BRIAN SWEATT: So we're the real time ray tracing group. And as we like to call ourselves, Blue Steel. So one thing you might be asking yourselves and wondering from us is, where did we get the name Blue Steel? What does it have to do with this thing we call ray tracing?

And this morning when I was sitting up and finishing up the project, I was asking myself the same thing. We're doing ray tracing. Where did we come up with this name? So I'm going to give you a little background on what Blue Steel is.

The name came from the first day we got access to our PS3, we went to the student center like a bunch of excited kids on Christmas. And what did we do? We created a mailing list for our group. Didn't even mess with the PS3. Just created the mailing list and sat there for about an hour and a half, thinking of a name.

And somehow, we decided on Zoolander's famous face, Blue Steel. And it didn't have particular significance to us before, but as you'll see, it has a lot of historical value.

Blue Steel is also an alloy, a ferrous alloy. As you could see, we have blue tempered spring steel that many companies, industries, use. And Blue Steel is also a warhead made in the Cold War. But these have nothing to do with our project.

To us, and for the purpose of this project, Blue Steel is a distributed real time ray tracer. And Natalia is going to give you a little background on what ray tracing really is.

NATALIA Thank you. So ray tracing, there are two terms that you would generally hear, ray

CHERNENKO: tracing and ray casting. Basically, the idea is the camera is like the eye. And from the camera, rays are cast for each pixel in a picture.

And these rays, they intersect with all the virtual objects in the scene. When an object is hit, the color of it is computed. The closest object that is intersected is the object that shows up on the screen.

BRIAN SWEATT: We should credit these slides. They're Fredo Durand's from 6A37.

NATALIA CHERNENKO: And ray tracing is recursive. It accounts for shadows, positions of other objects in the scene. There is reflection and refraction.

For example, if this ray were to hit that object and then reflect it, then another ray is cast from that object in the reflecting direction. And this can go on for an indefinite number of reflections. These are normally specified in the program.

Here is an example of the recursion. So the three planes are reflective. They're like mirrors, and there are three spheres running between them. With no recursion, you can see there is no reflection at all. With one recursion, you can see the rays are reflecting just off the side mirrors.

At the end with two recursions, there is also reflection at the bottom. So it is a reflection of one of the reflections. And this is very computationally intensive because for every pixel, you're casting a ray. And for every object in the scene, you're intersecting that ray with the object and seeing if it's hit.

And then if it is hit and it is reflective or refractive-- so transparent-- then another ray is cast. This scene here is a resolution 1024 by 1024 and recursion depth is 100. And the rendering time in our program was 2.6 seconds. The rendering time for 20 depth recursion was 600 milliseconds, and for 10 was 300 milliseconds.

PROFESSOR: [INAUDIBLE] SPU [INAUDIBLE]. What would be the [INAUDIBLE].

R. J. RYAN: Many of us have taken 6A37, and in that class, we're all about ray tracers. And a typical scene like the one that is on the PowerPoint right now could take up to a minute or more. Often, if you were to do 100 bounces, I think mine would probably

take over 10 minutes.

PROFESSOR: [INAUDIBLE] you just to [INAUDIBLE] SPU or [? one SPU. ?]

NATALIA CHERNENKO: We did try different SPUs. This picture, we didn't try with different SPUs, but we have tried other scenes with various numbers. And we did get exact linear speed up. So if there were twice as many SPUs, it would be twice as fast.

R. J. RYAN: OK. So I'm going to talk to you about optimizations that we did. Natalia's told you about ray tracing, and that's only half of our project. The part that we have to focus on is optimizing. And meeting that goal of being real time.

And for us, real time, you have to ask, what is real time? If we go off of like, the American video standard NTSC, they define it to be around 27, 30 frames a second. So that was pretty much the goal that we set when we set out to do this.

And so we took the code and we think, OK, we can split this. This task is easily parallelizable. But we're going to have to do a lot more than that if we want to get better rates. So we took advantage of the hardware that we were on. And one thing that on the ray tracing end, we had to go crazy with, was SIMD.

Pretty much, the entire ray tracer never deals with a scalar value. We use SIMD for everything. And in the core of it, we do branch avoidance. So we really try not to take branches, because branch misses are very costly on the cell. So I believe there's something like three if statements spread across 4,000 lines of code for the ray tracer.

We really took SIMD to hear in this. Using SIMD, you can process arrays of structures. So we took--

NATALIA CHERNENKO: [INAUDIBLE]

R. J. RYAN: We would trace ray packets of four rays at a time, and from those ray packets, we

would generate hit packets. And then similarly, as you go down the line, we would produce an RGB packet, which is four color values tied into one. So at every step, we would use SIMD to our advantage. Now Brian's going to talk to you about pipe lining.

BRIAN SWEATT: So of particular importance on the cell is not only what instructions you execute, but the order that you execute them in, because there is no reorder buffer on the cell. There is no out of order execution. And despite the fact that compiler technology is very good, hand optimization and pipe lining instructions is still always better.

For instance, with our triangle intersection routine, we gained on the order of hundreds of milliseconds of render time. We shaved hundreds of milliseconds of render time off of our render time for some scenes you will see later, just by reordering our calculations in the intersection code. So it's very important, especially on the SPEs.

And one other optimization we made was the fact that we coded basically everything on the SPEs. Anything running on the SPEs is coded in C. Not in C++. And we're not using inheritance when we do use C++ because virtual functions have more overhead in respect to memory, the V table, and also looking up the function pointers from the V table. So these are just a couple of the optimizations we made. We'll talk about more later.

AUDIENCE: Just a quick question. So for the pipeline, you actually reordered instruction statements? The compiler wasn't very helpful in that respect?

BRIAN SWEATT: Yeah. So on that scene, 1024 by 1024 with the spheres, we were getting, Natalia mentioned with a depth of recursion 10, we were getting 300 milliseconds. That's with optimized.

Without the optimizations, we were getting closer to 500 with a depth of recursion of 10. And we gained that extra 200 milliseconds just by reordering instructions in the intersection with optimization level 03 in XLC. So hand optimization, like Mike Acton said in class, is still very important. And I'm going to hand you off to Mikey now to

discuss the regular grid.

MICHAEL D'AMBROSIO: So one of the problems, I guess, with ray tracing is that intersections are very expensive. And when you have scenes with n rays and m primitives, well, you have to generate n times m intersections. So one common acceleration structure is grid.

And the concept is very simple, as the picture depicts. You just take your screen, split it up into a number of evenly size box holes, place your primitives in them, and then you march your ray through. And you only intersect the ray with primitives that are around the area.

And the good thing about a grid is that it's very easy to construct, as opposed to, say, a k-d tree or an octree. Those take on the order of n log n, where n is number of primitives. Regular grid is order n, where n is the number of primitives. And the reason why we did this is because with an interactive scene where things are moving around, you need to rebuild the grid every frame or whenever something moves.

So we favored this choice of acceleration structure because it was quick to build and easy to traverse. And now Fish is going to tell you about the fun part of the project.

SCOTT FISHER: I was in charge of building the actual physics engine. Our original plan was to the 6170 well-known gizmo ball in 3D and ray traced. And I guess if we want to show that? I have it running on this laptop. We didn't quite get it ray traced, but I'm just going to show you a little bit of what I did as a physics engine. Maybe not.

R. J. RYAN: We'll get the demo working and then I think we can use it later.

SCOTT FISHER: Anyway, it was basically a 2 and 1/2 D version of the example that was just up on the screen. Flippers work, absorbers work, and due to complications, we never did get it fully ray traced. And I guess we'll go ahead and give you the real ray trace demo, since we do have a couple of those.

PROFESSOR: So their actual demo will be on a little LCD screen here. Hopefully everybody can

see the ray trace. Unfortunately, PS3 needs a specific kind of digital connection to display things in the right resolution when you're writing through Frame Buffer. So we can't project it onto the big screen.

PROFESSOR: [INAUDIBLE]

PROFESSOR: Can I do it from here?

PROFESSOR: Perfect.

BRIAN SWEATT: Now you got to tell [INAUDIBLE]. Oh, no. Wrong way.

R. J. RYAN: Really?

BRIAN SWEATT: By the way, he's still adjusting it.

R. J. RYAN: Whoops.

BRIAN SWEATT: Uh oh. Oh, no.

R. J. RYAN: OK. We've prepared a few demos for you. And these are really just going to show off the ray tracing part of it and the rates that we got after we applied our speed ups. Thank you.

Here we see both the camera is moving and we implemented full scene functional textures, so you see the checkerboard. It's generated every frame. We used no texture maps. This is completely generated functionally.

Bump mapping, as you can see in the large orange sphere, is also done on the fly, as well as blending between two textures using Perlin noise is done on the fly also. Showcased in the center sphere right here, you can see both reflection and refraction. The sphere is transparent. It acts as a glass sphere.

So you can see the reflection of the orange in it, but also the warping of the checkerboard as it goes through. This is one of the advantages of-- well, spheres are one of the advantages of ray tracing, because you can use geometrical equations to define your geometry versus in using rasterization methods, you have

to actually define the geometry in some concrete numerical way.

Going to go to the next demo. This should showcase some of the lights that we implemented.

AUDIENCE: So how many frames per second is this?

R. J. RYAN: This should be 15 frames a second. And let me reiterate. The refraction step requires at least two bounces, so two steps of recursion. If we were to turn that off, we would be achieving, I would estimate, 27 frames a second, 28. But then it doesn't look as pretty, so, you know, what's the point.

You can see-- this will be showcased later-- but you can see the spotlight. It's kind of moving. It's waving back and forth, illuminating parts of the scene. There's also a point light at some distance around here that's illuminating everything. But it's basically the same set of primitives.

If you are closer, you could probably notice that the checkerboard is fully reflective. This far wall, you can view the entire scene in it. Going to go to the next demo.

This is with the lights off. So you can get a better sense of what the spotlight is actually doing. If you notice on this orange sphere, we implemented two different forms of BRDFs. One was generic foam shading. And then the other was Cook-Torrance, which helps particularly with metals and other materials like that. It helps have very bright specular highlight.

And you'll notice as the camera's panning back and forth, the speed ups that we get from-- we used little tricks to glean speed here and there. Like when it's so dark or it's very far away, we stop recursing.

BRIAN SWEATT: We're over, so--

R. J. RYAN: We are?

BRIAN SWEATT: Yeah. So if there's any--

R. J. RYAN: Let's see.

SCOTT FISHER: Do the SPUs from one to six.

R. J. RYAN: I can demonstrate-- we actually enabled it so that you can turn it on either one through six SPEs using command line arguments. Let's go back to the first demo. If I say just use one SPE, we can watch it slowly trod along.

And if I were SHH-ed in, it would be giving me timing information so I could tell you exactly. It's really unfortunate that it's not, because it's very linear in its speed up. Right now it says 500 milliseconds.

But if I turn it on two, it drops literally in half to 250. You can get a sense of that, that it's moving like tick, tick, tick, tick versus tick, tick, tick. And then three, it's a similar speedup. And then I'll jump to five. It's getting closer. And then when we turn on the sixth, this should be about 12, 15 frames a second.

AUDIENCE: So you estimate if you had a dozen SPUs, you could go--

R. J. RYAN: Actually, it's funny you mentioned that. I think it was the Julia re-tracer or the IRT.

BRIAN SWEATT: Both of them.

R. J. RYAN: Both of them use two cell blades, which is a full eight?

MICHAEL D'AMBROSIO: It's a full eight times--

R. J. RYAN: Eight cores times two. So they use 16 SPUs. The PlayStation 3 do have certain factors. We're only limited to six.

AUDIENCE: The way you're dividing up the work, each SPU taking every sixth raster line, that sounds like-- could you improve on the performance by allocating the rays--

R. J. RYAN: The issue there is one of granularity of our distribution. And also communication time on the bus. We've designed it such that we can fine tune it in any way we want.

AUDIENCE: But it seems to me if you had a localized bundle of rays on a single SPU, you might be able to do some optimizations, like treat them as fat rays. Nearby rays are likely to intersect the same triangles.

MICHAEL D'AMBROSIO: Choosing the frosting variety.

R. J. RYAN: The entire material system works very much like that. It actually saves on computations based on which rays in a packet of four have collided with the same element. And if it does, then it takes full advantage of SIMD to quickly chug through those rays that are the same and then never call that particular shader again, because it's already done the work for them.

MICHAEL D'AMBROSIO: Another issue, I think, is--

PROFESSOR: Sorry, we should try to wrap so we can get through the others. Any more very short questions? Did you guys have any last concluding statements?

NATALIA CHERNENKO: Maybe we go through this like really fast?

[INTERPOSING VOICES]

BRIAN SWEATT: Point is we had two pieces.

PROFESSOR: Thank the speaker.

R. J. RYAN: Thank you.

[APPLAUSE]