PROFESSOR: All right, so let's go ahead and get started. A couple quick announcements before we jump into the day. We had a couple questions about what happens if one of my team members dropped out of the class for this first project? Has anybody had that? OK, so if somebody drops out of your team at any point in next first two projects, that's OK. You're not going to be penalized for having fewer people. You're not going to be propped for having fewer people.

It's basically we're not necessarily grading the quality of the games for this first project right now. Double check the project one assignment to see what the grading rubric is for that. What we are checking is are you doing iteration? Have you tested the game multiple times? Do you have multiple versions of the game? So that's basically reflected in your design change log that you're doing for this class.

There's also an individual write up. So we're asking you to write a one page paper about your experience with the requirements for that are in the assignment. We want to know what happened when your teammate dropped out. Did it feel different? If you had two people versus three people, do you feel like you had fewer ideas? That's actually a really interesting thing for you to understand about. When you're coming together and working on group projects and you're doing collaborative projects, what does team size do to the kind of ideas you have, the number of ideas you have, and the quality of ideas you have?

In all the later projects, your team is going to be big enough that dropping one member isn't going to be huge difference. Dropping two might. So if that happens, just let us know about it. We'll talk you through it. It'll be OK. We'll get through it.

ANDREW GRANT: If I may, this is actually part of the experience of working on a team. Sometimes you lose people, and sometimes that's a problem. I like to call it the hit by a bus test. If your team can't stand losing one person-- any one person-- then you have to plan around that and make sure you can handle it. Other people like to call it the winning the lottery test because that's nicer, but basically one person gets taken out of equation. You do have to adapt to that and be ready for it to happen. Again, on this project, it's a relatively-- the teamwork is easier, because

it's a simpler deliverable and a simpler project.

PROFESSOR: But things are going to come up throughout the semester-- somebody falling sick, somebody going away for interviews, especially seniors going for job interviews. Those are things you're going to plan for to react to and have alternate plans, just in case something like that happens during the project. So just be aware of that throughout the class.

So what are we doing today? Today's a very, very busy day that's going to involve lots of movement on your part. We're going to do a workshop on version control that Drew is going to walk you through. We're going to go into discussion groups and talk about the game engine tutorials that you all completed before today, and they'll be a mechanism for talking about that. Basically people talking about forming groups in the same project and then later on forming groups where we can have people compare-- somebody who worked on Unity talking to somebody who worked on Flixel. Things like that.

After that, we're going to have project one play test. So this is going to be an informal play test. Basically it's a check. Can you play your game today? If you can't, then play what you've got, because you're going to be forced to play it. Maybe you have some rules written down. Great. Just act them out in that 15 minute block we give you to act out those rules. And maybe within that 15 minute block, come up with a few different rules or a few different pieces. So that's going to happen, and we'll go through how that's going to work later.

And then at the end of the day, we're going to do another workshop. Sara is going to lead us on that-- talking about vision statements. So this is a tool that we use to help large teams understand what it is we're actually creating. So with that I'm going to switch slides and bring it over to drew.

ANDREW GRANT: So the first thing, I see a lot of laptops, which is pretty awesome. How many of you brought a computer that actually has the code you used for you tutorials on it? That is more than half. Awesome. So you get the short version of presentation, not the long one, as soon as we get it up there and going. Let's see. Any other questions?

Who here has used version control or source control before. Most of you. Fascinating. Who here feels confident that you know why you used version control, and that it was awesome, and you're cool, and you're good at this? OK, the hands went up and started to come down slowly, which would make sense. That's what they should have done.

[LAUGHTER]

All right, so I'm not actually going to talk about the nitty gritty details of exactly which commands you type, because there's a bunch different version control systems. They all work very similarly, but you're going to basically need to read the documentation on the particular one that you're using. I recommend today, if you're familiar with one, that you stick with that if possible, but don't worry about it if you can't.

So let's see. I'm going to give a high level description of what is revision control. I'm going to use different words for it. Source control, revision control are the big ones. The basic notion is that-- and this is a very simple notion-- that you've got one copy of the source on your local machine, and you've got one copy of the source somewhere else, and that copy out there is the perfect copy. That's the true copy of what you're working on. That's the platonic ideal of your project, and your job is to copy files down from it, change them, and copy them back up to that one. Not much going on there, right? All right.

I'll get into more details on that. So we started with a whole bunch of words that go along with this topic. Revision control system, RCS, is often used as a generic term. It's also a specific piece of software that you probably won't use because it's really old. Source control, revision control, and you'll also see SCM as a common TLA for source control management.

CVS, the concurrent version system, is also older, but one you might actually see used. It's mostly command line only. SVN is a more recent open source version of CVS. I think CVS may have been, too. I'm not sure on that one. SVN is more usable, and I highly recommend it. Perforce is one that's used quite often in the games industry, and then you're going to see Git and Mercurial, which are often mentioned in the same breath. They're both what's called distributed source control. We'll get to that, but they're interface is very similar to all of the above actually. Almost all have a command line interface where you can do it everything from command line, and most of them have a GUI that you can use as well, but you're going to see a bunch of different words used to talk about the stuff.

So again, in its most basic form, you've got a server. You've got your local. You copy stuff back and forth. Pretty basic. And it's a little bit easier to see why you might want that when you've got more than one person on the team. It's a really good way to make sure your files are synchronized. It's even more important the bigger your team.

So what's really, really important is that that server copy of your code is useful. So you want to

make sure that you are only contributing to that central repository of code the useful pieces. Now, what we really want to do is that central version of the code is sort of what's considered, as I mentioned, ideal version of your code. It's authoritative. It's the one you can trust.

So that actually becomes a little bit more than a backup system, right? So a backup system is quite useful. If you lose something or you make a mistake, you want to be able to revert to a previous version of your code, and that is certainly one of the things that source control does for you. And we'll also talk about how it's a way to share your code, but it's actually more important than that. So the history of your project is one of the big things you get out of it.

When you submit your files, which can be source code file-- it could be images. It could be anything. When you submit them, you should include a comment about what the heck you've done. And this is one of the things I find new programmers have the hardest time with. In fact, I know plenty of old programmers that can't do it, but really it's useful, and the reason it's useful is all because of this. We're going to talk about different methods of communication throughout this course, and I've talked about-- or I will talk about the fact that your source code is a way to communicate with other people.

But it turns out your comment log in your version control system is also great way to communicate with your team. So for example, if you submit a whole bunch of files and it's not obvious what that did to your game, that comment can tell everyone else on your team what just happened. Maybe you rewrote the entire AI so that the bad guys now run away from the player. And it may be that someone else on your team is trying to figure out why are the bad guys' AI broken now? And the answer is it's not broken. They're just doing something different.

And when they do get that code from the server, they're going to see that comment and say, oh, I get it. The AI changed. So if you fail to report that in a Scrum-- or maybe you don't have a Scrum because you did it in the middle of the night-- this is a great way to communicate with your team.

Even more than that, that history is also really useful. Let's say you check in this code that causes bad guys to run away, and then a couple days later you find out that there's some problem in the AI where things actually crash horribly if the game sits there for an hour. Well, where do you begin looking for that bug? You may not know that it's in the AI. You may just have some clues, but you know that it changed in the last two days. This behavior started. So you can look at the log and say, what happened in the last two days?

And someone further discovers that, well, it's crashing somewhere in AI. Then you can have a hint that it's in the AI system, but more than that, you go to the exact last change in the AI system and look at those files, and that's probably where your bug is. That's pretty handy.

And let's say that you do something really, really horrible. You also get an undo button out of this. You can say, you know what? That version two days ago-- let's go back to that. That's better than what we have right now. You don't often do that, but it is really nice to have that kind of security, especially when you're working in your local version.

Let's say you're trying to find a bug. You need to change 15 files, and putting into debug statements everywhere to figure out what the heck's going on. And then you find the bug finally, and you realize it's a one line change. Well, it's easier just to revert back to the server version and make that one line change than to make sure you undo every little piece. Let the computer be your memory. Let the computer be your undo button. You don't need to rely on knowing which files you changed. It will tell you.

And this is the cool thing that is kind of scary, but pretty awesome-- is you can actually simultaneously edit the same files if they are text files. This involves a merge process afterwards which is a little bit frightening the first several times you do it, but actually the tools are pretty good. As long as you're not modifying the same line of the file, it's pretty good about figuring what should happen, but you do need to manually check it with your eyes to verify it didn't do anything stupid, but it's actually pretty common to work to be able to work with same file with two people and actually have it work out OK.

The exception to that is extremely complicated text files or binary files-- you can't merge changes to a PNG of GIF. That doesn't make sense, at least on the file level. And a lot of the data files in Unity, especially the scene files and prefab files, are going to be too complicated probably for you to actually manage to emerge. So for those, you need to be a little bit more careful about it, but it's actually a really useful tool to lean on that.

So we're going to talk about the various operations very, very quickly. When you copy things from the server, different source control systems will call it different things. You're going to see the term get to mean pull the files down from the server. Update, pull, and checkout in different systems will mean that same thing-- copy files from the server.

I also have in there the word revert, which is kind of a fancy special case. When I said you can

undo your changes-- you've got a whole bunch of files changed, and you don't like those changes-- you can revert to the server version, which is-- under the covers, what's really going on there is you're copying down from the server, but you'll call that revert because it's going to ask you, wait a minute, you changed that file. Are you sure you really want to wipe that out? And at that point, you should say yes or no depending on how sure you are that you really want to do that.

Similarly, when you have decided that you've finished with your changes locally and you want to copy them out to the server for everyone else to see, that's going to be called submit, commit, put, push or check in. Generally speaking, people talk about that copying up to the server directionally. The reason they use some fancy words like commit is that they want you to realize that this is not-- I'm just copying a file. You're copying a file for everyone to see. So you want to ensure that your changes are good.

This one's easier. If you want to find out the status of what's going on, almost all the source controls use the word status for that. But basically what you can do is you can say things like, so I've changed a bunch of files, and I forget what I've changed. I'm sure there's a lot of them. You can actually ask your source control system, what did I change? And it'll say, well, these five files have changed, and in fact, you can even get more information from that if you want to compare what those actual changes are. And diff is the term for that.

Locking files so no one else can change them is not often used in source code, but it's very useful for-- as I was talking about, there are some files that are so complicated that it can't automatically merge them for you, and the binary files as well. You may find that some source control systems don't automatically lock these for you, and so in that case, you might want to make as your team policy that you want to lock the file so that no one else messes with it. So again, I'll go back to Unity as an example. If you're changing the main scene file for your game and it's this complicated gigantic XML thing, you might lock that file so no one else changes it while you're working on it.

And then finally, merge is the last thing that you'll do. And I talk about, if you're changing one code file and someone else changes a code file while you're working on it and checks it in before you do, you're going to include their changes into your version, and that's called a merge. Most of them have tools that do it for you automatically, but if there's a conflict, it will then force you to do what's called a resolve, which is to say, hey, this file changed more than I could figure out automatically, so a human needs to interfere with it, and that point-- that's

called resolve. And that's what the process is called, too. You'll have to go through that manually a couple times.

So we talked about how this is an authoritative server, and that's true for Perforce And Subversion. And basically there's the one server that's going to be taking care of you, but it's not always true, especially if you're looking at Git and Mercurial, which use what's called distributed source control. There actually is no difference between your version of the code, and the servers version of the code, or your teammates version of the code. It's just a bunch of lists of changes, and you can send them left, and right, and upwards, and downwards. It doesn't matter.

But if you actually do this, you're in trouble, unless you're really careful about it. So whenever you do a sideways shift-- you send some code to your teammate, but you don't send it to the, server you want to make sure you know what you're doing and be very careful with it. So I recommend, if you're using a distributed source control system, until you're really comfortable with what's going on there, that you mostly treat it like a standard server based system until you're really-- and make sure if you do any other kinds of cross submits that you know what you're doing and you think about it a little bit.

The cool thing about these systems, though, is that, if I make a change and I send it to my teammate, and both of us submit it to the central server, it actually keeps track of that. It says, oh, it's the same change. I don't need to apply it twice. It's actually a really very clever technology. It's very, very cool and very powerful, but like many powerful technologies it can be very confusing. So as I say, start out here, and if you're feeling really cocky, go there.

A couple quick rules about this. They're not going to apply to you yet, because you're not yet writing code in teams, but the policy is for most teams never ever, ever break the build. And when I say break the build, I mean you want it to be the case that that one canonical copy, the perfect copy of your project, anybody can go to an empty machine, run a get, get all the files, hit the Build button for whatever environment you're using, and have it build and work perfectly. That is the goal. That's what you always want. And if you ever submit something to the server that makes that impossible, it's highly likely that you have just made everyone else in your team have to wait.

And that's a big pain because they're going to be sitting there wishing they could get work done, but they did a get to make sure they could do the merge, and then all of a sudden they

can't get any work done at all. So you want to make 100% sure that you can always, always, always do that build. Another thing-- and this is me sort of stepping aside to my coding hat-- you should always have zero errors and zero warnings if at all humanly possible in you project. A lot of times I see programmers who are just starting out who think that warnings are optional, and it's true. Unfortunately, most of them are kind of optional, but there's two problems with leaving them in your build.

Problem number one is some of them are optional, but they do in fact highlight an issue that you should think about. And if you leave it there on the screen and ignore it, you might be ignoring an important tool that'll help you find a bug. But in the second one, I think it's even more important actually-- is that as soon as you get used to seeing warnings in your build every time, you start ignoring all the warnings.

So let's say there are five warnings that pop up in your build every time you build it, and this has been true for a week and a half now. And you're writing some code and you add two warnings, and you don't notice because there are warnings. Five to seven, you didn't notice the difference, but one of two warnings that you added might be really important. So if you can get down to zero warnings, as soon as you had one morning, it will instantly hit your brain, and your brain will say, oh, wait a minute. I did something that the computer says is a little bit risky.

And in computer science, you want to avoid a little bit risky, because it's going to come back and bite you, and I promise it's easier to fix a bug 10 minutes after you wrote it than a day after you wrote it. And a week after you wrote it, it's even harder. We in this class won't get to a year after you wrote it, but sometime in your career, you probably will, and you won't enjoy it. Avoids bugs the first time if you can.

So therefore I call use a check build. On a team, you might say to someone sitting next to you, hey, can you do a get and build and make sure I didn't mess it up? That's one easy way to do it, but actually you don't need to do that. On all these source control systems you can have more than one-- what we call-- workspace. It might be your local version of the code. You can have two workspaces-- one where you're doing your work. You check it all into the central server. Second one, you go to the second workspace. You pull the code down. You do a build. You verify that everything still works, and that's a safety way that you can in a couple minutes verify that you aren't about to make the entire team wait and lose time.

Couple little tips that, again, you probably won't need a little bit, but I'm going to talk about it

very quickly now so you don't forget. All these things I'm talking about are things you're probably going to have to look up the details on how to do them, but I want you to at least have heard the words, so that when you see them on the screen, they'll look familiar to you and you'll know what you can find.

Ignore unneeded files is a really powerful thing. Almost all these systems can ignore types of files. For example, if you're working in C#, there might be some object files or whatever, and you don't want to check those in. It doesn't help you to check them in. In particular, if you're using Perforce, checking them in is actually problematic, because one of the things that Perforce does that the other systems don't do is Perforce uses the read only flag as a way on your local machine to tell you whether or not you have a file checked out for changes.

So if a file is read only, it's saying you haven't told me that you're going to change this file, so I'm not going to let you. If you try to do a build on a bunch of read only files, your build environment has two choices. It can ignore that, which is kind of scary, or it can just fail. So that's kind of annoying, but more to the point, there's no reason to check in these gigantic product files that are all part of your build when you could actually build everything from the bare source.

The only things you want to actually check in are the original source files for your project, whether that be art, audio code, whatever, but you want to be able to rebuild the entire thing. And all the intermediate files or the final result files-- generally speaking, you don't want to check those into source control, unless you've got a special build folder or something. You want to have one folder, which is purely pull down the project and build it, and it contains all the information necessary to regenerate everything that you can. So if you can ignore all the files that you don't want to check in, that's good.

This is particularly a problem with Unity. You don't want to check in anything but the asset folder and the project settings folder. There's a bunch of gigantic folders in Unity that look important, and they are important, but Unity will regenerate them for you, and you don't want to check them in.

And then finally, I talked about locking before. Lock you binary files and your big text files that don't merge well just to avoid headaches. If you don't do that and more than one person touches that file, eventually you'll get in a situation where you lose your changes because someone else checked in something over you.

Similarly with a merge be careful. If someone has checked in a-- even to text file, and you do get it down to the server, and it has to merge with your changes, and you don't do that, and you just say, nah, forget their changes. Use mine. That's going to be a problem, because someone just loss some work there. Luckily with source control you can recover from that, but it won't be smooth teamwork. No one will be happy that you wiped out their changes.

That is the quick version, and now we're going to go on to a workshop. So you should be sitting near people who used the game engine that you used. Hopefully most of you actually did this thing. So what we want to do is team up in pairs with people ideally who used your engine. If there's only three of you who used your engine, I guess you're a triplet.

And what we want to do is-- ideally two ways, but one way will do. One of you should say, yes, I've got working code. It's pretty cool. Run it. Show everyone. See it works. And then pick a source control engine of your choice. I recommend GitHub, because it's easy to get things set up. Check in your code, have the other people check it out, and see if they can build it. And then if you succeed at that, swap and do the other way around. Actually, you can do this simultaneously. Everyone can start checking in to new projects and then pull down.

Well I'll take that as a cue that I can start talking a little bit, but not for long. I promise. So the next step we want to do is discuss the game engines that you did the tutorials for. You probably did a little bit of that already because you've been hanging out with people near you who use them, but I want you to try to get into groups of five to six people. If you can't, that's fine, but try to balance it out. There probably should be about two groups per game engine. And talk about what your experiences were with the tutorials.

So things like, what was easy? What was hard? What was more difficult than you expected? Compare your experiences on different operating systems if you happen to have that experience. Compare your experiences if you used different tutorials or the same tutorial, but basically I want you to get an idea of what other people's experiences we're like compared to your own.

And the main reason for this is that you want to figure out something a little bit more about your game engine, because after this, we're going to shuffle and get into groups where we've mixed up the game engine. So for example, if you used HaxeFlixel, you'll be in a group with someone who used Flixel and someone who used Unity, and you want to compare and contrast those engines. And so think of this first discussion as a way to figure out how to talk

about your game engine and maybe learn more about other people's experiences with it.

So I believe that we're already seated mostly near each other in the game engines. I think that's all I need to say really. So split up into groups of five or six in your game engine, and if you need help with that, we'll help.

AUDIENCE: [INAUDIBLE].

ANDREW GRANT: 15. 15 minutes of that, and then we'll swap in to do multiple-- we'll help you with that part, too.

PHILIP TAN: Your attention, please. May I have your attention, please. So I think a lot of you-- I see quite a few groups are winding down on their discussion within the same game engine. Now what we're going to do is ask you to gather back in your original brainstorming groups, which is probably going to be pretty close to the combination of the teams that you are currently in. So remember when we did brainstorming on Monday? Those people? You are going to gather with them, and hopefully you're going to get a good spread of different game engines within that discussion group.

And what we want you to do is share your experience having used that game engine. First, take a quick survey around of which game engines are being represented, have a talk so that you know which game engines you're discussing. Then talk about what you found good, what you found terrible about those game engines, and what you recommend it for. OK?

PROFESSOR: As a primary motivator for this, project two starts next Monday, and you're going to choose an engine and make a game in it. And you're going to be committed to that engine for about 2 and 1/2 weeks. So that's the whole reason why we're talking about this stuff.

ANDREW GRANT: This might not need to be said, but if half of you used one game engine, do some swapping. I would like to see we can put a final couple of details as to what we discovered here. If you were to try to describe the experience of Haxe in a sentence like thumbs up, thumbs down, what was the hardest thing? Anybody? Please?

AUDIENCE: There weren't a lot of resources out there [? for it. ?]

ANDREW GRANT: Not a lot of resources. So you couldn't find help me when you got stuck?

AUDIENCE: Yeah. It's pretty [INAUDIBLE].

ANDREW GRANT: OK, I've heard that the install process is painful. Let's see. What else have I heard? Anything

else? Anything to recommend it?

AUDIENCE: Once you got it up and running, it was easy.

ANDREW GRANT: Once you got it up and running it was easy. All right, cool.

AUDIENCE: Personally I was on a Mac, and I found the installation really simple. It was just a couple of [INAUDIBLE] calls, and it seemed like it was really useful for a simple game. I'm not sure how [INAUDIBLE] will get when we were trying [INAUDIBLE], but for getting a very simple game running fast, it's nice.

ANDREW GRANT: So it sounds like there's some mixed experience on the install. Mac may be the difference. I don't know.

AUDIENCE: I was also on a Mac, and once I figured out how to install on a Mac, it was extremely simple, but the thing was the tutorial was just for Windows. Install Flash [INAUDIBLE]. Do this. Do that. I was like, OK. [INAUDIBLE].

ANDREW GRANT: So the main thing I've heard then is the support is limited. The install may or may not be hard, but if you get stuck, the support isn't going to help you through it, at least in that experience. The main reason why I think HaxeFlixel might be interesting is because I actually know that Flixel is interesting, but you guys may have a different opinion on that. So Flixel. Any other big picture takeaway for Flixel? You're all going to disagree with me now, which is fine, but say something.

AUDIENCE: I think it's support is also limited.

ANDREW GRANT: OK. And one of problems I've seen in Flixel is that it, in fact, has evolved over about five or six years now. And so if you look for a bug in Flixel and try to figure out what was going wrong, it'll say, well, in Flixel 4, you want to do this. It does end up being a little bit more complicated there, too. So in addition to there not being as much support as there is for, say, Unity, it's fragmented support. And then of course, Flash is one of those technologies that people are considering to be aging.

I will say, however, that in my experience for 2D stuff, Flixel is really, really easy to pick up once you get going, and it can really do the 2D stuff pretty well. On the other hand, we have a new contender there, Phaser, which I heard the [? trial ?] was really, really good for Phaser. Does anyone have any takeaway on Phaser? The trial was good. It handled 2D stuff pretty

well. We haven't gone in depth on collisions.

AUDIENCE: [INAUDIBLE] it was definitely good for 2D. It's very intuitive, and it's all written in JavaScript, so not too complicated. [INAUDIBLE].

ANDREW GRANT: Does anybody know if it supports TypeScript?

AUDIENCE: [INAUDIBLE].

ANDREW GRANT: It does support TypeScript. Did anyone use TypeScript for Phaser? No? Oh, you did?

AUDIENCE: Yeah.

ANDREW GRANT: Right, I just talked to you about that. In fact, you were able to use a debugger with Phaser through Visual Studio, right? That actually I find pretty encouraging. Someone's put some thought into that.

How about support? Where there are a lot of people using Phaser? Enough that you can get some help through searches? Yeah. OK, cool. So it sounds like Phaser is a-- I haven't heard about many of the HTML5 engines that I would actually want to recommend, but it sounds like so for your experiences with Phaser have been pretty positive, which is pretty cool.

AUDIENCE: I have a question about Phaser. What about the install process? Any thoughts on that? [INAUDIBLE] files could just [? be ?] [INAUDIBLE] HTML [INAUDIBLE] scripts.

ANDREW GRANT: So you just copied it in, and it works?

AUDIENCE: Yeah, so Phaser has version control pretty much [INAUDIBLE] because you have to pull everything from Git. The second tutorial of making the game was great, but as a couple of us noticed, the first tutorial had installation-- like setting up a web server and then choosing your ID was a little bit more obfuscated. It's like, why are they telling us to do these things [INAUDIBLE]?

ANDREW GRANT: Oh, right. Right. Yeah, because I do think that-- there was somebody said if you set up a web server, this is how I would do it. And I think the answer is there are easier ways to do it than they have suggested, especially if you're an MIT student. You can copy it to your web locker, and whoa, you've installed it-- is a fine way to do it.

AUDIENCE: [INAUDIBLE]. With the TypeScript installation it enables to [INAUDIBLE] copy paste in

[INAUDIBLE].

ANDREW GRANT: Cool. All right, and then I split Unity into concentrate on 2D, concentrate on 3D, but fundamentally there is a difference between those two-- that Unity pretty much-- and I heard someone even saying that Unity is designed for 3D, and they've kind of pasted 2D on top of it, which I think is an accurate criticism about usability on that. Has anyone experienced the-- trying to use the 2D, was that a problem? Are you going to still need to think in 3D to make it work?

AUDIENCE: Yeah, the other thing was the initial learning curve was pretty-- took a while to get something simple started. [? But I feel ?] like there's a lot of potential making more complex 2D stuff.

ANDREW GRANT: Sure.

AUDIENCE: It definitely doesn't feel lacking. It's like Photoshop in the sense that there's all these features, and you maybe not even 10% of them, but they're there if you need them, which may or may not be a good thing, because that can be very intimidating and sort of hard to work.

ANDREW GRANT: Sure. I agree with that 100%. There's too much there, perhaps. If you're trying to do a quick project, you might be tempted to use more than you should. Also [INAUDIBLE] some things get lost in the features.

AUDIENCE: Yeah, in some of the videos they demonstrate how you can actually have 3D objects in the 2D environment when you're developing 2D games, which I guess they were excited about. And they were just like, hey, you can do this cool thing, but I think it has the opposite effect in that it intimidates people. There's a lot of options that were built originally for 3D, and looking through menus and looking through all drop down things, most of them are things you can' touch or you can't use, because they're meant for 3D, and there's no way you can restrict the UI to be only 2D. So I feel like they should instead make it into a standalone Unity 2D software, which would be easier.

ANDREW GRANT: I understand why they don't turn it into standalone Unity 2D software, but I think your point is well taken. I think that it would help with their uptake if they were to try to limit their feature set, oddly enough. This is not the first time this has happened actually.

Way back when there was Torque, which is a 3D game engine. It was really cheap indie-- one of the first indie game engines for like $100 for use it all you want. And they started with their big fancy Torque 3D package, and everyone said, wow, this is awesome. And a bunch of

people bought it, and they never made any games, because 3D is, as we've mentioned, more than three times as hard as 2D, and so they came out with Torque 2D, and that did a lot better.

Again, because developers-- 3D is an extra hurdle to jump over, and when all the developers have to deal with a short time frame, and that 3D can get in your way. So I think the takeaway for me on Unity is basically the same, which is it's a lot. You can do a lot with it, but it's got a learning curve. It's going to be a little bit harder for you, so you might want to think twice about doing that, especially if you're concentrating on a 2D game, which you probably should be doing, if you can, this semester.

Unity also has a couple things that are a little bit different. I think mostly Flixel certainly-- and sounds like Phaser has a little Flixel blood in it. They're both designed for 2D. Very simple, straightforward implementations in an object oriented kind of way. Unity has this crazy component system where you've got one object, but you staple anything you want to to it. And it's really, really powerful, but also it takes a while to your brain around it.

So long story short, I would say that, if you can do what you want to do without taking on a big engine like Unity, I kind of recommend that, but obviously if you need to do it and will kind of-- the reason [INAUDIBLE] if you tell us that you do need to do it, but if you do need to do, it's worth considering, because I think it is a very powerful and useful friendly engine.

AUDIENCE: Which one has [? faster ?] development cycles?

ANDREW GRANT: Good question, but I'm only going to guess at this point in our experience we have not used Phaser in a class before. I know Flixel is faster than Unity. It sounds like from the feedback we're getting here that Phaser might be a good contender, so I would recommend in project two you might want to take a look at those two engines as your first go to things and check it out, because there's not good options. I would go there first, though, rather than going straight to Unity, unless you've got a team that really wants to take it on.

AUDIENCE: Are we limited to these options [? that ?] [INAUDIBLE].

ANDREW GRANT: Yes, you're limited to these options. So let's see. I think our next step then is something else.

PROFESSOR: Yes, it is. So congratulations. You've gotten through the first 2/3 of class today.

[APPLAUSE]

Yay. And that was our gaming tutorial, so keep all of these thoughts in mind as we start up project two next week. The remainder of class is going to be devoted to finishing up project-- or doing more work on project one. We're going to do two activities that we mentioned at the beginning of class. We're going to start off with play test. So if you have something testable, great. You're ahead of the game. If you don't have something testable, we're going to give you a little bit of time to get you to something testable.

What do we mean by testable? It means somebody else that is not on your team with a little bit of guidance by somebody on the team can make things happen in the game. That gives you feedback that says that was cool. That's what we were going for or that's really broken. That's awful. Let's not go that direction. That's all we're asking for you to do for class today. So at 3 o'clock, we are going to start the play test. It is 2:47 PM right now.

Ignore that first part. Basically what we're going to do is set up your in your teams. Every team should have their game set up and ready to play at 3 o'clock. The team that is closest to them-- you're going to first one team is going to play the other team's game. You can have 15 minutes to do that and then swap.

If you followed project one's examples correctly and you have a five minute game play game, so your game can be played in five minutes or less, you can get three tests done today. Awesome. But if you only get one or two, that's OK. You've got the whole weekend to finish up on that, too. So come back into this room 3 o'clock with your game set up, and we'll start the play test.

Does everybody have a playable game? Yes? Thumbs up if you're ready to go. OK. All right, so we are not just playing our games. We are play testing our games, and we are following the *Wizard of Oz* method that Philip mentioned on Monday. By that what we mean is all of your games probably have some kind of computer that's doing something in the game that the player is interacting with the computer. What that means is there is some constrained communication between the computer and the player. This is a time where you want to think about what are the things that the player knows versus what are the things that the computer is doing. There's going to be a little bit of a difference going on there.

Don't let the player know what the computer is thinking, unless it's actually part of the game. If you could imagine the computer saying here's a screen with an icon on it that says some stuff,

that's something the player knows. If it's just some calculation going on in the background, that is not something the player knows. So just think about it that way when you're doing this.

So how many teams do we have over here? One, four? OK, so pair. How many teams do we have in here. One, two, three, four, five, six. Is that right? So pair to your closest, and then pair. And how many teams over here? One, two-- is that three total? All right, instructors will test the back one. So you have 15 minutes to play one game. I'm going to come back in at 3:15 PM and tell you to switch.

SARA VERILLI: OK, so we're going to do the shift from the very fun play testing and working on games to talking a little bit about project management. And as he said, we're trying to get the project management in it bit by bit. Hopefully the point at which it becomes helpful in each of your projects. So the first thing we're going to talk about here is vision statements. Why are we talking about vision statements? I'd rather be play testing someone else's game.

The very short answer is because it's good for you and because it's good for your project. The longer answer is because a vision statement is really one of the shortest and simplest ways to make sure that you, the people on your team, and the person you are making the game for-- assuming you are making the game for a client, or an instructor or someone else who might be interested in it-- are all on the same page and understand that understand what they expect to get at the end of the project. It can't guarantee that you all believe you're getting the same thing, but you can get started by being on the same page.

Another very common early design document for games is the design doc. Has anybody here heard of design documents, I assume? Yeah. Do you know what they are? Does anyone not know what I mean when I say a design doc?

OK, traditionally-- and this is actually going further and further back into the history of game development, for which I am thankful. Traditionally, a game design doc was a document, usually a big word page that might occasionally get printed out, that pretty must described the game from start to finish-- everything that was going to happen in it, all the mechanics, all the characters, all the stories, all the interactions, everything that's going to happen. It was traditionally a very big, long file that, A, either someone spent a lot of time maintaining and nobody be spent any time reading, or, B, nobody spent any time maintaining and nobody spent any time reading. So it was not what I would call a very useful investment of time, but a lot of time tended to get invested in it.

So you may ask, why? At least partially because people feel better when things are written down. They'd like to have a place they can go to look at to say, am I doing what I think we're supposed to be doing? I'd like to go check the documents and see if that's true. Unfortunately, a really long document that tries to detail everything is not the right way to try and do this on a game. Games change too fast.

And the thing you're trying to get to, the end product of the game, is not written. It's not something that's going to be something you look at. It's something that happens in action. So our proposed substitute-- and if someone comes up with a better solution, I would love to hear it, because vision statements are also not the perfect thing. They're just better than anything else I've seen. Our solution to this is a vision statement. We tend to think that, in between a prototype, and a vision statement, and really good project communication, that's about-- and really good communication on your project, that's most of the design documentation you need.

All right, so we're going to talk about vision statements, and then we're going to make one. So there you go. I forgot that I included the animation on the slide. I did not time it right. Oh well. That's what I get for trying to be fancy.

So if a vision statement is going to define the project-- so vision statements defines the project-- or the game if it's for a game-- for the team, for the client, and its investors. What do you think it ought to include to do that? I see my chalk boards have already been used, and the chalk has been hidden. There it is.

All right, any ideas? Any suggestions? What would you like to have in such a document? You're the ones using it.

AUDIENCE: Goals. [? Scope. ?]

SARA VERILLI: Goals. Scope. I heard ideas. What kind of ideas?

AUDIENCE: [INAUDIBLE].

SARA VERILLI: The game.

AUDIENCE: Goals, ideas.

SARA VERILLI: Goals, ideas, the game. So I've got goals. I've got ideas. I've got scope. I heard game.

AUDIENCE: Theme.

SARA VERILLI: Theme. Theme, OK. Theme, that sounds like a really good idea.

AUDIENCE: Core mechanics.

SARA VERILLI: Core mechanics, yeah. Anything else?

AUDIENCE: The story if there is one.

SARA VERILLI: Story.

AUDIENCE: The audience.

SARA VERILLI: Audience. Anything else you think you really need?

AUDIENCE: [INAUDIBLE] characters.

SARA VERILLI: Characters. I think this is actually pretty darn good list. The tricky part with a vision statement is trying to get most of that in on one page or less. The other thing that you vision statement may want to include that is less relevant to the vision part of the statement and more relevant to the team is, who's working on the game? What's the name of the game? What is each person on the team going to contribute to the game, so that the vision statement becomes both a contract about what the game is going to be and what the team is going to do to get it done.

So surprise, surprise. We actually have a format for this. Actually, we have two different formats to help you get all of these things crammed in on one page and hopefully express it well enough that you will understand and be able to use it. We have what we call the back of the box vision statement and a high level design doc, and you guys are welcome to choose to use either one of those two frameworks. They're up on Stellar. They can be downloaded, and once I finish talking at you, I will pull them up so you can see them up on the screen and actually see what the files look like.

The idea behind the back of the box vision statement is it takes a marketing pitch approach. Instead of actually stating here's the goals, here's the theme, here's the core mechanic, here's the scope, the team comes together and tries to find three, or four, or five bullet points that would go on the back of the box that capture those ideas within the bullet points. You've all seen the box-- OK, maybe you haven't seen the back of a game box these days because half the games are being so downloaded from Steam, but they still have the little marketing pitches

where it says fire 38 different guns, form teams with your friends and back stab them. That sort of thing.

[LAUGHTER]

But if that is the way your team thinks and your team envisions the game back, that can be a very good way to formulate and encapsulate all of those ideas, especially if you can really drill down to what are the four things you want people to say about your game, and then every time you make a decision about your game, every time you add a mechanic, every time you add a story, a character, a setting, you make a change of the way the game works, you can go back. Read those four or five bullet points and say, is this change actually enhancing and working with that pitch? And if it is, it's probably a good change. And if it isn't, it probably is a bad change. It gives you something to really have a core of your game around.

Some people don't think like that. Some people don't think in terms of marketing pitches. They'd much rather have a better spelled out this is my goal. This is the setting. This is the mechanics. A high level design doc is the solution for those people, and in it, it asks for a goal statement for your game as a short paragraph written out, rather than as pitch.

It's got a little less bang, but it is often more precise, and both of them are equally effective. What really depends on is which one resonates with the team, which one the team thinks works well for them. That's why we have both of them. They're really covering the same information. You don't save anything by choosing one format over the other. It's just how are you presenting it to yourself and to your team.

Both documents also ask for a little bit of game play. A risk assessment-- what are the big risks on your game? They may be related to your team. Hey, we're trying an engine we never tried before. We may not be able to get to run. It may be related to the game play. We are going to try and do a mechanic no one has ever managed to do before. Whatever your big risk is. Just one or two sentences of it. And then some space for your team members to put in your names, what they're going to do, stuff like that. It's really pretty simple.

And the next step is, in fact, to do exactly that. So as a three person team, go ahead and decide which format you're going to use. Create your vision document. Decide who's going to upload it for Monday. Make sure you've got people's names on it so we know who did what, because if we're only uploading one document, that means we're only getting one document

for every three people in class, and we would like to give credit to everyone for having worked on them. And you're going to have until the end of class, so about 20 minutes. Let me see if I can find where Rick has hidden the forms. Are they--

PROFESSOR: Yeah, sorry. Don't look at my calendar.

SARA VERILLI: I'm not looking at your calendar. Oh no. I am looking at your calendar. I'm sorry. I'm not looking at the calendar. You're looking at the calendar. I'm sorry.

PROFESSOR: I don't know how well they're going to fit.

SARA VERILLI: Well, we'll do them one at a time. We'll just do them one at a time.

PROFESSOR: [INAUDIBLE] kind of look.

AUDIENCE: To clarify, the upload comes before Monday. Not before the end of class, right?

SARA VERILLI: Yeah. Yeah, the upload-- the turn in date is for Monday, before the start of class on Monday. And the main thing is you're not actually writing division doc for the prototype you just made. The prototype you made is the core-- is the core mechanics for the game you envision being made with it for project two, OK? So this is your vision document, your sort of pitchy document, for project two-- with the game that will be made for project two, OK?

All right, anyway so here's the high level design document. I think that there's both an RTF format and a Word format up there. When you upload them, we much prefer it to be in PDF. That way easy for us to read. I love trying to do this while not being able to see what I'm doing on the screen.

PROFESSOR: Look on Stellar.

SARA VERILLI: I like mice. I much prefer mice. I hate touch pads.

PROFESSOR: I also make mine kind of wonky.

SARA VERILLI: Oh, thanks.

PROFESSOR: You're trying to scroll.

SARA VERILLI: Thanks. OK, yeah, I'm trying to scroll. So this is basically the high level design doc, and what it's got-- vision, breakdown, major game play concepts, 30 seconds of game play, and then

your risks, and the team responsibility and breakdown. Let's go ahead and jump over to the vision document. And here's the vision statement one. Basically all of that-- goal, and stuff needs be contained in your back of the box advertisement. And then the rest of it is pretty much the same. Any questions? OK, if there's no questions, go.

PROFESSOR: OK, before you leave, just want to make sure we all understand what is due tomorrow, because all of our projects are pretty complex.

AUDIENCE: Tomorrow?

PROFESSOR: Monday. Monday. Thank you. You're awake. So for project one on Monday before class starts-- so before 1:00 PM-- turn into Stellar the following three things. An individual post mortem-- so this is a one page write up of your experiences working on this project. The specifications are in that project one assignment. That's so we can know what you got out of this project.

The design change log. You made some changes today. So on your change log, you should at least have one entry on it at the end today. Now you've got two entries on it hopefully. Test your game this weekend. Make some changes to the prototype this weekend. Add them to the change log. Only one per team. So appoint one person to submit all the team documents, submit it, make sure all of your names are on it.

The vision document, which you've just been working on-- one per team. And again, it's a little weird for project one because you're talking about what it would be like for project two. So under team roles and responsibilities, project two is a six person project. If you have an idea of what are the complex things that are going to be needed for project two that those mystery three people would be working on, put them down-- or in your case, four people. So imagine that you are working on this project for project two. Question?

AUDIENCE: Yeah, so for the vision statement things, are we-- for the vision documents [INAUDIBLE] write.

PROFESSOR: Can you hold on one second? I can't hear.

AUDIENCE: For the vision document that we have to hand in on Monday, are we assuming that-- for the team responsibility break down, are we assuming that we'll be working on this--

PROFESSOR: Assume you are working on this as-- yes.

AUDIENCE: Splitting up responsibility [INAUDIBLE].

PROFESSOR: Exactly right. Now, what we're doing in class-- one minute to pitch your game as a project two contender. You're going to come up here and just give us a quick elevator pitch. What is the game? Why is the game exciting? How does the game do what it's going to do? Based on this vision document. And then you're going to give us a demonstration of the prototype, and that's actually going to happen in this big-- basically how we did play testing today. You'll be demonstrating the game, and more people will play your game so that we can figure out which games we want to work on. Question there and then question there.

AUDIENCE: So does it have to be [INAUDIBLE].

PROFESSOR: Has to be paper. This project one is paper. Project one is non-digital.

AUDIENCE: So when do we know which projects everybody is going to be working on.

PROFESSOR: By the end of class on Monday, all of the teams will be formed for project two. So project two-- again, it's a single player game, project two. Some of you have made multiplayer games. That's OK. Imagine what the game is going to be like for a single player. Are those other players AI? Make them AI, but project two [? as a ?] digital game is only a single player game. Do not turn in a multiplayer game for project two. For project four, we're going to relax that, but for two and three, just to make sure we're getting some games in here let's make them single player. Again, five minutes of game play.

Again, planning for randomness. Is your game right now-- if project one doesn't quite fit that planning for randomness theme, that's OK. In your vision document, what would you do for project two to beef that up to make that really pop out. And again, project two-- those four engines that we worked on today. Not photon, Phaser. I miswrote that. Those are the game engines we're working for project two. So I kept you a little too late. Sorry. If you want to keep working in the room, I don't think anybody has it afterwards, so feel free to work. Feel free to leave.