PROFESSOR: Mike Acton today who has done a lot of programming on cell and also done a lot of game development. He came from California just like this today and [OBSCURED].

MIKE ACTON: Yeah, it's really cool. I'll tell you.

PROFESSOR: He's going to talk about what it's really like to use cell and PS3 and what it's like to program games. So I think it's going to be a fun lecture.

MIKE ACTON: All right, so anyway, I'm the Engine Director at Insomniac Games. I've only recently taken that position, previously I was working on PS3 technology at Highmoon studios, which is with vin studios. And I've worked at Sony. I've worked for Titus, and Bluesky Studios in San Diego. And I've been doing game development, 11, 12 years. Before that I was working in simulation.

So, the PlayStation 3 is a really fun platform. And I know you guys have been working on cell development. Working with the PS3 under Linux. Working as developers for the PS3 is definitely a different environment from that. I think I'm going to concentrate more on the high-level aspects of how you design a game for the cell. And how the cell would impact the design, and what are the elements of the game. Just stuff that you probably haven't had as part of this course that you might find interesting. And you can feel free to interrupt me at any time with questions or whatever you'd like.

So just, I wanted to go over, briefly, some of the different types of game development and what the trade-offs for each one of them are. Casual games, console games, PC games, blah, blah, blah. Casual games, basically, are the small, simple games that you would download on the PC, or you would see on Yahoo or whatever. And those generally don't have really strict performance requirements. Where a console game, we have this particular advantage of knowing the hardware and the hardware doesn't change for an entire cycle. So for five, six years, we have exactly the same hardware. And that's definitely an advantage from a performance point anyway. In this case, it's PlayStation 3.

As far as develpment priorities, development priorities for a console game-- and especially a

PS3 game-- development would be completely different than you might find on another kind of application. We don't really consider the code itself important at all. The real value is in the programmers. The real value is in the experience, and is in those skills. Code is disposable. After six years, when we start a new platform we pretty much have to rewrite it anyway, so there's not much point in trying to plan for a long life span of code. Especially when you have optimized code written in assembly for a particular platform. And to that end, the data is way more significant to the performance than the code, anyway. And the data is specific to a particular game. Or specific to a particular type of game. And certainly specific to a studios pipeline. And it's the design of the data where you really want to spend your time concentrating, especially for the PS3.

Ease of programming-- whether or not it's easier to do parallelism is not a major concern at all. If it's hard, so what? You do it. That's it.

Portability, runs on PlayStation 3, doesn't run anywhere else. That's a non-concern. And everything is about performance. Everything we do. A vast majority of our code is either hand up from IC, or assembly, very little high level code. Some of our gameplay programmers will write C plus plus for the high level logic, but as a general, most of the code that's running most the time is definitely optimized. Yeah?

AUDIENCE: If programming is a non-priority, does that mean to say that if you're developing more than one product or game, they don't share any common infrastructure or need?

MIKE ACTION: No, that's not necessarily true. If we have games that share similar needs, they can definitely share similar code. I mean, the point I'm trying to make is, let's say in order to make something fast it has to be complicated. So be it, it's complicated. Whether or not it's easy to use for another programmer is not a major concern.

AUDIENCE: So you wish it was easier?

MIKE ACTION: No. I don't care. That's my point.

AUDIENCE: Well, it's not as important as performance, but if someone came to you with a high performance tool, you would like to use it?

MIKE ACTION: I doubt they could. The highest performance tool that exists is the brains of the programmers on our team. You can not create-- it's theoretically impossible. You can not out perform people

who are customizing for the data, for the context for the game. It is not even remotely theoretically possible.

AUDIENCE: That didn't come out in assembly programming for general purpose but we'll take this offline? And there was a day when that was also true for general preferred cleary at the time, but it's no longer true.

MIKE ACTION: It is absolutely--

AUDIENCE: So the average person prefers to go on -- take it offline.

MIKE ACTION: Average person. We're not the average people. We're game programmers. Yeah?

AUDIENCE: So does cost ever become an issue? I mean--

MIKE ACTION: Absolutely, cost does become an issue. At a certain pont, something is so difficult that you either have to throw up your hands or you can't finish in time.

AUDIENCE: Do you ever hit that point?

MIKE ACTION: Or you figure out a new way of doing it. Or do a little bit less. I mean we do have to prioritize what you want to do. At the end of the day you can't do everything you want to do, and you have another game you need to ship eventually, anyway. So, a lot of times you do end up tabling things. And say, look we can get 50% more performance out of this, but we're going to have to table that for now and scale back on the content. And that's why you have six years of development. You know, maybe in the next cycle, in the next game, you'll be able to squeeze out a little bit more. And the next one you squeeze out a little bit more. That's sort of this continuous development, and continuous optimization over the course of a platform. And sometimes, yeah, I mean occasionally you just say yeah, we can't do it or whatever, it doesn't work. I mean, that's part and parcel of development in general. Some ideas just don't pan out. But--

AUDIENCE: Have you ever come into a situation where programming conflicts just kills a project? Like Microsoft had had a few times, like they couldn't put out [OBSCURED]. Couldn't release for--

MIKE ACTION: Sure, there's plenty of studios where the programming complexity has killed the studio, or killed the project. But I find it hard to believe-- or it's very rarely-- because it's complexity that has to do specifically with optimization. That complexity usually has to do with unnecessary

complexity. Complexity that doesn't achieve anything. Organization for the sake of organization. So you have these sort of over designed C plus plus hierarchies just for the sake of over organizing things. That's what will generally kill a project. But in performance, the complexity tends to come from the rule set-- what you need to do to set it up. But the code tends to be smaller when it's faster. You tend to be doing one thing and doing one thing really well. So it doesn't tend to get out of hand. I mean, it occasionally happens but, yeah?

AUDIENCE: So in terms of the overall cost, how big is this programming versus the other aspect of coming up with the game? Like the game design, the graphics--

AUDIENCE: So, for example, do you have--

MIKE ACTION: OK, development team? So--

AUDIENCE: So how many programmers, how many artists, how many--

PROFESSOR: Maybe, let's-- so for example, like, now it's like, what, $20 million to deliver a PS3 game?

MIKE ACTION: Between $10 and $20 million, yeah.

PROFESSOR: So let's develop [OBSCURED]

MIKE ACTION: So artists are by far the most-- the largest group of developers. So you have animators and shade artists, and textual artists, and modelers, and enviromental artists, and lighters. And so they'll often outnumber programmers 2:1. Which is completely different than-- certainly very different from PlayStation and the gap is much larger than it was on PlayStation 2.

With programmers you tend to have a fairly even split or you tend to have a divide between the high level game play programmers and the low level engine programmers. And you will tend to have more game play programmers than engine programmers, although most-- the majority of the CPU time is spent in the engine code. And that partially comes down to education and experience. In order to get high performance code you need to have that experience. You need to know how to optimize. You need to understand the machine. You need to understand the architecture and you need to understand the data. And there's only so many people that can do that on any particular team.

AUDIENCE: Code size wise, How is the code size divided between game playing and AI, special effects?

MIKE ACTION: Just like, the amount of code?

AUDIENCE: Yeah, it should be small I guess.

MIKE ACTION: Yeah, I mean, it's hard to say. I mean, because it depends on how many features you're using. And, you know sort of the scope of the engine is how much is being used for a particular game, especially if you're targeting multiple games within a studio. But quite often-- interestingly enough-- the game play code actually overwhelms the engine code in terms of size and that is back to basically what I was saying that the engine code tends to do one thing really well or a series of things really well.

AUDIENCE: Game play code also C plus plus?

MIKE ACTON: These days it's much more likely that game play code is C plus plus in the high level and kills performance and doesn't think about things like cache. That's actually part of the problem with PlayStation 3 development. It was part of the challenge that we've had with PlayStation 3 development. Is in the past, certainly with PlayStation 2 and definitely on any previous console, this divide between game play and engine worked very well. The game play programmers could just call a function and it did its fat thing really fast and it came back and they continue this, but in a serial program on one process that model works very well. But now when the high level design can destroy performance but through the simplest decision, like for example, in collision detection if the logic assumes that the result is immediately available there's virtually no way of making that fast. So the high-level design has to conform to the hardware. That's sort of a challenge now, is introducing those concepts to the high-level programmer who haven't traditionally had to deal with it.

Does that answer that question as far as the split?

AUDIENCE: You said 2:1, right?

MIKE ACTON: Approximately 2:1, artist to programmers. It varies studio to studio and team to team, so it's hard to say in the industry as a whole.

So back basically to the point of the code isn't really important. The code itself doesn't have a lot of value. There are fundamental things that affect how you would design it in the first place. The type of game, the kind of engine that would run a racing game is completely different than the kind of engine that would run a first person shooter. The needs are different, the optimizations are totally different, the data is totally different, so you wouldn't try to reuse code from one to the other. It just either wouldn't work or would work really, really poorly.

The framerate-- having a target of 30 frames per second is a much different problem than having a target of 60 frames per second. And in the NCSC territories those are pretty much your only two choices- 30 frames or 60, which means everything has to be done in 16 and 2/3 milliseconds. That's it, that's what you have-- 432 milliseconds. Of course, back to schedule and cost, how much? You know, do you have a two year cycle, one year cycle, how much can you get done?

The kind of hardware. So taking for example, an engine from PlayStation 2 and trying to move it to PlayStation 3 is sort of a lost cause. The kind of optimizations that you would do, the kind of parallelization you would do is so completely different, although there was parallelization in PlayStation 2, the choices would have been completely different. The loss from trying to port it is much, much greater than the cost of just doing it again.

AUDIENCE: [OBSCURED]

MIKE ACTON: I don't know that there's an average. I mean, if you wanted to just like homogenize the industry, it's probably 18 months.

The compiler actually makes a huge, significant difference in how you design your code. If you're working with GCC and you have programmers who have been working with GCC for 15 years abd who understand the intricacies and issues involved in GCC, the kind of code you would write would be completely different than if you were using XLC for example, on the cell. There are studios-- Insomniac doesn't, but there are other studios who do cross platform design. So for example, write Playstation 3 games and Xbox 360 games and/or PC titles. At the moment, probably the easiest approach for that is to target the PlayStation 3. So you have these sort of SPU friendly chunks of processing SPU chunks, friendly chunks of data and move those onto homogenous parallel processors. It's not the perfect solution, but virtually all cross platform titles are not looking for the perfect solution anyway because they cannot fully optimize for any particular platform.

I wanted to go through-- these are a basic list of some of the major modules that a game is made out of. I'll go through some of these and explain how designing on the cell impacts the system. I'm not going to bother reading them. I assume you all can read. So yeah, I'm going to go over the major system, a few of the major systems and then we're going to drive a little bit into a specific system, in this case an animation system. And just talk it through, basically you

see how each of these steps are affected by the hardware that we're running on it. So just to start with when you're designing a structure, any structure, anywhere-- the initial structure is affected by the kind of hardware that you're running. And in this particular case on the SPU and there are other processors where this is equally true, but in this conventional structure where you say structure class or whatever and you have domain-constrained structures are of surprisingly little use.

In general, the data is either compressed or is in a stram or is in blocks. It's sort of based on type, which means that there's no fixed size struct that you could define anyway. So as a general rule, the structure of the data is defined within the code as opposed to in a struct somewhere. And that's really to get the performance from the data, you group things of similar type together rather than for example, on SPU, having flags that say this is of type A and this is of type B. Any flag implies a branch, which is-- I'm sure you all know at this point-- is really poor performing on SPU. So basically, pull flags out, resort everything and then move things in streams. And all of these types are going to be of varying sizes. In which case there's very little point to define those structures in the first place because you can't change them. And the fact that you're accessing data in quadwords anyway. You're always either loading and storing in quadwords, not on scalars, so having scalar fields in a structure is sort of pointless. So again, only SPU generally speaking structures are of much use.

When you go to define structures in general you need to consider things like the cache, the TLB, how that's going to affect you're reading out of the structure or writing to the structure. More to the point of you cannot just assume that if you've written some data definition that you can port it to another platform. It's very easy to be poorly, a performing platform to platform. In this case, when we design structures you have to consider the fundamental units of the cell. The cache line is a fundamental unit of the cell. Basically, you want to define things in terms of 128 bytes of wide. What can you fit in there because you read one you read them all, so you want to pack as much as possible into 128 bytes and just deal with that as a fundamental unit.

16 bytes, of course, you're doing load and stores through quadword load and store. So you don't want to have little scalar bits in there that you're shuffling around. Just deal with it as a quadword. And don't deal with anything smaller than that. So basically the minimum working sizes, in practice, would be 4 by 128 bits wide and you can split that up regularly however you want. So to that point I think-- here's an example-- I want to talk about a vector class.

Vector class js usually the first thing a programmer will jump onto when they might want to

make something for games. But in real life, it's probably the most useless thing you could ever write. It doesn't actually do anything. We have these, we know the instruction set, it's already in quadwords. We know the loads and stores, we've already designed your data so it fits properly. This doesn't give us anything. And it potentially makes things worse. Allowing component access to a quadword, especially on the PPU is ridiculously bad. In practice, if you allow component access, high-level programs will use component access. So if you have a vector class that says get x, get y, whatever, somebody somewhere is going to use it, which means the performance of the whole thing just drops and it's impossible to optimize. So as a general rule, you pick your fundamental unit. In this case, the 4 by 128 bit unit that I was talking about and you don't define anything smaller than that. Everything is packed into a unit about that size. And yes, in practice there'll be some wasted space at the beginning or end of streams of data, groups of data, but it doesn't make much difference. You're going to have that wasted space if you are-- you're going to have much more than that in wasted space if you're using dynamic memory, for example, which when I get to it-- I don't recommend you use either.

So some things to consider when you're doing this sort of math transformation anyway is, are you going to do floats, double, fixed point? I mean, doubles write out. There's no point. Regardless of the speed on the SPU of a double, there's no value in it for games. We have known data, so if we need to we can renormalize a group of around a point and get into the range of a floating point. It's a nonissue. So there's no reason to waste the space in a double at all, unless it was actually faster, which it isn't. So we don't use it.

Sort of the only real problematic thing with the SPU floating point is its format and not supporting denormalized numbers becomes problematic, but again, you can work around it by renormalizing your numbers within a known range so that it won't to get to the point where it needs to denormalize-- at least for the work that you're actually doing. Yeah?

AUDIENCE: [OBSCURED]

MIKE ACTON: Every program will write its own vector class. And I'm saying that that's a useless exercise. Don't bother doing it. Don't use anybody else's either. If you're writing for the cell-- if you're writing in C you have the SI intrinsics. They're already in quadwords, you can do everything you want to do and you're not restricted by this sort of concept of what a vector is. If you want to deal with, especially on the SPU where you can freely deal with them as integers or floats or whatever seamlessly without cost, there's plenty that you can do with the floating point number

if you treat it as an integer. And when on either AltiVec or the SPU where you can do that without cost there's a huge advantage to just doing it straight.

AUDIENCE: [OBSCURED]

MIKE ACTON: Well, I'm saying write it in assembly. But if you have to, use the intrinsics. But certainly don't write a vector class.

So memory management. Static allocations always prefer the dynamic. Basically, general purpose dynamic memory allocation, malloc free, whatever has just absolutely no place in games. We don't have enough unknowns for that to be valuable. We can group our data by specific types. We know basic ranges of those types. The vast majority of the data is known in advance, it's actually burned onto the disk. We can actually analyze that. So most of our allocations tend to calculate it in advance. So you load the level and oftentimes you just load memory in off the disc into memory and then fix up the pointers. For things that change during the runtime, just simple hierarchical allocators, block allocators where you have fixed sizes is always the easiest and best way to go. These are known types of known sizes. The key to that is to organize your data so that's actually a workable solution. So you don't have these sort of classes or structures that are dynamically sized. That you group them in terms of things that are similar. Physics data here and AI data is separately here in a separate array. And that way those sort of chunks of data are similarly sized and can be block allocated without any fragmentation issues at all.

Eventually you'll probably want to design an allocator. Things to consider are the page sizes. That's critically important, you want to work within a page as much as you possibly can. So you want to group things, not necessarily the same things, but the things that will be read together or written together within the same page. So you want to have a concept of the actual page up through the system. Probably the most common mistake I see in a block allocator, so somebody says-- everybody knows what I mean by block allocator? Yeah? OK. So the most common mistake I see people make is that they do least recently used. They just grab the most least recently used block and use that when summoning a request. That's actually pretty much the worst thing you can possibly do because that's the most likely thing to be called. That's the most likely thing to be out of cache, both out of L1 and L2. Just the easiest thing you can do to change that is just use most recently used. Just go up the other way. I mean, there are much more complicated systems you can use, but just that one small change where you're much more likely to get warm data is going to give you a big boost. And again, like I

said, use hierarchies of allocations instead of these sort of static block allocations. Instead of trying to have one general purpose super mega allocator that does everything.

And again, if it's well planned, fragmentation is a non-issue, it's impossible. Cache line, oh, and probably another important concept to keep in mind as you're writing your allocator is the transfer block size of the SPU. If you have a 16K block and the system is aware of fixing K blocks then there are plenty of cases where you don't have to keep track of-- in the system-- the size of things. It's just how many blocks, how many SPU blocks do you have? Or what percentage of SPU blocks you have? And that will help you can sort of compress down your memory requirements when you're referring to blocks and memory streams and memory.

AUDIENCE: About the memory management for data here, you also write overlay managers for code for the user?

MIKE ACTON: Well, it basically amounts to the same thing. I mean, the code is just data, you just load it in and fix up the pointers and you're done.

AUDIENCE: I was just wondering whether IBM gives you embedding --

MIKE ACTON: We don't usse any of the IBM systems at all for games. I know IBM has an overlay manager as part of the SDK.

AUDIENCE: Well, not really. It's --

MIKE ACTON: Well, they have some overlay support, right? That's not something we would ever use. And in general, I mean, I guess that's probably an interesting question of how--

AUDIENCE: So it's all ground up?

MIKE ACTON: What's that?

AUDIENCE: All your development is ground up?

MIKE ACTON: Yeah, for the most part. For us, that's definitely true. There are studios that, especially cross platform studios that will take middleware development and just sort of use it on a high-level. But especially when you're starting a first generation platform game, there's virtually nothing there to use because the hardware hasn't been around long enough for anybody else to write anything either. So if you need it, you write it yourself. Plus that's just sort of the general theme of game development. It's custom to your situation, to your data. And anything that's general

purpose enough to sell as middleware is probably not going to be fast enough to run a triple A title. Not always true, but as a general rule, it's pretty valid.

OK, so-- wait, so how'd I get here? All right, this is next. So here's another example of how the cell might affect design. So you're writing a collision detection system. It's obvious that you cannot or should not expect immediate results from a collision detection system, otherwise you're going to be sitting and syncing all the time for one result and performance just goes out the window, you may as well just have a serial program. So you want to group results, you want to group queries and you want potentially, for those queries to be deferred so that you can store them, you can just DMA them out and then whatever process needed then we'll come back and grab them later.

So conceptually that's the design you want to build into a collision detection system, which then in turn affects the high-level design. So AI, scripts, any game code that might have previously depended on a result being immediately available, as in they have characters that shoot rays around the room to decide what they're going to do next or bullets that are flying through the air or whatever, can no longer make that assumption. So they have to be able to group up their queries and look them up later and have other work to do in the meantime. So this is a perfect example of how you cannot take old code and move it to the PS3. Because old code, serial code would have definitely assumed that the results were immediately available because honestly, that was the fastest way to do it.

So on a separate issue, we have SPU decomposition for the geometry look up. So from a high-level you have your entire scene in the level of the world or whatever and you have the set of queries in the case of static-- did I collide with anything in the world? Or you have a RAID that, where does this RAID collide with something in the world? And so you have this problem of you have this large sort of memory database in main RAM and you have the smallest spew, which obviously cannot read in the whole database, analyze it, and spit out the result. It has to go back and forth to main RAM in order to build its result. So the question is how do you decompose the memory in the first place to make that at least somewhat reasonably efficient?

The first sort of instinct I think, based on history is sort of the traditional scene graph structures like BSP tree or off tree or something like that. Particularly, on the SPU because if TLB misses that becomes really expensive, really quickly when you're basically hitting random memory on every single node on the tree. So what you want to do is you want to make that hierarchy as

flat as you possibly can. If the leafs have to be bigger that's fine because it turns out it's much, much cheaper to stream in a bigger group of-- as much data as you can fit into the SPU and run through it and make your decisions and spit it back out than it is to traverse the hierarchy. So basically, the depth of your hierarchy in your scene database is completely determined by how much data you can fit into the SPU by the maximum size of the leaf node. The rest of the dep is only because you don't have any other choice. You know, And basically the same thing goes with dynamic geometry as you have geometry moving around in the scene, characters moving around in the scene-- they basically need to update themselves into their own database, into their own leaves and they'll do this in groups. And then when you query, you basically want it to query as many of those as possible, as you can possibly fit in at once. So you could have sort of a broad faced collision first, where you have all of the groups of characters that are potentially maximum in this leaf, so bound and box or whatever. So even though you could in theory, in principle narrow that down even more, the cost for that, the cost for the potential memory miss for that is so high that you just want to do a linear search through as many as you possibly can on SPU. Does that make sense?

Procedural graphics-- so although we have a GPU on the PlayStation 3, it does turn out that the SPU is a lot better at doing a lot of things. Things basically where you create geometry for RSX to render. So particle system, dynamic particle systems. Especially where their systems have to interact with the world in some way, which will be much more expensive on the GPU. Sort of a dynamic systems like cloth.

Fonts is actually really interesting because typically you'll just see bitmap fonts in which case are just textures. But if you have a very complex user interface then just the size of the bitmap becomes extreme and if you compressed them they look terrible, especially fonts. Fonts need to look perfect. So if you do do procedural fonts, for example, two type fonts, the cost of rendering a font actually gets significant. And in this case, the SPU is actually a great use for rendering a procedural font.

Rendering textures is basically the same case as font. Procedural textures like if you do noise-based clouds or something like that. And parametric geometry, it's like nurbs or subdivision services or something like that, is a perfect case for the SPU. Is there a question?

Geometry database, OK. First thing scene graphs are worthless. Yeah?

AUDIENCE: So of those sort of differnet conceptualized paths, are you literally swapping code in and out of

the SPUs with the data many times per frame? Or is it more of a static---

MIKE ACTON: OK, that's an excellent question. It totally depends. I mean, in general through a game or through, at least a particular area of a game the SPU set up is stable. So if we decide you're going to have this SPU dedicated to physics for example, it is very likely that that SPU is stable and it's going to be dedicated physics, at least for some period of time through the level or through the zone or wherever it is. Sometimes through an entire game. So there are going to be elements of that where it's sort of a well balanced problem. There's basically no way you're going to get waste. It's always going to be full, it's always going to be busy. Collision detection and physics are the two things that you'll never have enough CPU to do. You can always use more and more CPU. And basically, the rest can be dynamically scheduled.

And the question of how to schedule it, is actually an interesting problem. It's my opinion that sort of looking for the universal scheduler that solves all problems and magically makes everything work is a total lost cause. You have more than enough data to work with and in your game to decide how to schedule your SPUs basically, manually. And it's just not that complicated. We have six SPUs. How to schedule six SPUs is just not that complicated a problem, you could write it down on a piece of paper.

OK, so scene graphs are almost always, universally a complete waste time. They store way too much data for no apparent reason. Store your databases independently based on what you're actually doing with them, optimize your data separately because you're accessing it separately. The only thing that should be linking your sort of domain object is a key that says all right, we'll exist in this database and the database and this database. But to have this sort of giant structure that keeps all of the data for each element in the scene is about the poorest performing you can imagine for both cache and TLB and SPU because you can't fit it in individual node on the SPU. I think I covered that.

Here's an interesting example, so what you want to do is if you have the table of queries that you have-- bunch of people over the course of a frame say I want to know if I collided with something. And then if you basically make a pre-sort pass on that and basically, spatially sort these guys together, so let's say you have however many you can fit in a SPU. So you have four of these queries together. Although they might be a little further apart then you would hope, you could basically create a baling box through a single query on the database that's the sum of all of them and then as I said, now you have a linear list that you can just stream through for all of them. So even though it's doing more work for any individual one, the

overhead is reduced so significantly that the end result is that it's significantly faster. And that's also what I mean by multiple simultaneous lookups. Basically you want to group queries together, but make sure that there's some advantage to that. By spatially pre-sorting them there is an advantage to that because it's more likely that they will have overlap in your queries.

So game logic. Stuff that the cell would affect in game logic. State machines are a good example. If you defer your logic lines and defer your results, SPUs are amazingly perfect for defining state machines. If you expect your logic lines to be immediately available across the entire system, SPU is absolutely horrid. So if you basically write buffers into your state machines or your logic machines then each SPU can be cranking on multiple state machines at once where all the input and all the output lines are assumed to be deferred and it's just an extremely straightforward process.

Scripting, so scripting things like-- I don't know, lewis script or C script or something like that. I mean, obviously the first thing to look at is the size of the interpreter. Will it fit into an SPU to begin with? Another option to consider is, can it be converted into SPU code, either offline or dynamically? Because you'll find that most off the shelf scripting languages are scalar, sequential scripting languages. So all of a P code within the scripting language itself basically defines scalar access. So not only are you switching on every byte to every two bytes or whatever, so it's sort of poorly performing code from an SPU point of view, but it's also poorly performing code from a memory point of view. So I guess the question is whether or not you can optimize the script itself and turn turn it into SPU code that you can then dynamically load or come up with a new script that's just much more friendly for the SPUs.

Another option if you have to use a single, sort of scalar scripting language like lua or C script or whatever, if you can run multiple streams simultaneously so that while you're doing these sort of individual offline memory lookups and reads and writes to main memory, that once one blocks you can start moving on another one. As long as there's no dependencies between these two scripts we should be able to stream them both simultaneously.

Motion control actually turns out to be a critical problem in games in general that's often overlooked. It's who controls the motion in the game. Is is the AI? So is it the controller in the case of the player? I say, push forward, so the guy moves forward. Is that really what controls it? Or is it the physics? So all the AI does is say, I want to move forward, tells the physic system I want to move forward and the physics tries to follow it. Or is it the animation? That

you have the animators actually put translation in the animation, so is that translation the thing that's actually driving the motion and everything else is trying to follow it? Turns out to be a surprisingly difficult problem to solve and every studio ends up with their own solution. I forget what point I was making on how the cell affected that decision. But--

AUDIENCE: [OBSCURED]

MIKE ACTON: I think the point, probably that I was trying to make is that because you want everything to be deferred anyway, then the order does become a clearer sort of winner in that order. Where you want the immediate feedback from the controls, the control leads the way. You have the physics, which then follows, perhaps, even a frame behind that to say how that new position is impacted by the physical reality of the world. And then potentially a frame behind that or half a frame behind that you have the animation system, which in that case would just be basically, a visual representation of what's going on rather than leading anything. It's basically an icon for what's happening in the physics and the AI. The limitations of your system are that it has to be deferred and that it has to be done in groups. Basically, some of these sort of really difficult decisions have only one or two obvious answers.

All right, well I wanted to dig into animation a little bit, so does anybody have any questions on anything? Any of the sort of the high-level stuff that I've covered up to this point? Yeah?

AUDIENCE: So does the need for deferral and breaking into groups and staging, does this need break the desire for the higher-level programmers to abstract what's going on at the data engine level? Or is that not quite the issue?

MIKE ACTON: Well, let's say we get a game play programmer, right? Fresh out of school, he's taught in school, C plus plus in school. Taught to decompose the world into sort of the main classes and that they all communicate through each other maybe through messaging. All right, well the first thing we tell him is that all that is complete crap. None of that will actually work in practice. So in some sense, yes, there is a sort of tendency for them to want this interface, this sort of clean abstraction, but abstraction doesn't have any value. It doesn't make the game faster, it doesn't make the game cheaper. It doesn't make--

AUDIENCE: [OBSCURED]

PROFESSOR: There's a bit of a religious. Let's move on. He has a lot of other interesting things to say. And we can get to that question--

AUDIENCE: It sounds like there's two completely different communities involved in the development. There's the engine developers and there's the higher-level--

MIKE ACTON: That's a fair enough assessment. There are different communities. There is a community of the game play programmers and the community of engine programmers. And they have different priorities and they have different experiences. So yeah, in that way there is a division.

PROFESSOR: I will let you go on. You said you had a lot of interesting information to cover.

MIKE ACTON: OK.

AUDIENCE: I can bitch about that. Don't worry.

MIKE ACTON: So just to get into animation a little bit. Let's start with trying to build a simple animation system and see what problems come creep up as we're trying to implement it on the cell. So in the simplest case we have a set of animation channels defined for a character, which is made up of joints. We're just talking about sort of a simple hierarchical transformation here. And some of those channels are related. So in the case of rotation plus translation plus scale equals any individual joint. So the first thing, typically that you'll have to answer is whether or not you want to do euler or quaternion rotation.

Now the tendency I guess, especially for new programmers is to go with quaternion. They're taught that gimbal lock is a sort of insurmountable problem that only quaternion solves. That's just simply not true. I mean, gimbal lock is completely manageable in practice. When you're trying to rotate on three axes and two axes rotate 90 degrees apart and the third axis can't be resolved or 180 degrees apart. So you can't resolve one of the axes, right?

AUDIENCE: [OBSCURED]

MIKE ACTON: Yeah, it's where it's impossible to resolve one of the axes and that's the nature of euler sort of rotation. But sort of a quaternion rotation completely solves that mathematical problem, it's always resolvable and it's not very messy at all. I mean, from a sort of C programmers perspective, it looks clean, the math's clean, everything's clean about it. Unfortunately, it doesn't compress very well at all. Where if you used euler rotation, which basically just means that the individual rotation for every axis. So x rotation, y rotation, z rotation. That's much, much more compressible because each one of those axes can be individually compressed. It's very unlikely that you're always rotating all three axes, all the time, especially in a human

character. It's much more likely that only one axis is rotating on any one given time and so that makes it-- just without any change, without any additional compression-- it tends to make it about 1/3 of the size.

AUDIENCE: [OBSCURED]

MIKE ACTON: The animation data. So you have this frame of animation, which is all these animation channels, right? And then over time you have these different frames of animation, right? If you store-- for every joint, for every rotation you store a quaternion over time, it's hard to compress across time because you're basically, essentially rotating all three axes, all the time. Well, with-- yeah? All right. So let's say, of course, the next step is how do we store the actual rotation itself? Do we store it in cloth, double, half precision, fixed point precision? Probably the national tendency at this point would be to store it in a floating point number, but if you look at the actual range of rotation, which is extremely limited on a character, on any particular joint there are very few joints that would even rotate 180 degrees. So a floating point is overkill, by a large margin on rotation-- for the range.

For the precision, however it's fairly good. Especially if you're doing very small rotations over a long period of time. So probably a more balanced approach would be to go with a 16 bit floating point from a half format where you keep most of the precision, but you reduce the range significantly. There's also the potential for going with an 8 bit floating point format depending on the kind of animation that you're doing. And I'll probably have this on another slide, but it really depends on how close-- how compressible a joint is depends on how close to the root it is. The further away from the root the less it matters. So the joint at your fingertip, you can compress a whole lot more because it doesn't matter as much, it's not going to affect anything else. Where a joint at the actual root, the smallest change in motion will affect the entire system in animation and will make it virtually impossible for you to line up animations with each other, so that that particular joint needs to be nearly perfect.

And how do you store rotation? Do you store them in degrees, radians, or normalized? I have seen people store them in degrees. I don't understand why you would ever do that. It's just adding math to the problem. Radians is perfectly fine if you're using off the shelf trigonometric functions- tan, sine whatever. But typically, if you're going to optimize those functions yourself anyway, it's going to be much more effective go with a normalized rotational value. So basically between zero and 1. Makes it a lot easier to do tricks based on the circle. Basically you can just take the fractional value and just deal with that. So normalized rotation is

generally the way to go and normalizing a half precision is probably the even bet for how you would store.

So looking at what we need to fit into an SPU if we're going to running to an end machine. Yeah?

AUDIENCE: You talked a lot about compressing because of the way it's impacting data, what's the key driver of that?

MIKE ACTON: The SPU has very little space.

AUDIENCE: OK, so it's just the amount of space.

MIKE ACTON: Yeah, well OK. There's two factors really, in all honesty. So starting with the SPU. That you have to be able to work through this data on the SPU. But you also have the DMA transfer itself. The SPU can actually calculate really, really fast, right? I mean, that's the whole point. So if you can transfer less data, burn through it a little bit to expand it, it's actually a huge win. And on top of that we have a big, big game and only 256 megs of main ram. And the amount of geometry that people require from a current generation game or next generation game has scaled up way more than the amount of memory we've been given, so we've only been given eight times as much memory as we had in the previous generation. People expect significantly more than eight times as much geometry on the screen and where do we store that?

We have the Blu-Ray, we can't be streaming everything off the disc all the time, which is to another point. You have 40 gigs of data on your disc, but only 256 megs of RAM. So there's this sort of a series of compression, decompression to keep everything-- basically, think of RAM as your L3 cache. So we look at what we want to store on an SPU. Basically, the goal of this is we want to get an entire animation for a particular skeleton on an SPU so that we can transform the skeleton and output the resulting joint data. So let's look at how big that would have to be.

So first we start with the basic nine channels per joint. That's not assuming and again, you'd probably have additional channels, like foot step channels and sound channels and other sort of animation channels to help actually make a game. In this case, we just want to animate the character. So we have rotation times 3, translations times 3, and scales times 3. So the first thing to drop and this will cover, this will reduce your data by 70%, is all the uniform channels. So any data that doesn't actually change across the entire length of the animation. It may not

be zero, but it could be just one thing, one value that doesn't change across length of the animation. So you pull all the uniform channels out. And most things that's going to be scale, for example, most joints don't scale. Although occasionally they do.

And translation, in a human our joints don't translate. However, when you actually animate a character in order to get particular effects, in order to make it look more human you do end up needing to translate joints. So we can reduce, but in order to that we need to build a map, basically, a table of these uniform channels. So now we know this table of uniform channels has to be stored in the SPU along with now the remaining actual animation data. Of course, multiplied by the number of joints. So now we have what is essentially raw animation data. So for the sake of argument, let's say the animation data has been baked out by Maya or whatever at 30 frames a second. We've pulled out the uniform data, so now for the joints that do move we have these curves over time of the entire length of the animation. The problem is if that animation is 10 seconds long, it's now way too big to fit in the SPU by a large margin.

So how do we sort of compress it down so that it actually will fit? Again, just first of all, the easiest thing to do to start with is just do simple curve fitting to get rid of the things that don't need to be there that you can easily calculate out. And again, the closer that you are to the root, the tighter that fits need to be. Conversely, the further away you are from the root, you can loosen up the restrictions a little bit and have a little bit looser fit on the curve and compress a little bit more. So if you're doing a curve fitting with the simple spline, basically you have to store your time values in the places that were calculated. Part of the problem is now you have sort of these individual scalars with time can be randomly spread throughout the entire animation. So any point where there's basically a knot in the curve, there's a time value. And none of these knots are going to line up with each other in any of these animation channels. So in principle, if you wanted to code this you would have to basically say, what is time right now and loop through each of these scalar values, find out where time is, calculate the postition on the curve and then spit out the result.

So one, you still have to have the unlimited length of data and two, you're looping through scalar values on the SPU, which is really actually, horrible. So we want to find a way to solve that problem.

Probably the most trivial solution is just do spline segemnts. You lose some compressibility, but it solves the problem. Basically you split up the spline into say, sections of 16 knots and you just do that. And in order to do that you just need a table, you need to add a table that

says what the range of time are in each of those groups of 16 knots for every channel. So when you're going to transform the animation, first you load this table in, you say, what's my time right now at time, t? You go and say which blocks, which segments of the spline you need to load in for each channel, you load those in. So now you have basically one section of the spline, which is too big probably for the current t, but it covers what t you're actually in. So one block of spline for every single channel.

So the advantage of this, now that the spline is sorted into sections is that rather than having all the spline data stored, sort of linearly, you can now reorder the blocks so that the spline data from different channels is actually tiled next to each other. So that when you actually go to do a load it's much more likely because you know you're going to be requesting all these channel at once and all on the same time, t, you can find a more or less, optimal ordering that will allow more of these group things to be grouped in the same cache or at least the same page. And the advantage of course again, is now the length of animation makes absolutely no difference at all. The disadvantage is its less compressible because you can only basically compress this one section of the curve, but a huge advantage is it solves the scalar loop problem. So now you can take four of these scalar values all with a fixed known number of knots in it and just loop through all of the knots. In principle you could search through and find a minimum number of knots to look through for each one of the scalars, but in practice it's much faster just to loop through all four simultaneously for all 16 knots and just throw away the results that are invalid as you're going through it. That way you can use the SPU instruction set. You can load quadwords, store quadwords, and do everything in the minimum single loop, which you can completely unroll. Does anybody have the time?

PROFESSOR: It's [OBSCURED]

MIKE ACTON: So I'm OK.

PROFESSOR: [OBSCURED]

MIKE ACTON: Yeah? AUDIENCE: In

context do you make like rendering the animation or it seems like there would be a blow to whatever you're doing on the SPUs.

MIKE ACTON: Basically the SPUs are taking this channel animation data and baking it out into-- well, in the easiest case baking it out into a 4 by 4 matrix per joint.

AUDIENCE: So the output time's much bigger than the input time? I mean, you're compressing the input by animation?

MIKE ACTON: No, the output size is significant, but it's much smaller.

PROFESSOR: [OBSCURED]

AUDIENCE: So the animation data it's [OBSCURED]

[INTERPOSING VOICES]

MIKE ACTON: No. I was just outputting the joint information.

PROFESSOR: [OBSCURED]

MIKE ACTON: Independently we have this skimming problem. Independently there's a rendering problem. This is just baking animation. This is purely animation channel problem.

PROFESSOR: [OBSCURED]

MIKE ACTON: OK, I'm just going to skip through this because this could take a long time to talk about. Basically, what I wanted to say here was let's take the next step with animation, let's add some dynamic support. The easiest thing to do is just create a second uniform data table that you then blend with the first one and that one. In principle, is basically all of the channels and then now a game play programmer can go and individually set any of those. So they can tweak the head or tweak the elbow or whatever. And that's definitely compressible because it's very unlikely thet're going to be moving all the joints at once. You can create a secondary map that says, this is the number of joints that are dynamic, this is how they map to the uniform values. But then once you add any kind of dynamic support, you have now complicated the problem significantly. Because now in reality, what you need are constraints. You need to be able to have a limit to how high the head can move because what's going to happen is although you could just say the head could only move so much, if that movement is algorithmic, so let's say follow a character or whatever-- it is going to go outside of reasonable constraints really quickly. So it's much cleaner and simpler to support that on the engine side, so basically define constraints for the joints and then let the high-level code point wherever they want.

AUDIENCE: [OBSCURED]

MIKE ACTON: Yeah. Yeah, you can have max change over time so it only can move so fast. The max range of motion, the max acceleration is actually a much harder problem because it implies that you need to store the change over time, which we're not actually storing. Which would probably blow our memory on the SPU. So as far as impacting animation, I would immediately throw out max acceleration if an animator were to come to me and say, this is a feature that I wanted. I would say, it's unlikely because it's unlikely we can fit it on the SPU. Whereas, on the PC, it might be a different story. And blending information, how you blend these things together. What's that?

AUDIENCE: [OBSCURED]

MIKE ACTON: OK. So as far as mixing, there's plenty of additional problems in mixing animation.

Phase matching, so for example, you have a running and a walk. Basically all that means is if you were going to blend from a run to a walk you kind of want to blend in basically the essentially same leg position. Because if you just blend from the middle of an animation to the beginning of the animation, it's unlikely the legs are going to match and for the transition time you're going to see the scissoring of the legs. Which you see that in plenty of games, but especially in next generation, especially as characters look more complicated they are expected to act more complicated.

Transitions handling either programmatic transitions between animations, so we have an animation that's standing and animation that's crouching and with constraints, move them down; or artist driven animated transitions and/or both.

Translation matching is actually an interesting problem. So you have an animation that's running and you have an animation that's walking. They both translate obviously, at different speeds, nonlinearly and you want to slowly run down into a walk, but you have to match these sort of nonlinear translations as his feet are stepping onto the ground. Turns out to be a really difficult problem to get perfectly right, especially if you have eye key on the feet where he's walking on the ground or maybe walking uphill or downhill and the translation is being affected by the world. In a lot of cases you'll see people pretty much just ignore this problem. But it is something to consider going forward and this is something that we would consider how to solve, regardless of whether or not we could get it in.

As far as actually rendering the geometry goes, you now have your sort of matrices of joints and you have-- let's say you want to send those to the GPU along with the geometry to skin

and render. Now the question is, do you single or double buffer those joints? Because right now basically, the GPU can be reading these joints in parallel to when you're actually outputting them. So the traditional approach or the easiest approach is just to double buffer the joints. So just output into a different buffer that the R6 is reading from. It's one frame or half a frame behind, doesn't much matter. But it also doubles now the space of your joints.

One advantage that games have is that a frame is a well defined element in the games. We know what needs to happen across the course of a frame. So these characters need to be rendered, the collisions of this background needs to happen, physics need to happen here. So you can within a frame, set it up so that the update from the SPUs and the read from the GPU can never overlap. Even without any kind of synchronization or lock, it can be a well known fact that it's impossible for these two things because there's actually something in the middle happening that has its own synchronization primitive. That will allow you to do single buffering of the data. But it does require more organization. Especially if you're doing it on more than just one case. So you have all these things that you want single buffered, so you need to organize them within the frames so they're never updating and reading at the same time.

So I'll make this the last point I'll make. Optimization, one of the things that you'll hear, save optimization till the end. My point here being is if you save optimization till the end, you don't know how to do it because you haven't actually practiced it. If you haven't practiced it you don't know what to do. So it will take much longer. You should always be optimizing in order to understand, when it actually counts, what to do. And the fact that real optimization does impact the design all the way up. Optimization of the hardware impacts how an engine is designed to be fast does impact the data, it impacts how game play needs to be written, high-level code needs to be called. So if you save optimization till last, what you're doing is completely limiting what you can optimize. And the idea that it's the root of all evil certainly didn't come from a game developer, I have to say. Anyway, that's it. I hope that was helpful.

PROFESSOR: Any questions? I think it's very interesting because there is a lot of things you learn at MIT. Forget everything you learned so I think there's a very interesting perspective in there and for some of us it's kind of hard to even digest a little bit, but Question? AUDIENCE: Call of Duty 3 came out on the Xbox and on the PS3, is Call of Duty 3 on the PS3 just running on the GPU then or is it--

MIKE ACTON: No, it's very likely using the SPUs. I mean, I don't know. I haven't looked at the source code, but I suspect that it's using the SPUs. How efficiently it's using them is an entirely different

question. But it's easy to take the most trivial things right, say you do hot spot analysis on your sequential code and say, OK, well I can grab this section of thing and put it on the SPU right and just the heaviest hitters and put them on the SPU. That's pretty easy to do. It's taking it to the next level though, and to really have sort of the next gen of game-- now there's nowhere to go from there. There's nowhere to go from that analysis, you've already sort of hit the limit of what you can do with that. It has to be redesigned. So I don't know what they're doing honestly and certainly I'm being recorded so-- yeah?

AUDIENCE: You guys have shipped a game, on the PS3?

MIKE ACTON: Yeah, it was on action list.

AUDIENCE: OK, so that was more like the [OBSCURED] games and whatever You seem to talk a lot about all these things you've had to redo. What else is there-- games look better as a console was built on, what else is there that you guys plan on changing as far as working with the cell processor, or do you think you've got it all ready?

MIKE ACTON: Oh, no. There's plenty of work. There's plenty more to be optimized. It's down to cost in scheduling those things. I mean, we have a team of people who now really understand the platform and whereas a lot of what went into previous titles was mixed with learning curve. So there's definitely a potential for going back and improving things and making things better. That's what a cycle of game development is all about. I mean, games at the end of the lifetime of PlayStation 3 will look significantly better than release titles. That's the way it always is.

AUDIENCE: The head of Sony computer and gaming said that PS3 pretty soon would be customizable. You're be able to get different amounts of RAM and whatnot.

MIKE ACTON: Well, I think in that case he was talking specifically about a PS3 based, like Tivo kind of weird media thing, which has nothing to do with us.

AUDIENCE: [OBSCURED]

MIKE ACTON: We're not stuck. That's what we have. I mean, I don't see it as stuck. I would much rather have the-- I mean, that's what console development is about, really. We have a machine, we have a set of limitations of it and we can push that machine over the lifetime of the platform. If it changes out from under us, it becomes PC development.

AUDIENCE: Are you allowed to use the seven SPUs or are you--

[OBSCURED]

PROFESSOR: [OBSCURED]

MIKE ACTON: I don't know how much I can answer this just from NDA point-of-view. But let's say hypothetically, there magically became more SPUs on the PS3, right? Probably nothing would happen. The game has to be optimized for the minimum case, so nothing would change. Anything else? Yeah?

AUDIENCE: So what's the development life cycle like for the engine part of the game. And I don't assume you start by prototyping in higher-level mechanisms. Then you'll completely miss the design for performance aspects of it. How do you build up from empty [OBSCURED]

MIKE ACTON: No, you don't start with an empty. That's the perspective difference. You don't start with code, code's not important. Start with the data. You sit down with an artist and they say, what do you want to do? And then you look at the data. What does that data look like? What does this animation data look like?

PROFESSOR: Data size matters. [OBSCURED]

MIKE ACTON: Right. We have to figure out how to make it smaller. But it all starts with the data. It all starts with that concept of what do we want to see on the screen? What do we even want to hear on the speakers? What kind of effects do we want. And actually look at that from the perspective of the content creator and what they're generating and what we can do with that data. Because game development is just this black box between the artists and the screen. We're providing a transformation engine that takes the vision of the designers and the artists and just transforming it and spitting it on to screen. So where you really need to start is with the source of the data.

AUDIENCE: You've been doing game developoment for 11 years now is what you said. Have you had a favorite platform and a nightmare platform?

MIKE ACTON: I've been pretty much with the PlayStation platform since there was a PlayStation platform and I don't know, it's hard to get perspective because you're getting it and you always really love plarform you're working on. So it's hard. I mean, it's hard to get perspective. In the program where I am today is not the same program where I was 10 years ago. Personally, right now my favorite plarform is PS3.

PROFESSOR: So when put it in perspective, there are already some platforms that on the first time round, [COUGHING], it's like cost of development. So one platform that as time goes we have [OBSCURED]

MIKE ACTON: Well, like with the PS3, some of the things that I like about the PS3, which is sort of a different question are the fact that the cell is much more public than any other platform has ever been. With IBM's documentation, with Toshiba's support and Sony support, I've never had a platform where I can get up on a website and actually talk about it outside of NDA. And that for me is an amazing change. Where I can go and talk to other people-- exactly like this group here- that have used the same exact platform I've used. That's never been able to happen before. Even on PS2, for quite a long part of the lifespan, even though there was a Linux eventually on the PS2, virtually everything was covered by NDA because there was no independent release of information. So that's one of the great things about PS3 is the public availability of cell.

PROFESSOR: Thank you very much for coming all the way from California and giving us some insight.