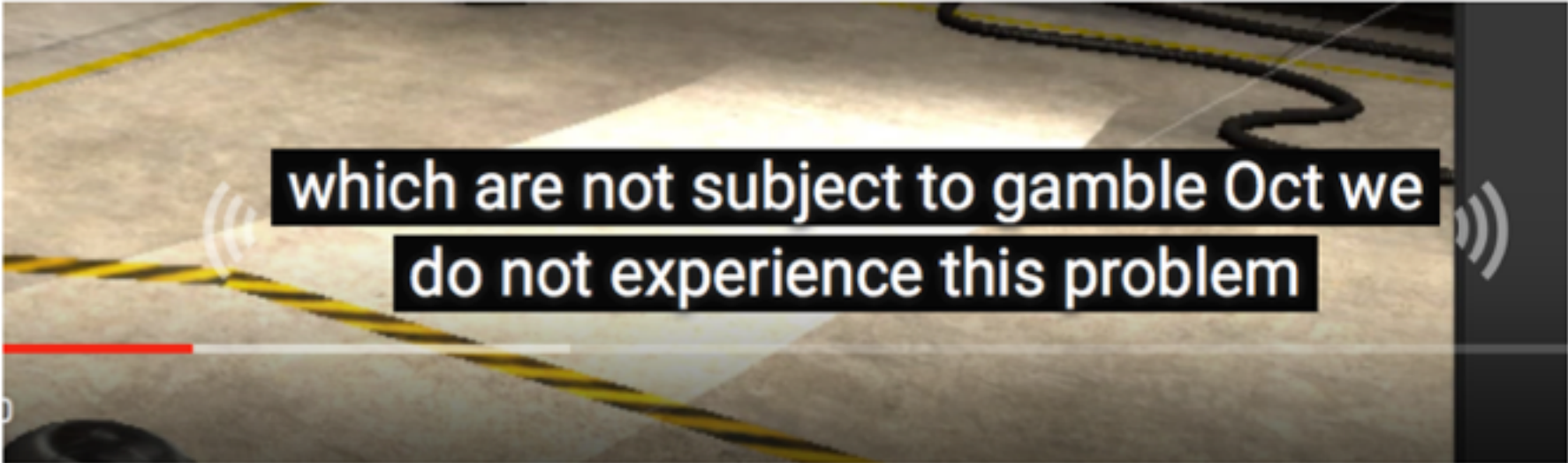
I'm not saying nothing changed, but it wasn't as different as I would have thought.

But they were solving a very different problem anyway. In a CS class it's reasonable to assume some amount of intrinsic motivation to work with code. My students are smart, curious, hardworking people, but they're not there specifically to learn how to code. It's a means to an end. My approach has to reflect that.

So back to the lesson plans. How about resources from Unity?

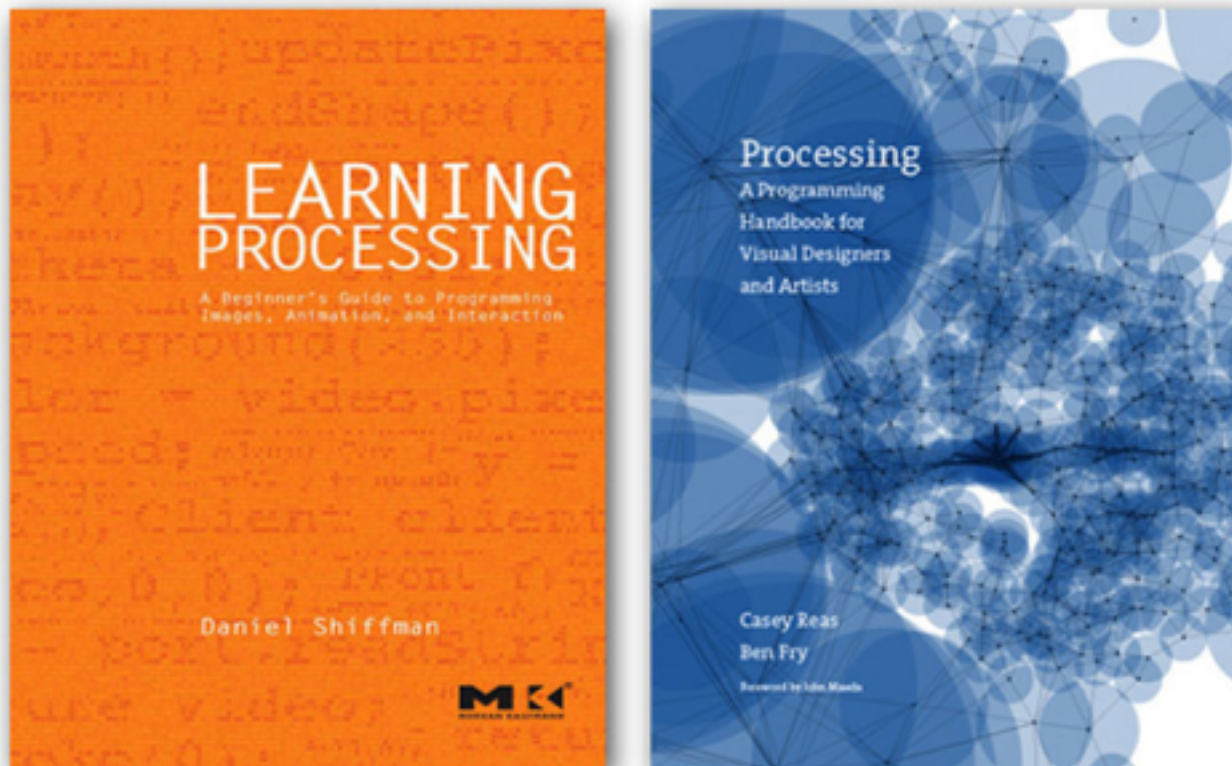


They had some tutorials. After a while they even had videos. Problem solved...



which are not subject to gimbal lock we
do not experience this problem

This is just bad subtitling for gimbal lock, but they are pretty rife with jargon – they seem to have their engine coders do the tutorials. They're not useless, but they're not what I needed either.



So, okay, there's Processing, which is a good model for what I'm after. I teach a Processing class, which we've recently changed to half Processing and half Unity. I can probably adapt some of that material.

But in order to bring all of these sources together, I needed to look at the big question:



GDC EDUCATION
SUMMIT

What makes learning to program hard?



GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 18-19, 2016 #GDC16

In particular, what do my design students find difficult, versus your average CS major? And then naturally



GDC EDUCATION
SUMMIT

How can I make it better?



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

I refuse to just teach them less. I really want them to be able to express themselves and feel empowered to learn more when they need it.

And I guess I'm attracted to hard problems.



GDC EDUCATION
SUMMIT

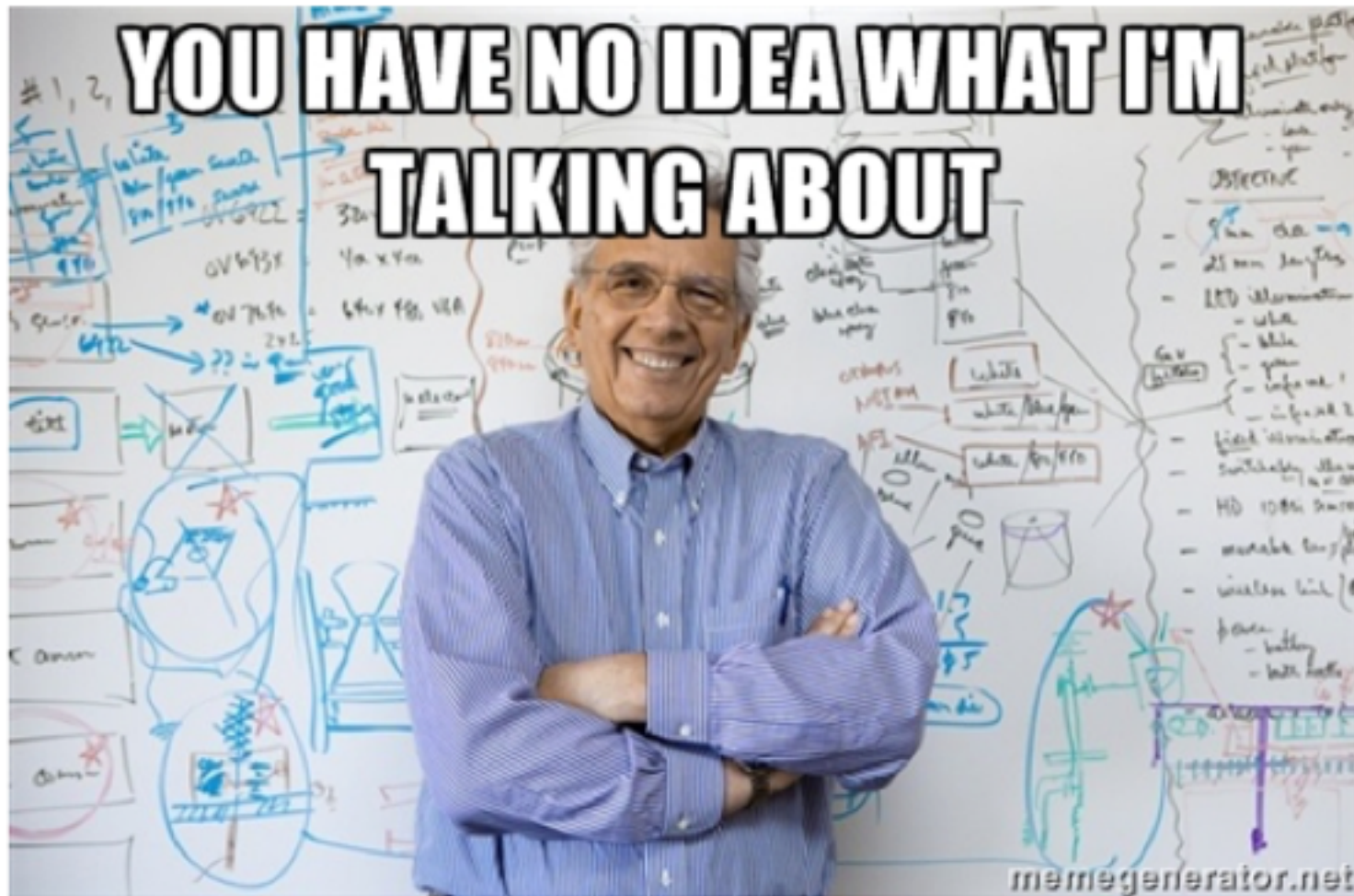
Principles



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

I'd like to preface this section saying I don't have any formal education in the pedagogy computer science. If you do, I would love to talk to you! You'll see my Twitter handle at the end.

So, when I got this job, I didn't have much teaching experience, and I started trying to educate myself. The first thing I needed was a framework for understanding the pedagogical problem. What are we teaching when we teach people how to program?



And what are we **not** teaching?

For me it felt like there was something missing in that C++ course, but I couldn't codify it.



Learning to Program: A Breakdown

1. General orientation
2. The notional machine
3. Notation
4. Plans and schemas
5. Pragmatics

Source: Benedict du Boulay,
Some Difficulties of Learning to Program

So here's a breakdown I liked. It's many years old now, but it's still cited. I tried to map this list to my experiences as a student and as a teacher, and it seemed to me that



We usually focus on these two:

1. General orientation
2. The notional machine
3. **Notation**
4. **Plans and schemas**
5. Pragmatics

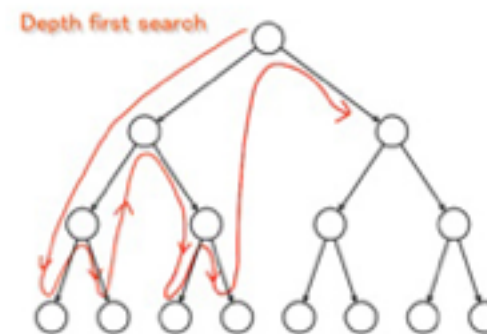
The classes I took had focused almost exclusively on notation and schemas.


```

1  using UnityEngine;
2  using System.Collections;
3
4  public class Master : MonoBehaviour
5  {
6      public enum Commands    // List of Commands
7      {
8          FORWARD,
9          BACK;
10     };
11
12     public Commands command;
13     public Slave reportBack;
14
15     void Start ()
16     {
17     }
18
19     // Update is called once per frame
20     void Update ()
21     {
22         SlavesOrder(reportBack);
23     }
24
25     private void SlavesOrder(Slave order)
26     {
27         {
28             if(reportBack.order.command == Commands.BACK)
29             {
30                 Debug.Log("Master I am moving FORWARD");
31             }
32
33             if(reportBack.order.command == Commands.BACK)
34             {
35                 Debug.Log("Master I am moving BACKWARDS");
36             }
37         }
38     }
39 }
40

```

Notation



Schemas

Notation is just syntax – where do you need braces, semicolons, etc. Schemas are various kinds of reusable patterns that come up in solving problems: search algorithms and collision listeners and factories.



But the other ones
are important!

1. **General orientation**
2. **The notional machine**
3. Notation
4. Plans and schemas
5. **Pragmatics**

These seemed to describe what was missing from my education.

These might seem fluffy, or like a waste of precious class time. But have you ever encountered a smart, capable student who nonetheless struggled with CS? It could be that they were missing some of this.

Let's go through these briefly.



General Orientation

- What is code?
- What are you doing when you write it?
- Why would you want to work with it?

So, starting at the beginning. These are some questions that we might address very briefly in the first class.



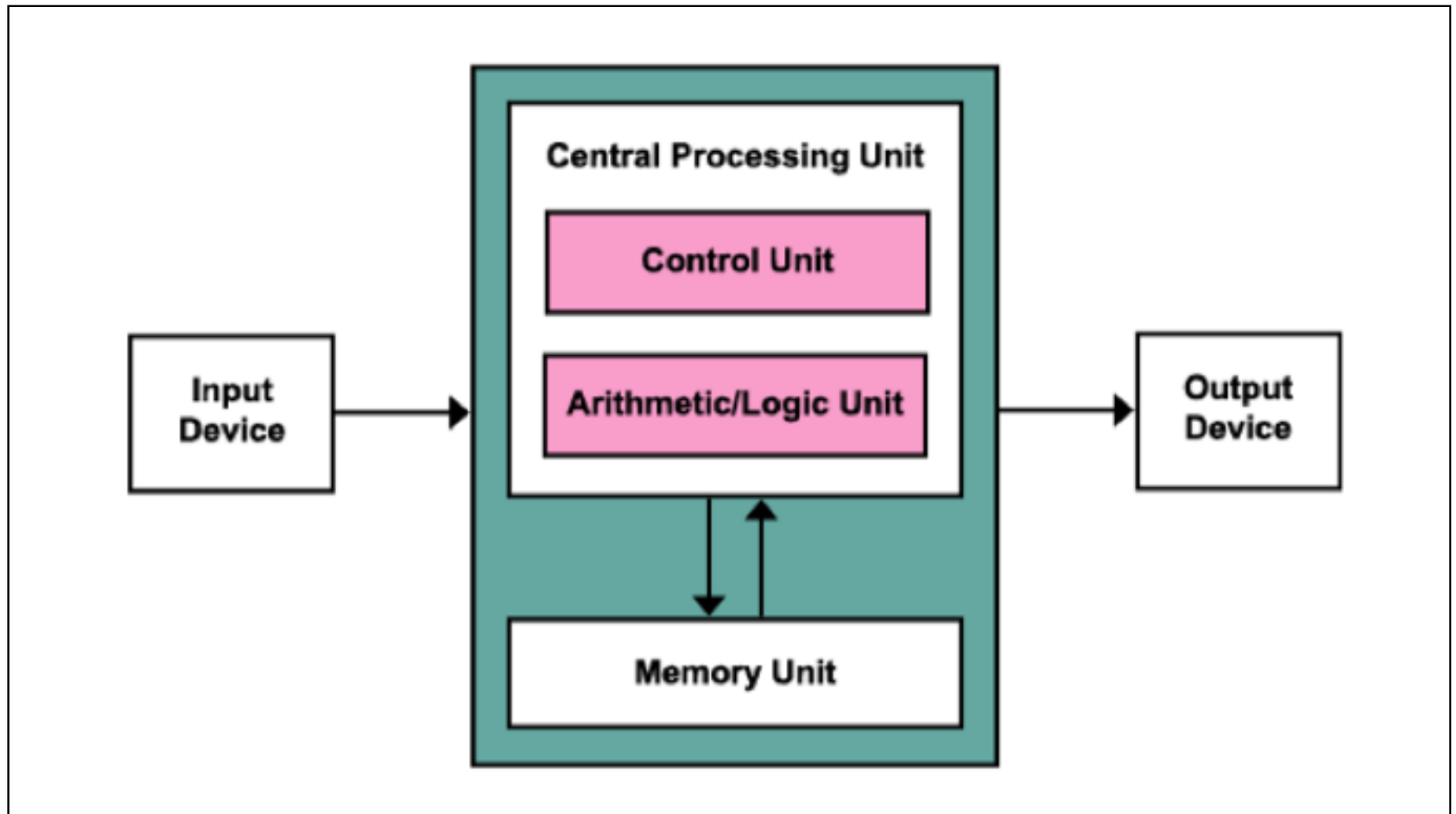
What is code? What is it like to write it? I try to convey that the act of coding is difficult because you are basically constructing a universe from scratch. We are not gods, so we make mistakes. Sometimes I even make them read a bit of semiotics.

But orientation is also about WHO programs, and why.



Remember, my students don't think of themselves as programmers. So I spend quite a while explaining my background to them. I emphasize that I'm a German major, that I'm self-taught. I like using code because it lets me express my ideas. But there are many reasons that people come to code, and many reasonable stopping points depending on what you want.

I talk about soft skills and attitudes: fearlessness, patience, laziness (the good kind), plain old stubbornness. And most of all, asking for help.



The notional machine is your mental model of what the computer is doing – your ability to predict its behavior.

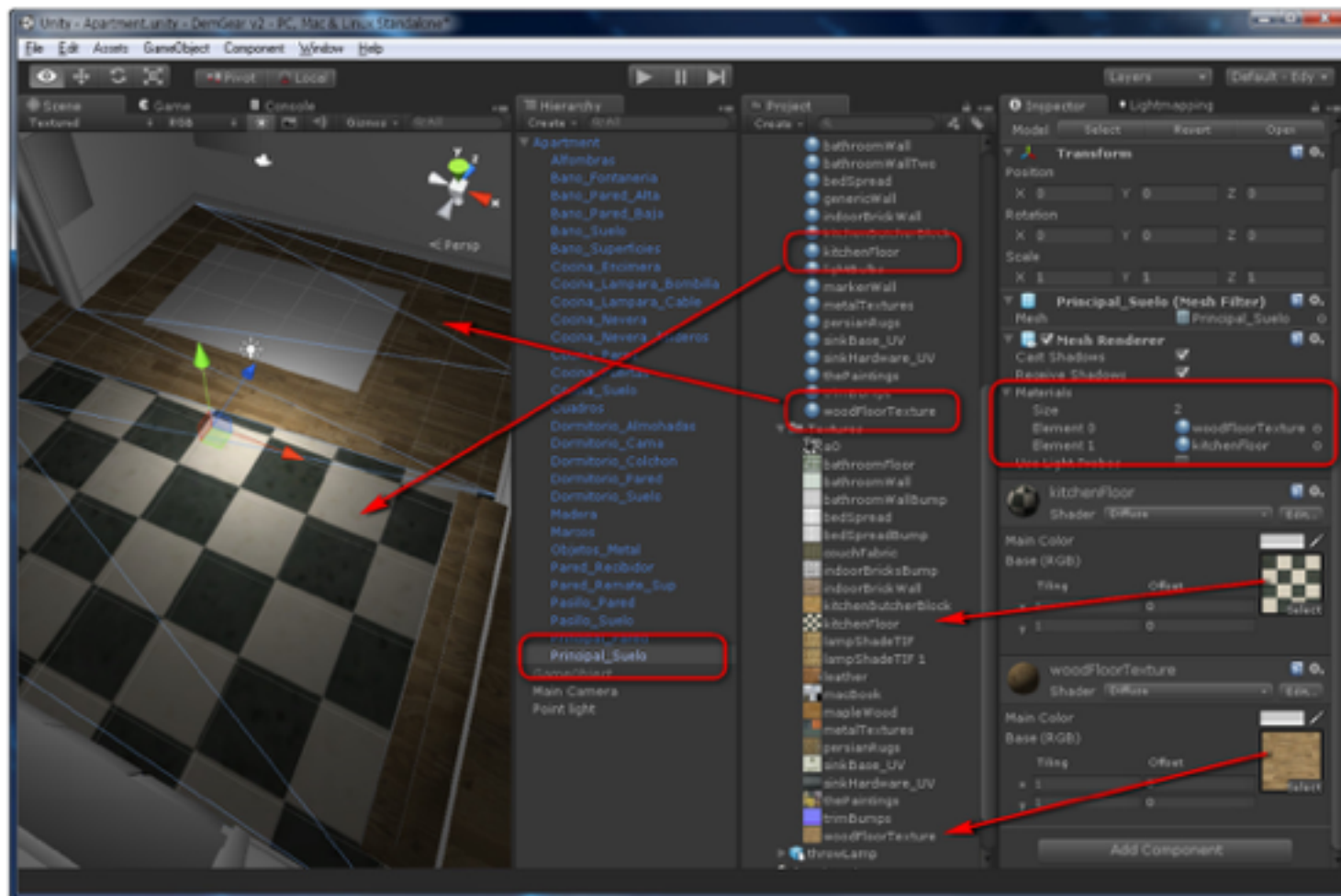
We often use analogies for this: variables are boxes and so on. Or we talk about von Neumann architecture, or perhaps discuss stacks and heaps and registers. These things are all useful, but

Computers forget things a lot

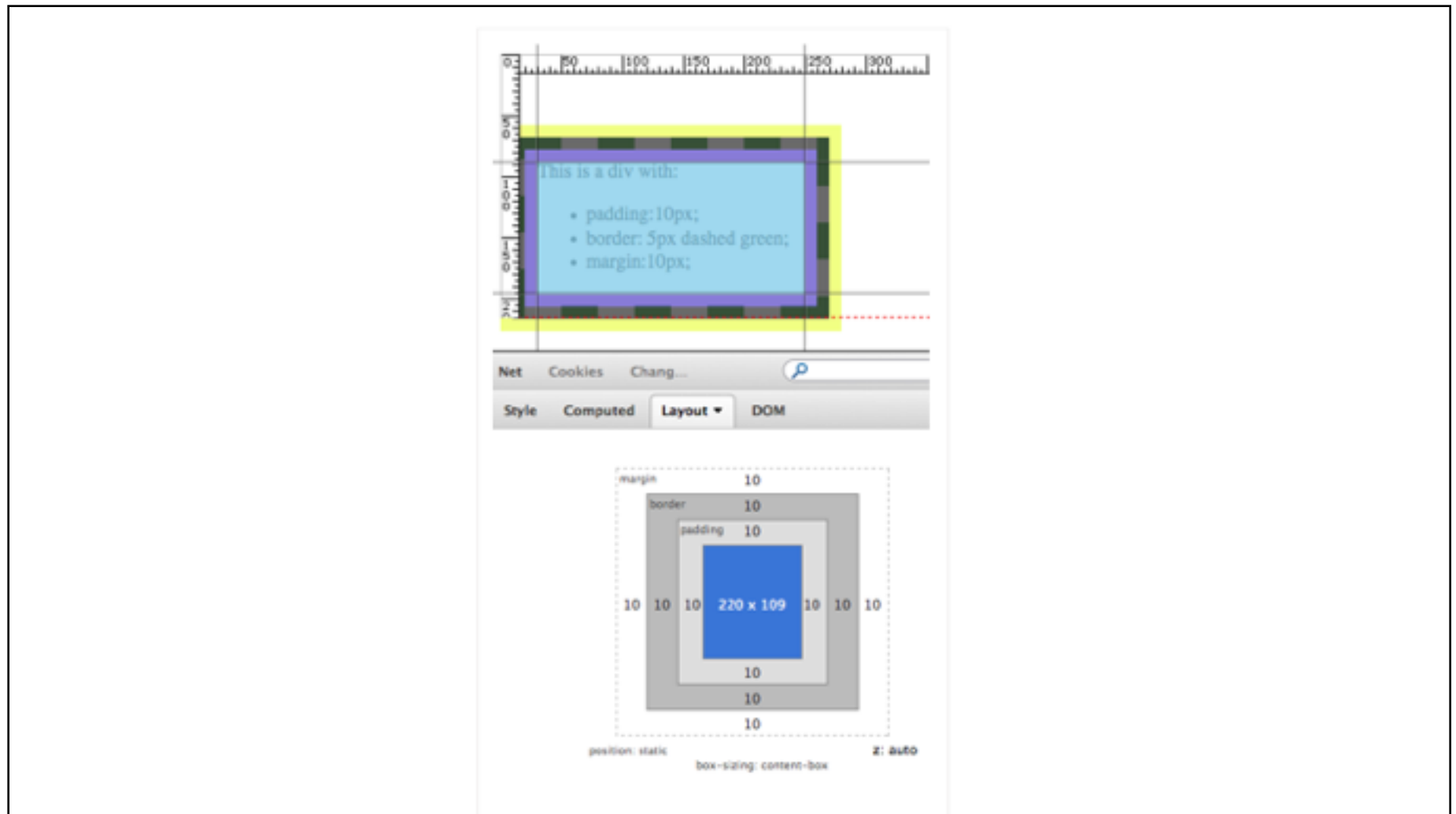


what students need to know is how to work with computers: things like **why** computers need variables. Yes, this is implicit in the architecture, but pedagogically it's like if I said "humans have two elbows, now write Crime and Punishment".

Now let's look at the last item, pragmatics.



Pragmatics is using the tools of programming: IDEs, engines, compilers. If we talk about our tools at all, we tend to say “this button does this, that button does that”.



And this is where I thought – aha! Here's a place I can bring in my professional experience.

As a UX designer my argument is that our tools should be better – they should help us to see and understand what we are doing. I am a disciple of Bret Victor in this way.

One example is dev tools, built into every modern browser. Here it's showing what's called the box model, right next to the object you're trying to position. Exposing the model like this is extraordinarily helpful for making web pages. But our tools generally don't do this.



GDC EDUCATION
SUMMIT

Pragmatics



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

So I'm focusing on PRAGMATICS, because it's where I feel I have the most to say.



Errors come from the **system** (user + tool).

The tool permits or perhaps encourages the error just as much as the human causes it.

Remember, I am working from the UX perspective. Our tools for programming, and often our design tools, make it far too easy to make mistakes.



I want to identify the usability problems that are specific to beginners. How do I determine what's hard for them?

I needed a language for what's going on mentally when someone is using Mono and Unity. So I went back to the library. Here's one way to do it:

- Step 1:* Recall that the first command is called "cut"
- Step 2:* Recall that the command "cut" is in the right click menu
- Step 3:* Locate the icon of the source file on the screen
- Step 4:* Accomplish the goal of selecting and executing the "cut" command
- Step 5:* Recall that the next command is called "paste"
- Step 6:* Recall that the command "paste" is in the right click menu
- Step 7:* Locate the icon of the destination folder on the screen
- Step 8:* Double click with left mouse button
- Step 9:* Locate empty spot on screen
- Step 10:* Move mouse to the empty spot
- Step 11:* Accomplish the goal of selecting and executing the "paste" command
- Step 12:* Return with goal accomplished

This is a breakdown of cutting and pasting a file in Windows. Did you know there are twelve steps? I didn't. Once you learn it you forget it.

So what are some small, atomic tasks in Unity? Let's say I ask you to add a collider to a game object in Unity. For me and you, it's like cut and paste.

Step 1: Locate the target GameObject.

Step 1a: Locate the Scene view in the Unity interface.

Step 1b: Navigate within the scene view using pan and zoom until the target GameObject is visible on screen.

Step 1c: Click to select the target GameObject.

Step 1d: Observe that the Inspector now shows information about the selected GameObject.

Step 2: Recall that colliders are components.

Step 3: Navigate to bottom of Inspector view.

Step 4: Click 'Add Component' button.

Step 5: Recall that Unity considers collider components part of the physics system.

Step 6: Choose 'Physics' submenu.

Step 7: Locate the desired shape from the 15 available options under Physics.

Step 8: Click on the desired collider type.

But – well, when I broke this down I was surprised. I was able to remember some of what it's like to do this the first time. But remember, this is only one small step of a larger process

Make a GameObject respond to collisions

Step 1: Add a collider to a game object in Unity with appropriate settings.

Step 2: Add a collider to another game object in Unity with appropriate settings.

Step 3: Add a Rigidbody component to one or both of the objects.

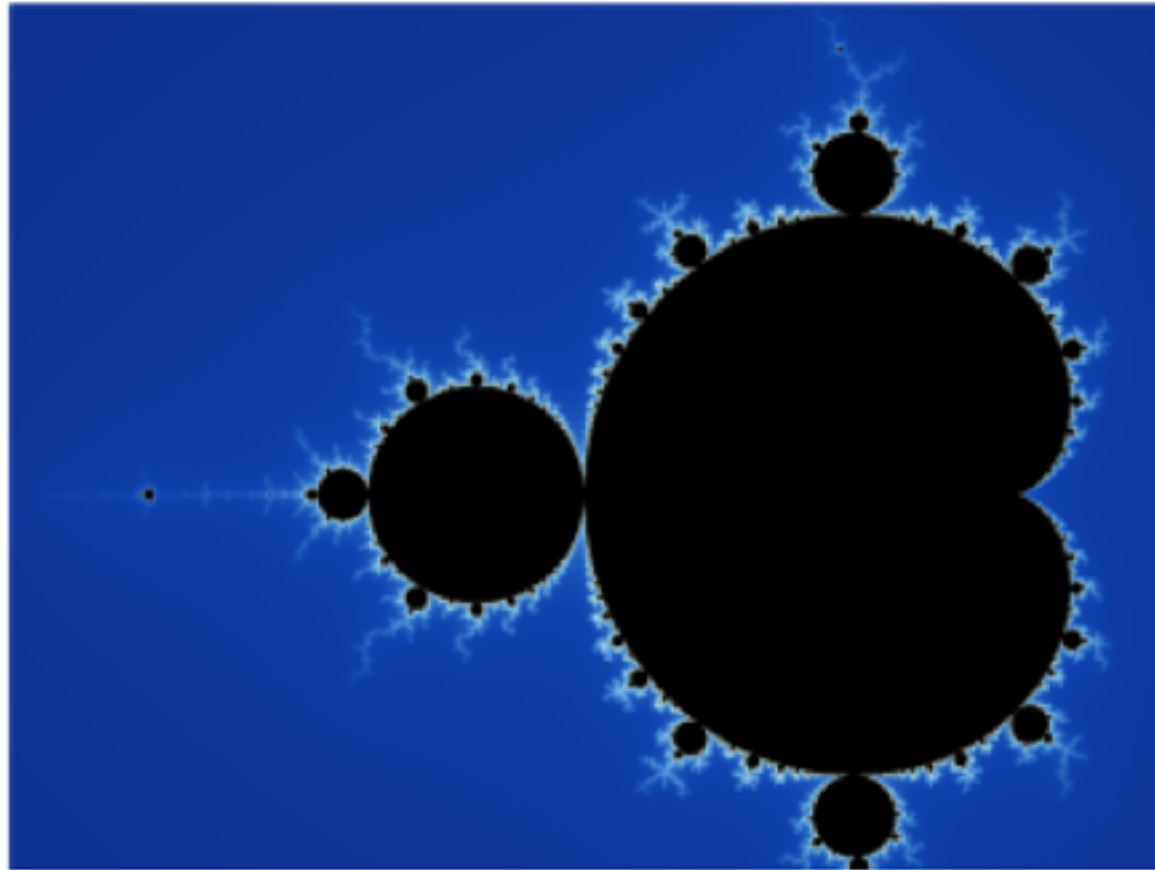
Step 4: Attach a script to one of the game objects.

Step 5: Open the script.

Step 6: Enter (or locate and paste) Unity's specific method name and signature for the collision event you're listening for.

Step 7: Add the code that will be executed when that collision event occurs.

where this is the larger schema. And then each of these has its own set of steps

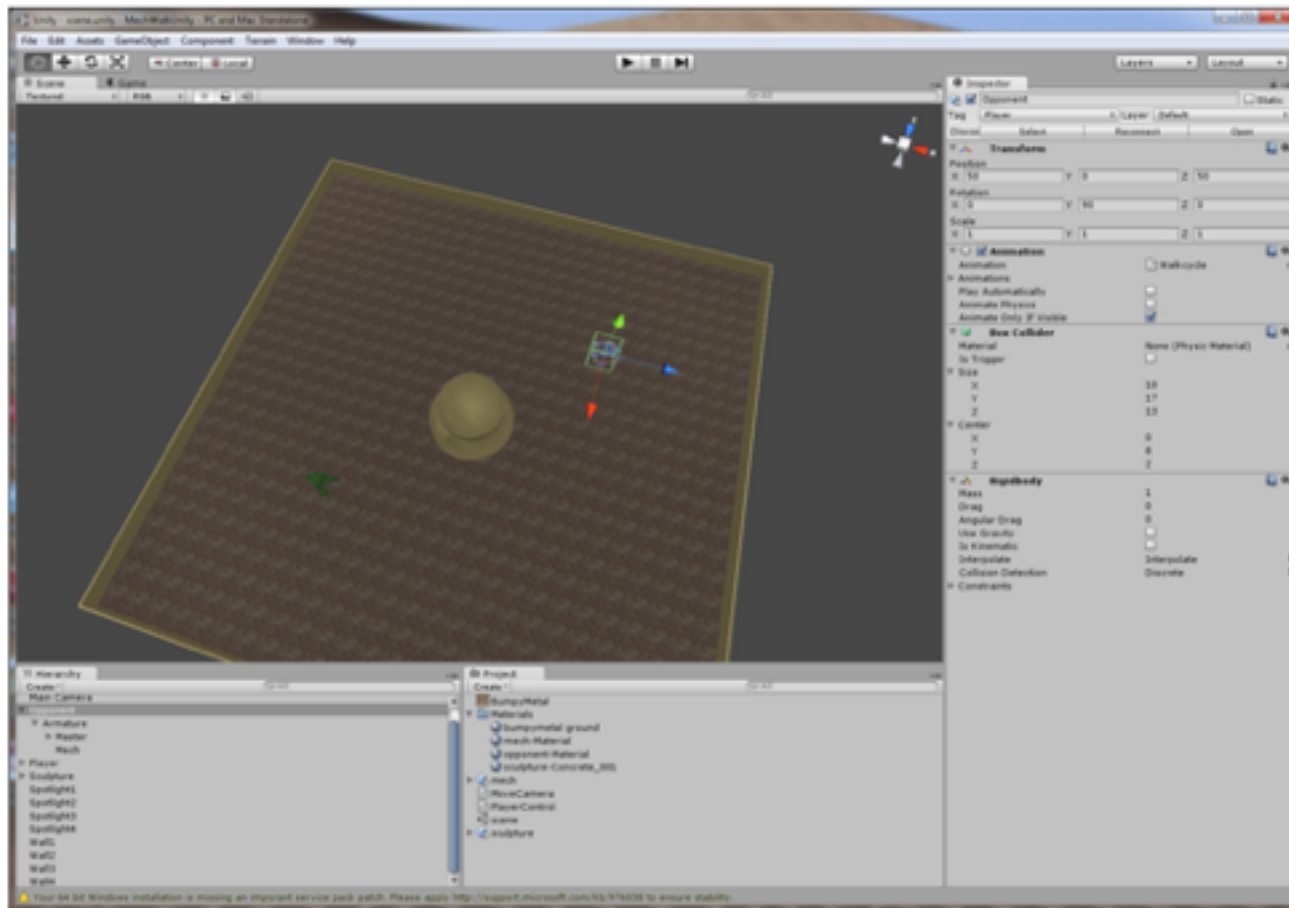


And they are themselves part of a larger task like “respond to player death”... it goes on, as deep as you like.

So how to help students situate themselves in this process?



It struck me that a lot of those steps started with words like “Recall”. I remember struggling with losing my mental place a lot when programming, although I didn’t with other subjects. Aha! A hook!



Okay, so particularly with 3D environments like Unity or Maya, you have a huge amount of information on the screen.

You have to sort out what's important, remember the location of buttons and so forth while also maintaining your mental stack of tasks and schemas and so on. It's a delicate dance of mental focus and it can break down in a number of ways. Here's a quick list of five from Claudia Roda.



Habitation errors are auto-pilot mistakes.

Task resumption means re-establishing mental context when you return to an interrupted task.

We usually mean habitation errors when we say we weren't paying attention.

Task resumption is hard for everyone – you've read (and perhaps written) programmer rants about being interrupted. It can only be harder for beginners.

So these two are less important.



Prospective memory allows you to remember that you need to do something.

Retrospective memory allows you to remember facts and concepts necessary to your task.

These are frequent sources of problems for beginners. For one, they haven't yet built the short-term memory skills. Perhaps more importantly they haven't yet built up the higher-level chunks that experts use to recognize problems and maintain a sense of what they are doing.



Primary task disruption is what we usually call distraction or being sidetracked.

This includes things like alert dialogs, but for beginners the main source of this is bugs and particularly compiler errors. You have to shove back your whole stack and fix the bug before you can continue. Perhaps understandably, beginners often try fixing problems with very small changes, basically hoping to make the disruption go away. They already feel sidetracked and they feel like they need to keep getting results.

For this reason, debugging is considered the hardest skill to acquire for beginners. Lastly we have



GDC EDUCATION
SUMMIT

Missing important information comes from
incorrectly prioritizing information that is available
to you.



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

This is particularly tough for beginners. There's all that stuff on the screen, in both Unity and Mono. So many buttons and menus! You only learn to prioritize what you're seeing when you've already built up a sense of the tool and a strong mental model of what you're doing.

So, how can our tools help with this? I'll mostly focus on Unity here, with just a brief look at mobile and web tools.



GDC EDUCATION
SUMMIT

Tools: Unity



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16



Not for Beginners

- Haste Pro, where you need to know what you're looking for
- PlayMaker, where you are avoiding scripting
- ProBuilder, which certainly makes Unity easier to use but does not make its guts more legible

So these are anti-patterns. A lot of people have tried to improve the Unity editor. These are great tools, but they do not particularly help beginners.



Unity Tools

- Inspector Navigator
- Editor Console Pro
- Script Inspector 3
- Full Inspector

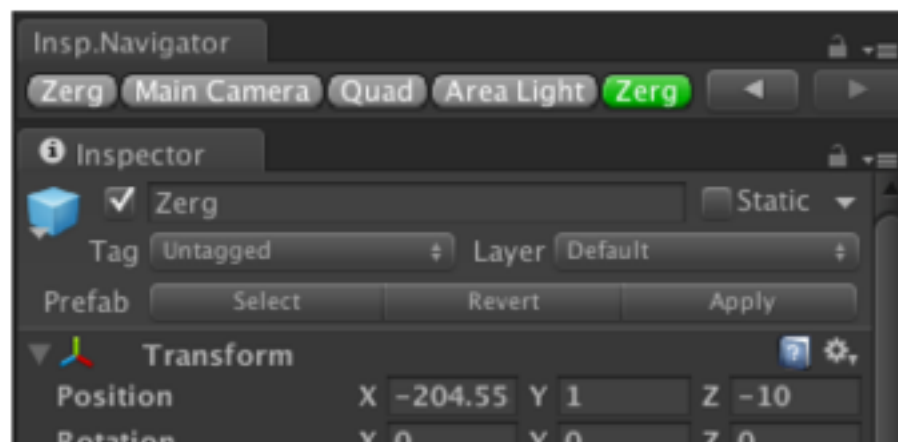
These are the four that I've found that most directly address the problems beginners have.

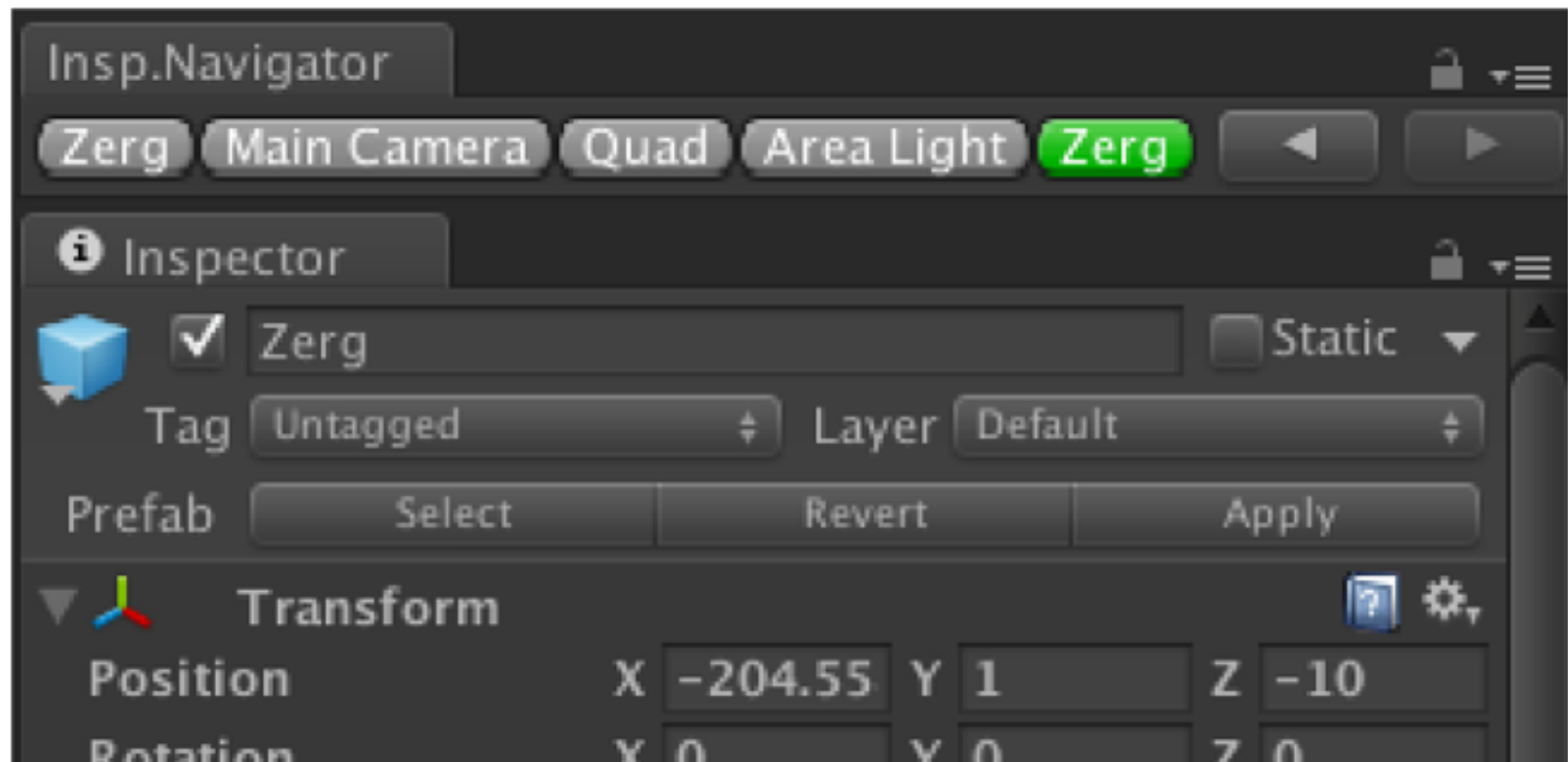


Inspector Navigator

Adds back/forward and breadcrumbs to the Inspector pane.

Optionally focuses scene camera on selected item.







Inspector Navigator

Maintains mental context very effectively, with a familiar UI metaphor.

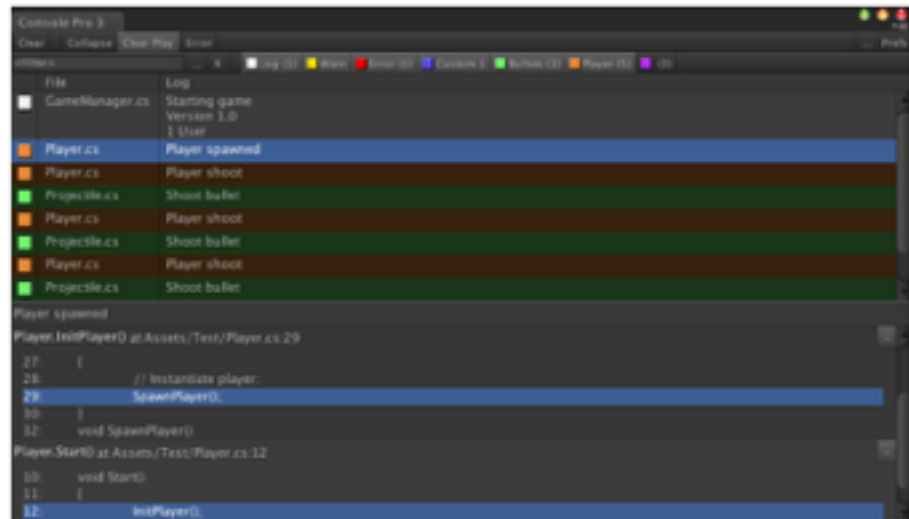
Helps with:

- prospective memory
- primary task disruption

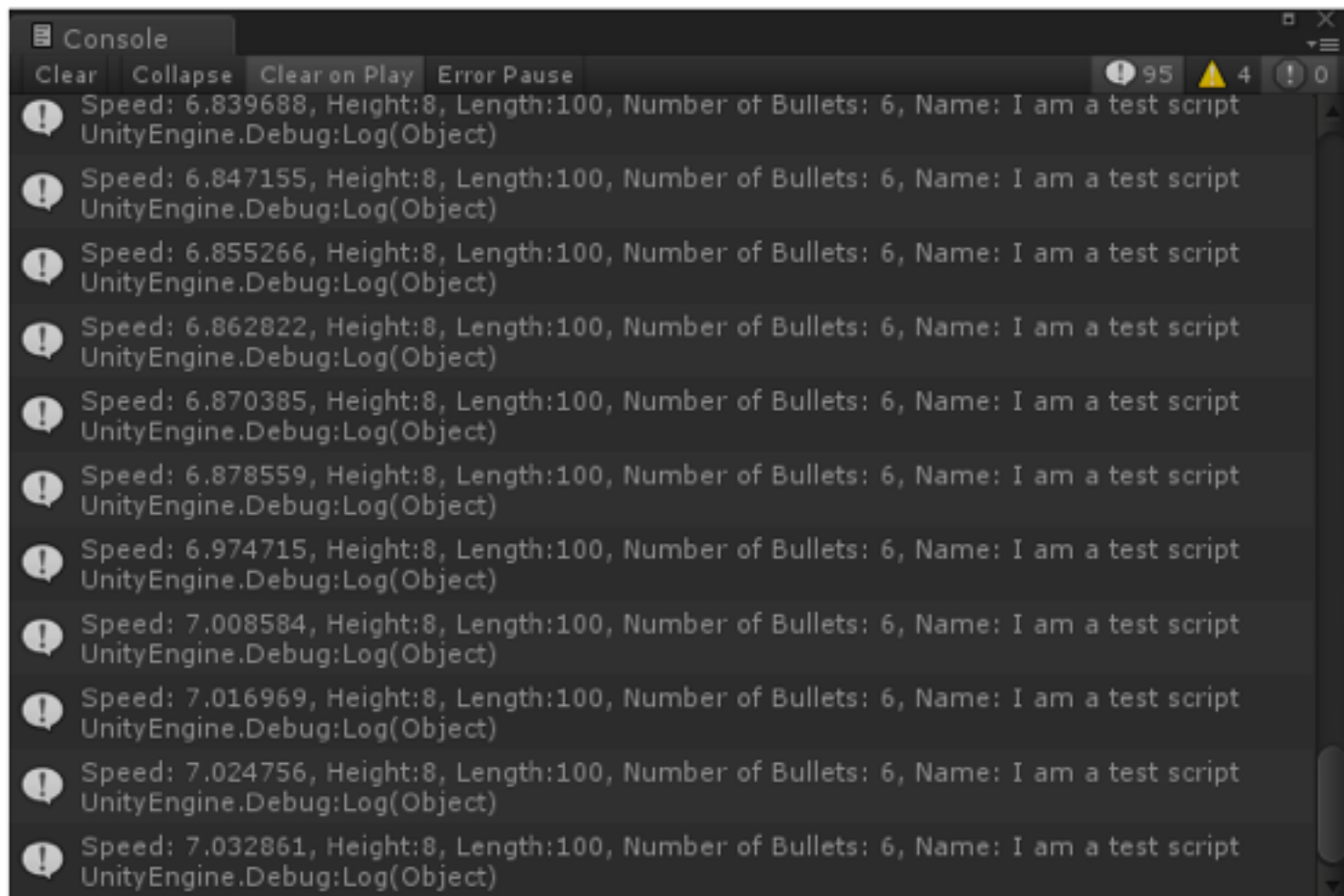


Editor Console Pro

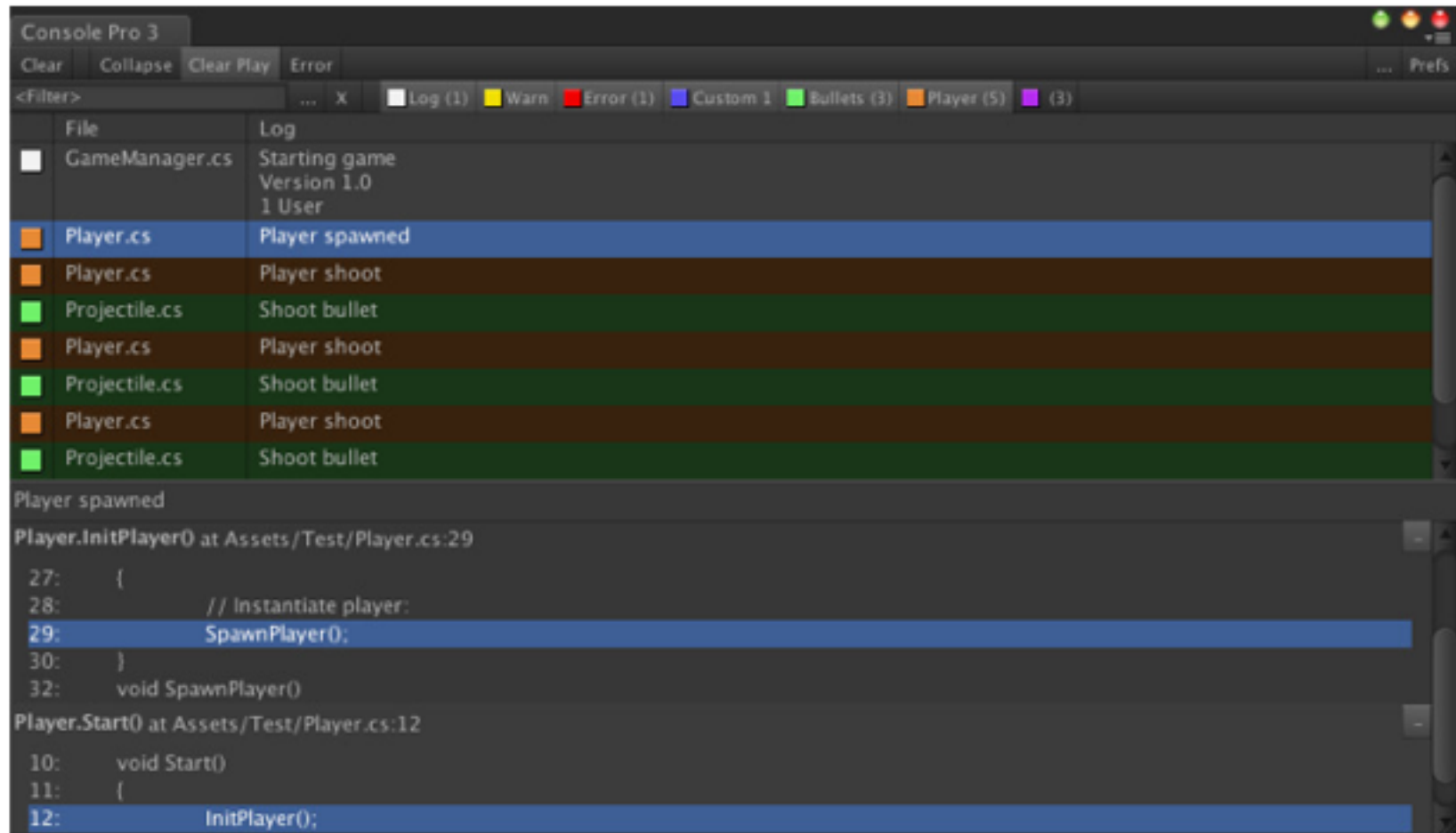
Improved console window offering visual prioritization, search and filter functions, error context and more



As mentioned before, debugging happens to be the hardest cognitive task for beginners. The Unity console does not help.



This is the usual console display. All the information has the same visual weight. You find whatever you're looking for by scrolling through. You can't sort it, search it or export it. It's absolutely terrible!



Now we have visual guidance with the use of color, and particularly letting you customize what you are looking at through search and filter.



Editor Console Pro

Allows user to customize display to show what they want to focus on. Guides attention with color.

Helps with:

- primary task disruption
- prospective and retrospective memory

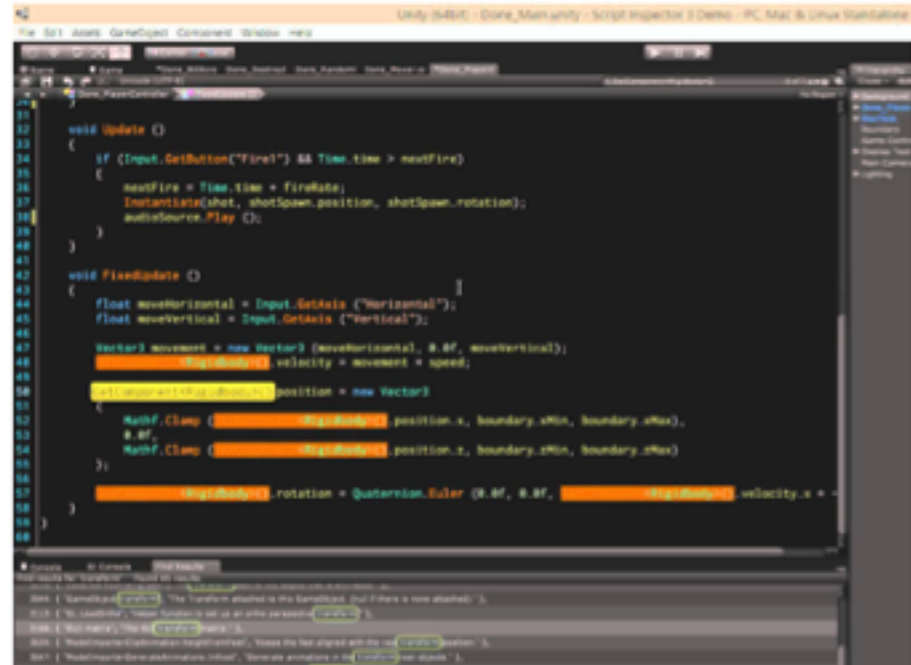
These features make debugging less painful, less distracting and more effective.



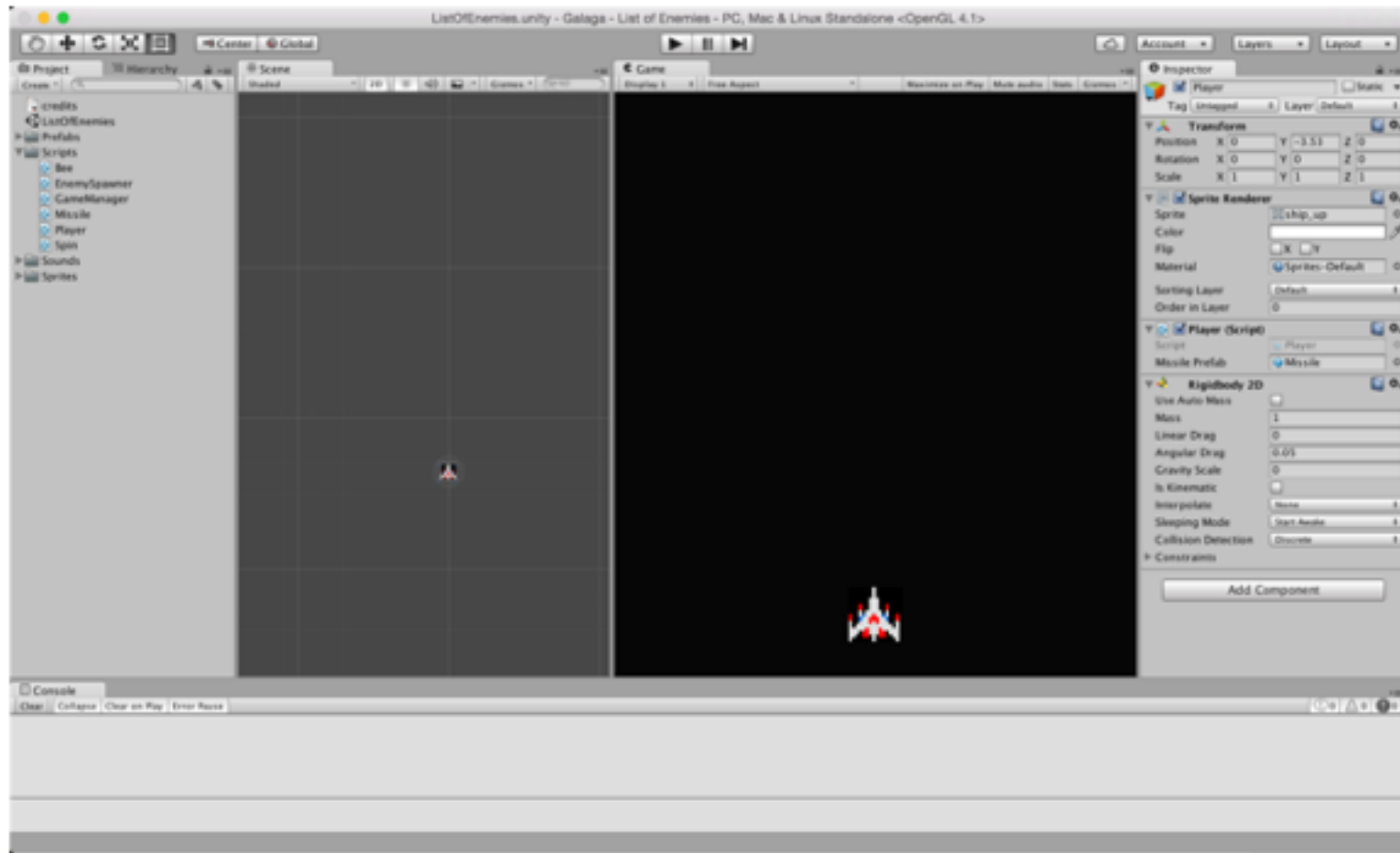
Script Inspector 3

In-editor IDE

Mature feature set
(supports code
regions, has auto-
complete, highlights
syntax errors, more)

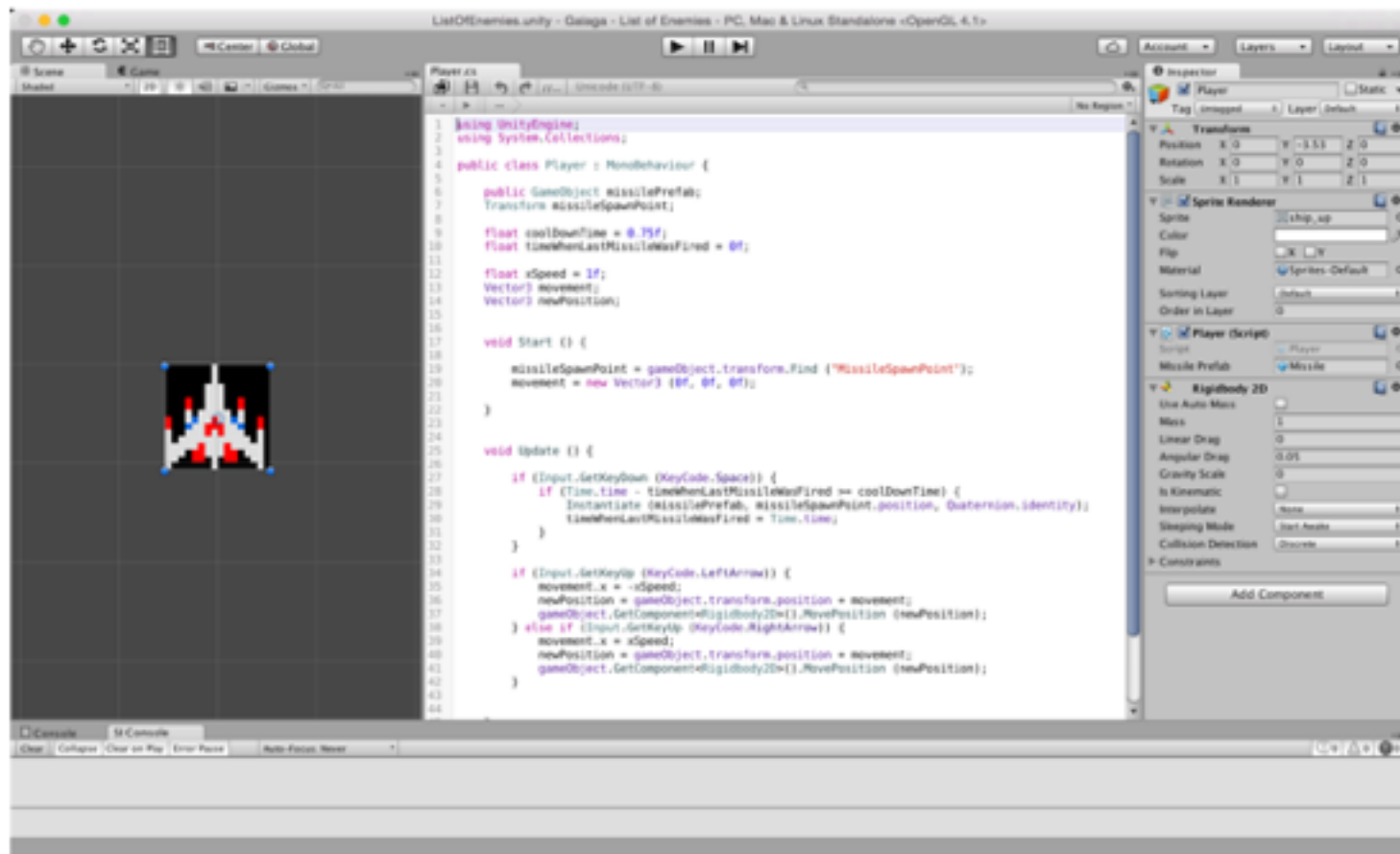


Script Inspector 3 is another great tool. Let's take a quick look at the problem it's solving.



So this is the user experience for writing code in the Unity world. [Flip back and forth rapidly between this and the next slide.] You are constantly flipping back and forth between Mono (or VS) and the Unity editor. Try to keep an eye on the properties of the rigidbody while I do this. Can you?

It's hard to keep your focus in one place when you're constantly retraining your eyes. You get better at it with practice, but when you're first using Unity this is really disruptive.



What if you could edit your code in the same visual context as your game objects?



Script Inspector 3

Significantly improves **directness** of connection between action and result. Does not disrupt mental context.

Helps with: primary task disruption, prospective memory, missing important information

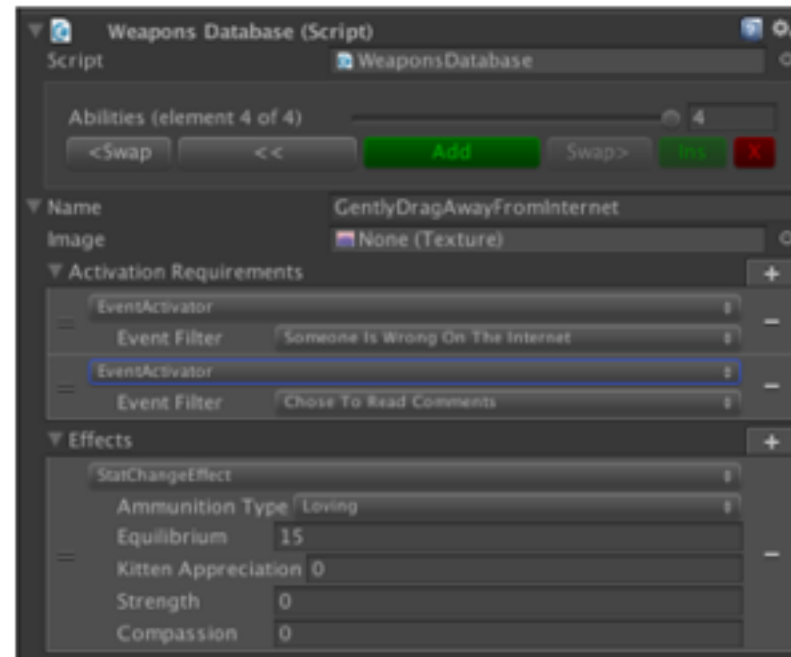
If you can edit your scripts in the same visual environment as your game object, you can maintain your sense of place and task. You get a much shorter feedback cycle as well, so you can learn the connections between your coding choices and their effects in the game.

This approach also rewards making small changes and testing them immediately, which experienced programmers do but beginners pretty consistently do not.

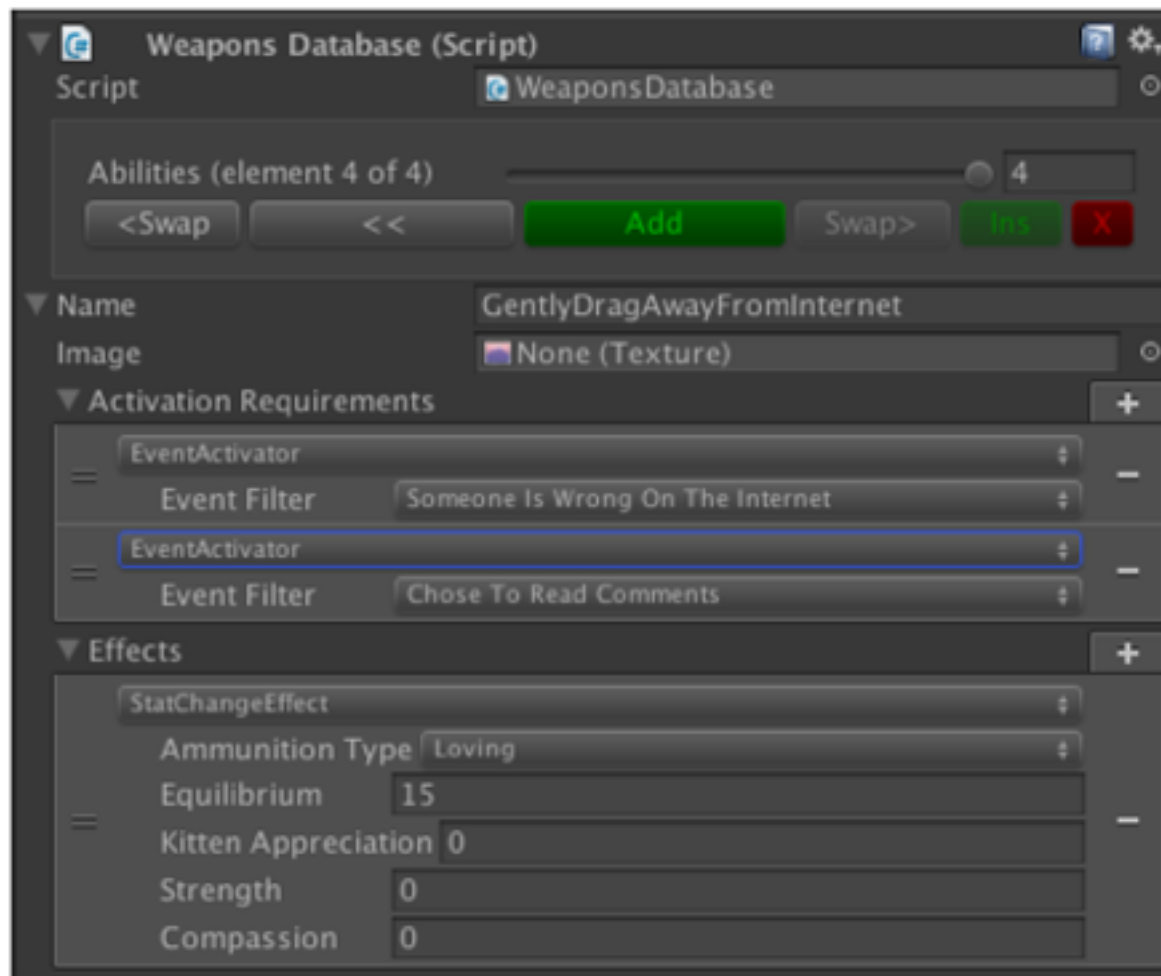


Full Inspector

Exposes advanced data structures.
Lets user edit the properties of referenced scripts in place in the Inspector



Full Inspector is more for intermediate students. It is tremendously helpful for explaining code structures like dictionaries and delegates.



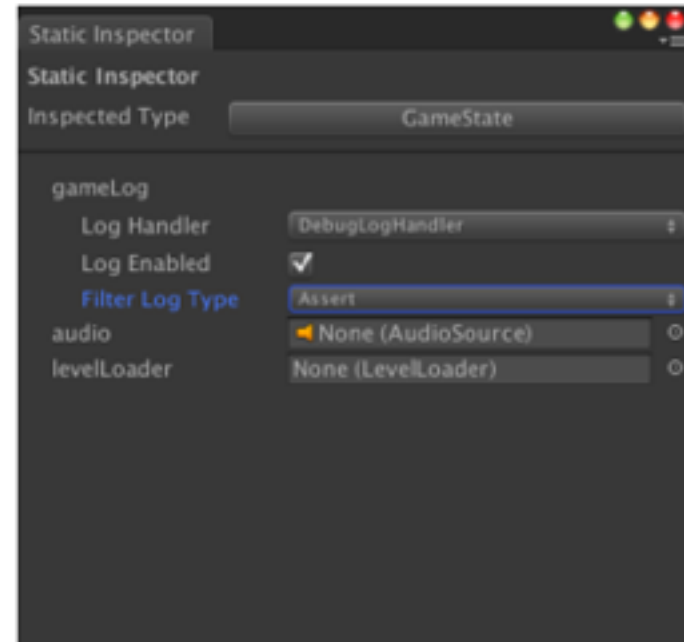
It allows you to see the cause and effect of script interactions by letting you define them in a visual editor.

Here, for example, you can specify events that will trigger effects. Letting the student see that multiple events can trigger the same effect and vice versa gives them a good, concrete hook for understanding the idea of loose coupling.

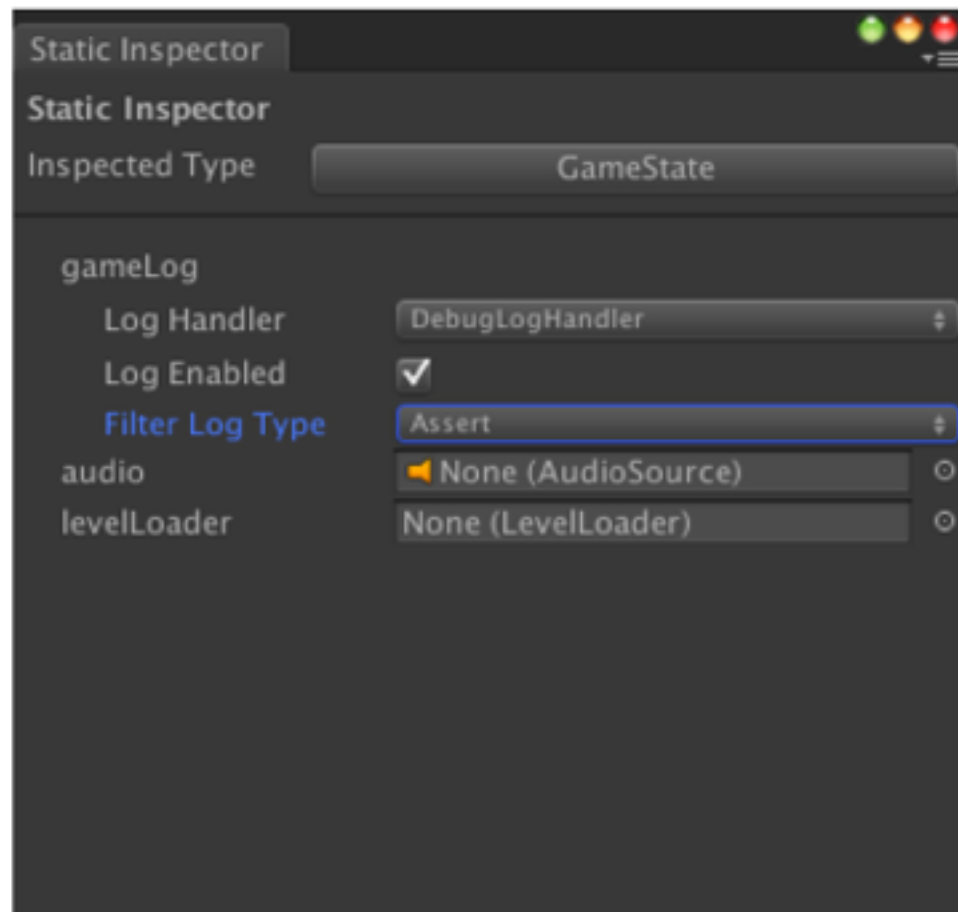


Full Inspector

Allows user to inspect and update statics while the game is running



You can also see and change statics, which are otherwise pretty abstract and mysterious for beginners.



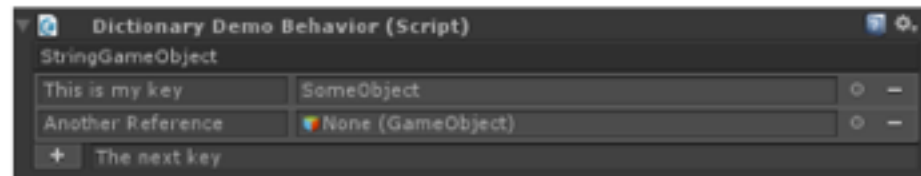
This lets them see, with their eyes, that statics are not attached to any specific game object or script instance.



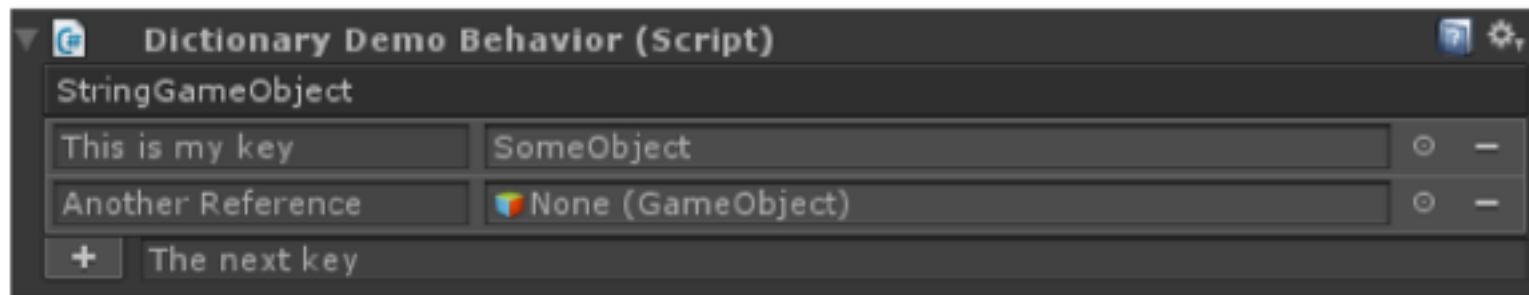
Full Inspector

Creates a visual interface for:

- generics
- interfaces
- dictionaries, structs
- properties
- delegates



More abstract things like generics and interfaces also become much clearer with a concrete, visual editor.



For example, you could create multiple kinds of dictionaries to let the students see how one data structure can represent relationships between many combinations of types.



Full Inspector

Greatly enhances directness and preserves context when working with more complicated code structures.

Helps with: the notional machine, primary task disruption, prospective memory, retrospective memory



On the Horizon/Honorable Mention

RelationsInspector
Methodic
ProGroups
ReView
Squiggle

TypeSafe
U2DEX
uIntellisense
Advanced Select

These are some other tools that I think could be adapted to be useful, or that are helpful but not revolutionary for beginners.



GDC EDUCATION
SUMMIT

Tools: Mobile and Web



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

Let's take a quick look at how these principles would apply outside the world of Unity.



Mobile and Web Tools

- Dev Tools
- Intel XDK
- CodePen, C9.io, Monaca

A couple of primary problems with mobile development are

(1) understanding the development and deployment process with an extra device, store submissions and so on, and

(2) the gap between working on your code on one device and viewing it on another, where you don't have all your tools.

These are tools I think help to keep students oriented to what they're doing. We've already talked about dev tools.



GDC EDUCATION
SUMMIT

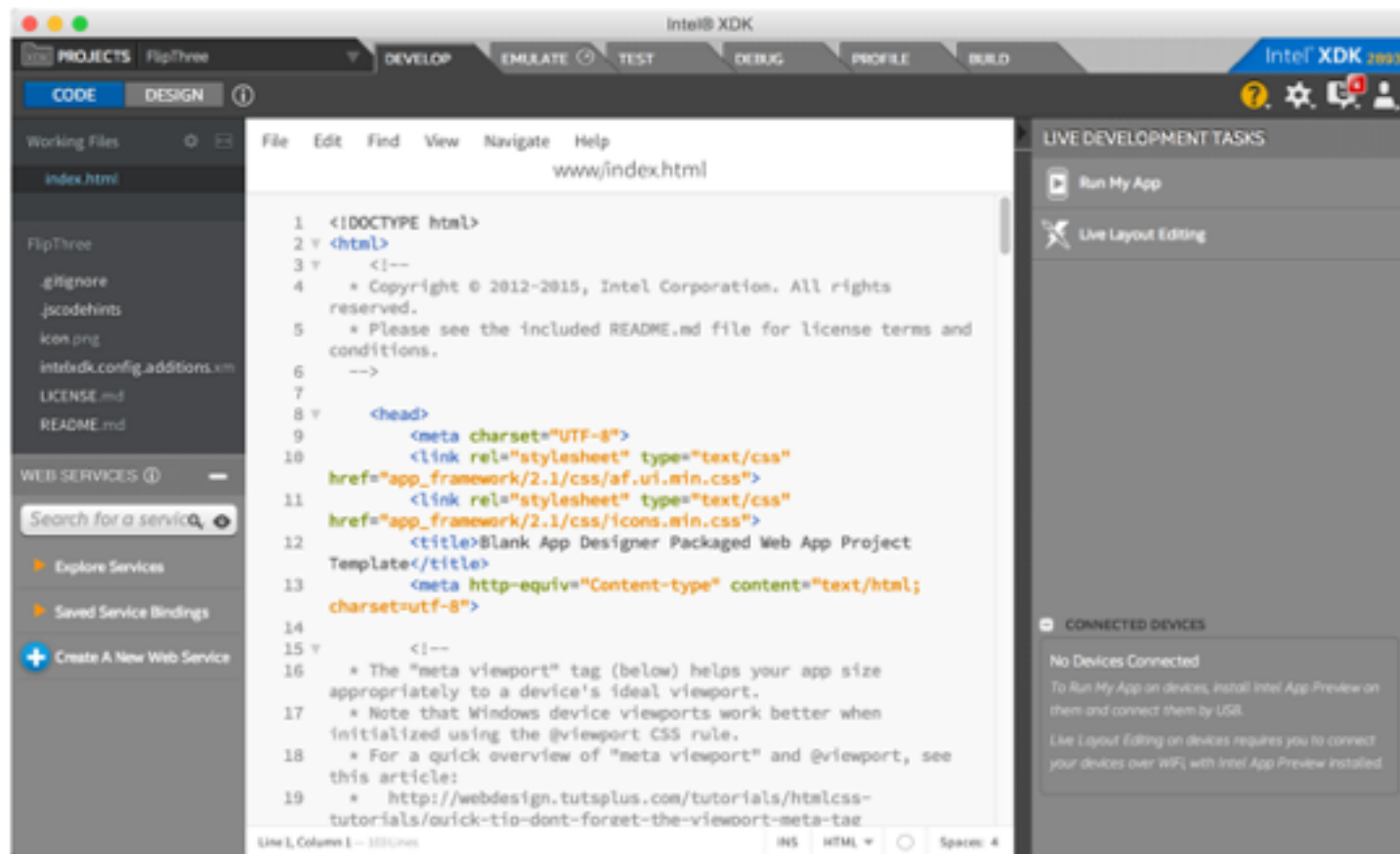
XDK

Reskinning of Atom editor, with tabs laying out the mobile development process



GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 18-19, 2016 #GDC16

My favorite currently is a tool developed by Intel called XDK. It's a set of tools for building hybrid mobile apps, including a reskinning of the Atom editor, integrated cloud builds, and a pretty solid emulator among others.



This is the “develop” view, where you manage files and write code.



The tabs at the top lay out all the steps of mobile development, including the cloud builds there at the end. It's pretty much everything but store submission.

XDK is free, and Intel is actively developing it.



Monaca

Cloud-based editor with instant update
on phone. On-device collaboration tools
like editable screen shots.

Monaca is a commercial service that provides a similar toolset. The main difference is the cloud-based code editor, allowing real-time collaboration.

I doubt anyone really likes writing code in a web browser, but they've done some work to make it less painful.



The tool also provides a live connection to the devices for debugging (via weinre).



GDC EDUCATION
SUMMIT

Conclusions



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16



Remember, our students are doing all of this at once!

1. General orientation
2. The notional machine
3. Notation
4. Plans and schemas
5. Pragmatics

Source: Benedict du Boulay,
Some Difficulties of Learning to Program



And in the process we are asking them to stretch their attention and short-term memory to levels they usually haven't gone to before.



GDC EDUCATION
SUMMIT

We can make it easier for them by finding tools that address the specific cognitive issues they are facing.



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

And we owe it to them to try!



GDC EDUCATION
SUMMIT

Credits



GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 18–19, 2016 #GDC16

I want to thank the makers of the Unity packages, USC Games and my students for helping me build this presentation.



GDC EDUCATION
SUMMIT

Thank you!

@cormorancy
mmoser@cinema.usc.edu



GDC

GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 18-19, 2016 #GDC16

And thank you for listening! I look forward to hearing your thoughts in the wrap-up area.