EPISODE 1041

[INTRODUCTION]

[00:00:00] JM: Building a game is not easy. The development team needs to figure out a unique design and game play mechanics that will attract players. There's a great deal of creative work that goes into making a game successful, and these games are often built with low budgets by people who are driven by the art and passion of game creation. A game engine is a system used to build and run games. Game engines let the programmer work at a high-level of abstraction by providing interfaces for graphics, physics and scripting. Popular game engines include Unreal Engine and Unity, both of which require a license that reduces the amount of money received by the game developer.

Godot is an open source and free to use game engine. The project was started by Juan Linietsky, who joins the show to discuss his motivation for making Godot. We have done some other great shows on gaming in the past. If you're looking for those, you can find them on softwaredaily.com, and if you're interested in writing about game development, we have a new writing feature that you can check out by going to softwaredaily.com/write. If you are knowledge or curious about game development, we would love to see an article from you. So go to softwaredaily.com/write if you're interested.

[SPONSOR MESSAGE]

[00:01:27] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite

© 2020 Software Engineering Daily

interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[INTERVIEW]

[00:03:45] JM: Juan Linietsky, welcome to Software Engineering Daily.

[00:03:47] JL: It's a pleasure.

[00:03:48] JM: We're going to be talking about Godot, which is a game engine. But speaking more broadly, what is the purpose of a game engine?

[00:03:59] JL: That's a really good question. Originally, I have to admit that it wasn't really a game engine. [inaudible 00:04:05] more context on the origins of the technology. I've been doing technology for people to make games for probably over 20 years at this point. My main experience has been working with companies and working on creating tools for people to make games since then, because originally when I started working with video games, there wasn't any kind of tool or game engine you could download and use like you can now. There wasn't like Unity. I think Unreal costs like half a million dollars. It wasn't very easy to get a hold of technology for making games.

Initially, I think I'm mainly working on tools for just companies and myself for making games. I made games with a lot of companies, on licensed technology to a lot of companies for a long time. It only began taking shape of kind of like the game engine, for modern game engine maybe in 2008, 2009 maybe because we arrived to it from previous iterations that were not so usable. It was mostly about improving usability. When you look at the engine, what you see is something that had its own evolution and not so much copying something else. Most people who uses it needs some a lot meantime because it's not following most of the conventions of the other game engines because it had like a separate evolution.

Right now it's like a game – A game engine is – For me, it was something that I used to make tools for people to make games. Maybe when I was technical director companies then when I was technical consultant, it was always about that. But nowadays there are sort of stuff that people make, they make like interactive stuff, like interactive graphics that are used for movies, for medicine, for simulation. Game engines are not really only about games now, I guess, but I guess mostly they are. I'm not exactly sure what you call it overall, more like they are still called game engines, but they can be used to make other stuff. There're people making applications with Godot because you have very nice UI framework to work. But yeah, that's kind of my view.

[00:06:16] JM: Going back, you're the head of development for Godot, and Godot is a game engine. Why did you start Godot?

[00:06:26] JL: Well, as I was mentioning before, there wasn't any kind of game engine we could license, but then when I started working on games like 2000s, so that's kind of why. It's also why some years ago when general purpose game engines came out, like Unity, Unreal, and they became more accessible both interface and usability-wise and price-wise, I guess, because now we can kind of use them for free. This is why back then I put what I had of technology as open source, because I wasn't really interested in just making a product. This was all discussed together with Ariel, Ariel Manzur, who actually is the one who created everything from scratch with me. We just it put in source for people to use, and it was kind of like a hobby for me for a while because I continued – I just moved on to more like business consulting for companies because I had a different experience in the industry and I realized that business consulting paid better [inaudible 00:07:27] paying a CTO or technical lead. Godot was kind of like a hobby, and I

just work on it a few hours a day just like improving and working off feedback from people. From there, well, it started growing and becoming something big with a lot of user-base right now. That's kind of surprising. I didn't expect it.

[00:07:50] JM: Now, a game engine is a gigantic project. Taking on such a big project is a real big undertaking, and I don't know if there's a clear roadmap to making it something that would be profitable. It sounds like it was just started – You started kind of as a hobby or just because you wanted to make it possible to build games without licensing one of these engines. But in any case, it's this big undertaking. What was going on in your life when you first started the project?

[00:08:25] JL: I was working as a technical consultant. So all the technology I made was for licensing to other companies. It wasn't so much for probably anything like that initially. It became a hobby later when we open sourced it. I was pretty much just working for clients and giving them technology usually for working for consoles or desktop or mobile because it wasn't so common to game engines back then. At the beginning, it was more like that.

[00:08:54] JM: Okay. You did build this as a commercial effort originally.

[00:08:59] JL: Yes. Well, not something like a product that you sell to anybody who wants to buy it. More like a service rather than a product.

[00:09:06] JM: Can you explain that in more detail?

[00:09:08] JL: Yeah. Basically, if you make something like Unity, and you sell it. Back then, Unity was commercial too. When you want to sell it to your clients, you have to spend most of your time doing support. You sell it for some money, and they must be able to more or less have something usable and polished. They require the least amount of support as possible, because you just can't give support to pretty much everybody on the planet. While Godot was used for services, it didn't really need to be something super polished. We had to do whatever we had to do, and if it has a bug, or clients would just tell us about it, and we would fix the bug and make the features they need for the game. It was more like a platform. We would give them stuff they

need to make their games. That's something like a polished product, I guess. This is why it was a more service than a product.

[00:10:00] JM: Got it. Now, I'd like to talk more about what actually goes into a game engine. So maybe we could start with just the examples of Unity and Unreal Engine, which are two popular game engines that existed when you were starting Godot. Can you give me a brief history of these engines, like what they do? What the game engine ecosystem is like in those respect?

[00:10:29] JL: I think to be honest, game engines have to be some of the most complex pieces of technology you will find around broadly behind like operating systems on these kind of things. Game engines can do a lot of different things. They have to handle like rendering, and handling rendering is super complex. They have to handle like physics. They have to handle audio. They have to handle scripting. They have to handle a lot of IO, like loading resources on background efficiently. They have to handle just so many things, like image loading. They are really, really big and complex pieces of technology that you require a huge amount of expertise to development. I guess this is why there are not so many game engines around. If you look at like big game engines that handle pretty much everything, you will only find like Unity, Unreal and Godot because it's something so complex and requires such an experience in so many amount of fields that you won't see many, many around. But yeah, it's a lot of different disciplines and theory that mix together to put something in just one place.

[00:11:40] JM: I'd like to just walk through a very basic toy example and then use that to talk about the game engine from an engineering point of view. Let's say I'm using Godot and I want to build a very simple game. Let's say I want to make a game where there's just a 3D character that walks around in a small room. How do I make that game in Godot?

[00:12:04] JL: Well, actually, 3D is not so simple. Most users begin with 2D because it's just easier to picture in your head and the match is much simpler because it's 2D. But if you want to do 3D, it's not so much different.

In Godot you have something called scenes and nodes and you pretty much do everything with it. The closest I can think of to the way Godot works is something like a UI framework, like JDK

or QT. I'm pretty sure you're mostly familiar with that. They have like objects. In Godo, it's similar. We call them nodes and you arrange them in a tree. For example, your character is going to be a node, usually a kinematic controller, which is something that lets you like move around, splicing like a direction in a velocity and it's going to just [inaudible 00:12:53] and collide with stuff. You would probably need to put children to that kinematic body, like the collision shapes, which are like the define the shape of a character, or is a child or like kinematic body node. Collision shape is also a node.

You probably want to have a model of like your character. So you need a mesh, a mesh instance. Which instances are mesh and puts it back into a tree. When you move one of those nodes, the children nodes move with it. You will probably need an animation player to play animations to move the character. You would probably know also the skeleton. These are all just nodes you put in your scene.

Then you probably want to put a script and you can assign that script to any node, which is almost like inheriting it in programing terms. You create this scene with a character which has like as a base node the kinematic body and node, and all the nodes for like the meshes and the audio and the animation and everything and you just save it to a file. Then you probably want to create the game scene. Then you probably want to import like – I don't know, you instance a castle in Blender and you import it in Godot adding collision shapes like – You make special objects in Blender that will be converted to collision when you import it to Godot.

Then you just put – You use something called scene instancing in Godot, which is like the core of the engine. Then you will instance like the character scene. Before you will instance, you will instantiate it into the world that you have made with Blender that you imported also.

So, in the end when you make something with Godot, you're just creating like individual scenes with nodes that each node has a different function. Usually, nodes have a unique function and you put them [inaudible 00:14:37] tree. Then you compose your game scene using different scenes that each contains node inside. You just put everything together. It's very similar to when you make like UIs. You can probably create in QT a window widget. That window widget, we have patterns and texts and everything. Then you can instantiate that window widget into

another like apparent window or apparent widget or something like that. It's a very similar way of working very object-oriented, but mostly for making games. That's kind of how it works.

[SPONSOR MESSAGE]

[00:15:16] JM: DNS allows users to navigate to your web endpoints, and whether they're hitting your app from their mobile phone or accessing your website from their browser, DNS is critical infrastructure for any piece of software. F5 Cloud Services builds fast, reliable load balancing and DNS services. For more than 20 years, F5 has been building load balancing infrastructure. Today, F5 cloud services provides global DNS infrastructure for lighting fast access around the world.

If you're looking for a scalable, high-quality DNS provider, visit f5.com/sedaily and get a free trial of F2 Cloud Services. Geolocation-based routing, health checking, DDoS protection and the stability and reliability of F5 Networks. Go to f5.com/sedaily for a free trial of F5 Cloud Services. Fast, scalable DNS and load balancing infrastructure. That's f5.com/sedaily.

[INTERVIEW CONTINUED]

[00:16:25] JM: You mentioned this concept node. Can you explain how nodes are used to make a game? What is the concept of a node?

[00:16:34] JL: A node is something that has some sort of behavior. The behavior is just doing something. Not just a behavior, like it has to have like one function. It has to do something, but is one function. For example, if you think of inheritance, like in object-oriented, you have too much, and like Godot has a base class, which is the object base class. From the object base class, you have inherited the node class. The node class is something that you put in your scene. For example, you can inherit like the 3D node, the base 3D node from the node class. From the 3D node, you inherit like 3D stuff. For example, a mesh, which is some 3D model. You can instantiate like [inaudible 00:17:13]. You can inherit a light [inaudible 00:17:17] that you just put in the scene and gives you like light thing. You can put like a sun, a spotlight, an omnilight. You can instantiate that and also pull like, for example, a physics body. It can be a static collision or a rigid body that has physics. As a children of their bodies, you can put shapes that

will give it collision. It's just something that has one function and the relation between parent and child usually means something according to the context. Like for 3D nodes, the relationship the parent and a child node. They will like take over the position. Like you move the part and the child moves with it.

For physics, I only can have like shape children. That means that they will just give you like a volume to the body. A node is something that provides a function and you lay it down in a tree. Like it can have any number of children, and sometimes [inaudible 00:18:08] of the child, it may have a special behavior regarding to the child. That does kind of what the node is.

[00:18:16] JM: These nodes are used to create scenes. What goes into a scene?

[00:18:21] JL: A scene basically has one node, which is a root, and then you can put any other node as a child feat. The minimum requirement of a scene is that there is a single root. This allows you to instantiate it like somewhere else. So you just move it as a single entity. Yeah, basically you can think of it as an object-oriented that it's something that is part of the scene.

Well, the scene is something that has [inaudible 00:18:44] node, and the thing is like you can save it and load it and you can instantiate this in another scene. You can kind of divide and conquer to make your game. You just think of it in pieces. For example, if you're making a pong game, like the two pallets of pong, each can be a scene. The ball is probably a scene [inaudible 00:19:03] some logic, the sprite, the collision. It's kind of like when you imagine all the components or everything that makes up a game world, you just divide it [inaudible 00:19:12] to be able to split better and reuse better.

[00:19:18] JM: If we revisit this example of just a 3D character walking around inside of a room, can you put that in terms of nodes and a scene?

[00:19:30] JL: Yes, of course. Imagine for example I will tell you a typical game scene is composed in Godot. You probably have like the 3D model, which is when you work on the 3D application, like Blender or Maya or 3D Studio Max, it actually works also in a scene. It's for a scene in the terms of the 3D content creation program. You're just going to make your meshes and just lay them around and put anything.

When you import it Godot, it also is the same scene that you have edited like in Maya or in Blender. To add something locally in Godot, you will usually inherit that scene or instantiate it into a new scene. You have the base scene, which comes very much and you export like a VX file or a GLTF file which is like 3D data. That is just a scene for Godot, but you can't really modify it, because every time you save it over because of the change, it's going to overwrite it. You can't really modify that one, because you're going to editing it in the 3D program.

What you do is in Godot is you can just like – Imagine you create your character. You have a new scene is make character scene. It has like a root node, which is again a kinematic volume, which is used to control characters. Then I will instantiate the 3D scene with a character that I exported from Blender. Yeah, we have the scene and the instantiated scene. You can do local modifications to anything you instantiate in a scene in Godot. You can like, for example, change the shaders of the object or change the materials or anything else. Imagine your character scene. You save it again. It's a scene that container another one. You use this for the enemy. You want to make an enemy troll, for example, and you will create your enemy troll scene and you're going to save it to a file.

When you make a level, you probably want to create a new level scene. You probably instantiate, like for example, a few houses like the ground. Everything that you can think of something in the game can be a scene. Usually you save it as a scene to organize better. Then you can instantiate the enemies and place them in the map visually. The enemies, everyone, is a scene. But you'd already used the one. You just instantiate it like many times. Then you instantiate the player, maybe in the player scene, because it's the one you're controlling. You want to also add a camera node. The camera is just a node that you can position and it's going to show you in the screen what the camera is looking at as an example. But maybe you want to do something else, like in a real production game, usually what you do is the level design is saved in a scene. Then maybe you have somebody who knows how to do game lighting, and this person is going to put all the lights – You're going to inherit the level scene and going to add like light nodes to it just to make proper lighting and then going to save it.

Again, it's something like every part of the game you can think of, like an enemy, a light, a level, even the UI. Everything is just a scene in Godot and you just divide them. You could have like

one single scene with everything, but it's not very productive. Because it's just much easier to just subdivide and reuse stuff. That's kind of what the scene is. Just a part of the game you think of it and then you save it. Again, it's like divide and conquer to reach the goal faster.

[00:22:34] JM: A physics engine is key to building a game engine. If I want to add physics to the entities that are moving around within my game, what do I need to do?

[00:22:49] JL: In the specific case of Godot, you create like again a kinematic body is usually what you do and then you add shapes to it and then you can inherit the kinematic body using a script to control it. From that script, you can just for example, "Look, I am going to check if the key is pressed, then I'm going to move forward," and I tell the kinematic body, "Okay. Kinematic body, okay, move forward, and if you hit something, okay, slide it." For example, if you're falling down and you fall into a floor but the floor has like a slope, then you want it to slide down. For example, just like this kind of stuff you do from script. If you want to just like, for example, [inaudible 00:23:27] and you want to just [inaudible 00:23:29] realistic physics is different. You use a rigid body using the collation shape to it. Then when you touch it, it's going to just bounce off you because it has physics.

Internally, we are using the bullet physics engine for 3D. We also have a custom made physics engine that is pretty old. I've been thinking about making it more modern at some point. For 2D, we are using a custom 2D physics engine. Yeah, you have to make – Usually, physics engines behave more like game engines one less than the way physics engine work to be more user-friendly in general. Basically, we just wrap in some way the physics engine from [inaudible 00:24:09] so you can control them.

[00:24:12] JM: Tell me about programming a physics engine in Godot. Like you had to program it to actually make the game engine be able to run physics. What does that involve?

[00:24:25] JL: I would try to abstract that from users, because for me, physics is extremely complex. The code in a physics engine is super, super complex. Most of the code is like feedback-based. It's not something like when you do rendering, for example, you start with some information and you finish with a frame render. With physics, it's very more complex, because physics needs to like feedback from the previous frame. You do all the physics

calculation and you'll evolve the objects with all their forces and all their collisions and all the contact points and then you need to go back the next frame and continue processing that. Giving users the ability to do that such low level is generally unnecessary [inaudible 00:25:04] complex. We just wrap it in an easier two-way use with. The most common use case is like – This is a rigid body. It's going to bounce and roll and everything like any physics object. If you want to make a character, a character – Usually the characters don't use realistic physics, because it's very frustrating because it's just going to bounce or do stuff you don't want to. This is why we use the kinematic ones. You have more control of what that does, but still collides against the world.

Yeah, we try to just wrap it and make it easier for the user. Just giving them simpler functions like, "Okay, this is your velocity," but don't care about the collision. The collision is going to be handled by the physics engine. Maybe if you want to get like a callback or notification that something has collide, you can. Generally, we try to handle all the collisions and everything internally because it's way, way too difficult for users to mess with that, I guess.

[00:26:01] JM: Definitely. Now, we have talked about how to actually program some of the basics of a game in Godot. But I actually want to understand how the game engine is built and how it runs. If Godot loads on my computer, if I'm playing a game that is in Godot, give me a quick overview for the software architecture for Godot when I start up my game.

[00:26:35] JL: Godot is basically is something that has like a main loop that receives events, certain stuff and [inaudible 00:26:42] to just process every frame. The same tree I have mentioned before, which is that you make with nodes and scenes, just receives this stuff and has some logic. When you make a game, you just make a scene. You save it. When you want to run it – Basically what Godot does is just run – There's just a single Godot binary. When you don't load Godot, it's just a single binary. It's a very tiny binary. People usually are surprised because when they download like Unity or Unreal, they have to download gigabytes of stuff. Godot is just like a 20 megabytes binary. You download it, you click and it just – You have your full game engine in there.

This low binary, it can execute scenes. It's basically something that executes scenes. Internally, it also has an editor, which is the one you use to make the games. The editor is actually made

using the engine itself. All the UI, all the view ports [inaudible 00:27:35] actually made with the engine itself, which is pretty cool, like self-made. You use that binary, it's kind of the same, but it doesn't contain the editor. So it's much smaller. It's like 20% of the size. This is used what – What it does is you can still all your files when you export. But at Godot, we'll also let you like compress all your files and put them inside a single file just for easier deployment and distribution. It's basically that, just a single binary that lets you edit games and run them.

[00:28:08] JM: Right. As you're saying, there are – When you're developing a game in Godot, there is an editor and the editor allows the developer to change the game. When you deploy the game to the actual end user, what you said, it's 20% of the size of the binary when you're in editor mode?

[00:28:35] JL: Yeah, the editor is most of the binary. It's most of the code of the engine also. The editor is huge. But it's all C++ and inside of a same project. I mean, the editor uses the engine in C++. But when you use the engine, you actually don't really use C++. You use a scripting language that comes with the engine, or you can actually use mono if you want for C#. We have a lot of users that use C# with mono. In that case, you have to download the mono SDK and use it with Godot. Normally, most users just use – Or scripting language, because it's like easier to use and Godot is made to be an easy to use game engine. It all comes bundled in the binary. Yeah, it contains the editor, the script interpreter and everything. You can actually use – It's very funny, because you can extend the editor when you want to make plugins and stuff. The same way as if you were making a game, because you can run the same script, access the same objects. Maybe you have a few more objects for the editor that give you an API to extend. But it's still mostly the same as you used when making a game. If you know how to make a game, it's very funny because you can extend it very easily to make a plugin or something for the editor.

[00:29:42] JM: How resource intensive is it to run a Godot game on my machine?

[00:29:49] JL: It runs on a really low-end stuff. Godot can run – It's very optimized. Then you version Godot for, it's going to be more optimized. But it's very – It can run very [inaudible 00:29:59] computers. Right now there are just three, which is the stable one. Supports open GLES 3, which the same as open GL 3.3. Open GLES 2, which is similar to open GL 2.1. That

goes back to computers from 2005 or something like that. You can actually run it on very, very old computers maybe with more limitation, of course, but it will still run. You have to imagine that it also runs on mobile devices, which are like much slower than computers from desktop [inaudible 00:30:31] doesn't run on mobile, but the entity is the same, which it runs everywhere. You can run also the web via WebAssembly, which also is like more limited. But you can still run it and run your games on the web.

You get an idea. I think the engine binary is about 10 or 15 megabytes, and then everything else is whatever you make on your game. The rest is your own data from your game. If you want something much smaller, Godot, since it's open source, you can get the whole source code and you just disable parts of the engine you don't want and recompile it without that. Then you can create much smaller binaries if you want. When you're completely sure what you're using, what you're not using, you just can't recompile. You can do this on release at the end of the development. You just compile a smaller version of the engine to shape for your game if you really want and if you – Most users don't really need it, I guess. But so many, it's useful, especially for mobile I think.

[00:31:27] JM: If you were leaving out different parts of the game engine, what would you be leaving out? Like the audio part of the engine if you don't have any audio? Could you give me some example?

[00:31:38] JL: Yeah. For example, if you're making a 2D game, you will most likely don't want most of the 3D stuff. You just compile it out. Godot also comes with a lot of tools like image format loaders and this kind of things. You can just like audio formats. Godot has a lot of modules that compiles when you build it. A few [inaudible 00:31:57] and models of different functionalities. Let me check. I have it here.

Yeah, it has like many scripting languages. You can compile C#. You can use the Bullet Physics Engine or not use it. There's like a couple of dozen of models you can turn off if you want, if you don't want them, like web sockets, SSL for encryption. There's just way too many things you can turn off if you don't use.

[00:32:21] JM: Tell me more about the engineering process for building Godot. It started off just as you writing the game engine, and then eventually more people started latching on after the source community developed? Can you tell me the story about how it went from just you working on it to more people?

[00:32:45] JL: Yeah, sure. You make an idea to begin. Maybe it could be interesting to stress this, because it's an enduring podcast in the end. My way of seeing software engineering, there are many ways of software engineering. You will see lots of people making different softwares and APIs and operating systems and whatever. I'm a person that is very, very – How can you say? Like very pragmatic. Very use case oriented. To me, if you create software, you have to do it exactly for what you are requested to do it.

I started as a consultant and I wrote Godot for clients to use and I will hear what they need and I wouldn't mean not do exactly what they want me to do, but they would like do something that just solves their problem. The development is Godot is very problem solving oriented. If you're going to add like a new feature to the engine, you will have like contributors that will come and say, "Hey, I will add this because it's super cool." Usually we'll just reject that because, for us, what's important is to add stuff that people will use exactly in the way they need to use it without anything else. Because a very common engineers fall into is trying to like future proof or make something more complex that it needs to be.

Because you see like – Or try to like, for instance, you imagine three use cases and you want to create a feature that handles all three because it's very flexible. I'm actually always very against that kind of stuff. I actually try to write very down to the ground software that does exactly what it supposed to do. If it doesn't, we change it. But never try to predict what the users would need and try to get ahead and write software that will do that. Because in the end, if you do that, I will say that maybe 90% of the time you will be wrong. You will be ending creating something that is more complex and more difficult to understand and to maintain that nobody will use that complexity for because you just guessed that maybe somebody will use it and somebody will not.

Based on that, the engineering process of Godot is very proposal-oriented. It used to happen a lot, but because the source code is very, very understand. Since it's very so much to the point, I

mean, you're going to have to – When you use Godot, you don't really have to learn a lot of abstract systems. Everything is down to the ground and does exactly what it has to do. The code is very easy to read and understand. This makes that – We have other contributors that want to add new stuff and they send us pull request in GitHub.

Many times then send pull requests and we are like, "I have no idea if anybody is going to need this." What we're doing now is something very proposal-oriented. Before writing anything new, we ask users and contributors to submit proposals and then we discuss them, and we can discuss for months about the best way of doing something is. When everybody agrees, then we add it.

The enduring model is very like consensus-oriented. It's like nobody will – I mean, I usually can have the final word on something, but I will not try to. Usually, what we try to do is to look forward to consensus. You have to think that we have like – Or a thousand contributors to the engine. Managing this is extremely difficult especially for an open source project where those contributed do it on whatever they want whenever they want. You just can't force anybody do anything. It's not like they are your employees and you tell them, "Okay, do this," and they do it.

The only way to move forward with a team of such size is just to find the consensus. We try to work a lot based on proposals and discussing those proposals and finding consensus. When everybody agrees, then somebody will try to make an implementation and do a pull request for evaluation. It's a very bureaucratic maybe, but I think usually the end result is great. Because if you compare it maybe to commercial software, something like Unity or Unreal, you know they have clients. They pay for it and companies using it may need something, or even Epic itself is a company that makes games and they may need features and they may need them in an urgent timeframe because they need it for an upcoming game and they would implement it in the way they can because they have a fixed timeline, a deadline. But we don't have any deadline at any point. We take much longer to decide how to do stuff, but when we do it, we do it great.

What see like implementation, usually like – Let me give you examples. If you look at Unity, Unity has maybe like 3D for a user interface systems. Several ways to do like rendering, post-processing, entity component systems. You will see that Unity implements in something. Then they are like, "Let's do a new system." Throw away the old one. Make a new one. But you still

have the old one, they have like many particle systems. Because usually when you have to sell something, you have a deadline and how to implement something. They have many input systems. Godot has only one UI system. One input system. One way to do rendering [inaudible 00:37:46]. Really, what I mean, if you look at it, it's really easy to use, really well-designed. We took a lot longer to do it, but usually it's something easier to use. More to the point, exactly what users need. This is kind of the way we do engineering. I hope it's kind of clear. It may not be so simple to understand. But we are very use-case-oriented, very propose-oriented, very consensus-oriented. It's the only way we can, because it's not like we are paying a lot of people to do everything we want with a clear leadership. Everything has to be discussed, because if our contributors don't feel that they agree with the direction, they won't contribute. So we need to get all to agree and make sure we agree on what we want to do and move forward. It's very challenging. It's very interesting.

[SPONSOR MESSAGE]

[00:38:40] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[00:40:28] JM: When was the point where other people started working on the game engine? Can you tell me the story about going just from you to more people? Was there a certain point at which other people wanted to start working on it?

[00:40:45] JL: It's kind of pricey. For a longtime, it was just Ariel Manzur and I working as consultant. When we open sourced it the first year, we didn't have almost any user. Lots of people liked it and saw promise in it, but they filled our issue tracker with improvements like ideas, like how to improve visibility, what is broken?

The first year, maybe the first two years, since it was open sourced, it was open sourced in 2014. So maybe until '16. All I did was just improve usability and make it more usability for users. At some point between 2016 and 2017, I think by the time Godot 2.1 was released, it reached a certain level of maturity that most people using it really liked it. From then it started growing like exponentially. At some point, I wasn't able to deal with all the issues and everything because it was too many users and I was doing this as a hobby. I asked for help and then Remi Verschelde stepped up, who is the product manager. He's doing an amazing job so far. He coordinates all the contributors. He coordinates all the documentation. He basically knows what everybody is doing and what the state of everything is. While I can mainly focus on the technical side and so much on the people side.

That's actually working really well so far, but every time, like every year, it's just like doubles in size. It's really exponential. We need to figure out ways to keep doing what we are doing because it's too many people, too many users, too many contributors and you can't just cope with that. We are now looking to extend that. We have like our core developers, which are the most experienced contributors. We try to give them like areas of the engine they are in charge

of. Usually they have final say on that. They usually can ask me or other guys about that, but they usually have final say on this.

We can start distributing the load, but it's very difficult. We are usually running behind the growth all the time. We are always trying to cope with past grow while at the same it's growing bigger. This is why a few years ago we started asking for donations, and with the donations, I mean I was a business consultant for an industry. I was just doing this out of hobby. I had to decide at some point to just move fulltime to work on it based on donations. It wasn't easy because I had a whole career on this. I have to throw out the window to dedicate fulltime to this. But since people loved it so much and it was growing so much, I think it was the right decision, because it's really enjoyable to accompany the growth. We now have a few hired developers, not many. We have grants from companies like Mozilla Foundation and Microsoft. We got the Epic MegaGrant recently. That allows us to have a few more people hired, but we are still a very small core team of people working fulltime, like 6 or 7 versus a thousand that do it in the free time. It's not really easy, but we are slowly figuring out ways to do it. I mean, it's just a super interesting challenge. It's very stressing, but it's very enjoyable.

[00:44:03] JM: I imagine there must be this feedback loop between games that are using Godot and the open source project itself. I imagine there are games where people do things in Godot that push up against the boundaries of what the game engine allows for which causes breakages perhaps, and then an issue gets opened on GitHub like, "Hey, can we change this thing so my game doesn't break?" and it leads to developments in the game engine. Can you tell me the extent to which that's true? What's the feedback loop between individual games and the game engine?

[00:44:44] JL: That's actually very complicated. It depends on the user also. It depends mostly on the user I guess because sometimes you hit a boundary, and we know that that boundary exists, but we can't solve it right away. For example, for 3D rendering, Godot is quite behind the other engines in performance because it's using very old ways of doing rendering. We knew this for a while. We are working on Godot 4 right now that we right entirely written rendering-related. It's going to take a while. In that case, users have to wait. There's nothing we can do. In other cases, users just know how to fix it and they just submit a pull request like, "Hey, this fixes this

problem. Now it's much faster," like these kinds of things. That happens a lot. We have a lot of pull request happens. They need to be evaluated and everything.

We used to have people just submitting the pull request with something they need, but many times we are like maybe that's not the right solution. If it's for a new feature, usually we ask users to open up proposal explaining how they would solve it or maybe inviting others to think of a way, and then we start the process of discussing how to properly release and sometimes it's just something that it's so simple so they just submit a pull request for a new feature, but it's very simple, so just merge it.

It's a bit more difficult with large companies using it because they really need something and they have a deadline and they have a lot of money invested. Usually, what those companies do is they have their own fork and they implement their solutions, like their own solutions early. Maybe they will do a pull request for us with that solution. We probably may not merge it yet because we want to do something cleaner. But we know that it's something that needs to be solved. So we put our priority on that.

Eventually, maybe a few months later, we actually implement this properly and the company has moved from their own solution to what we implemented or maybe not. Maybe for that game, they already have that branch and that's it. This is in essence not so good, because companies have to alter the source code. But in another sense, it's also pretty good because many companies using like Unity or something like that, when they run to engine limitations, there's nothing they can do. So they just try to compromise quality and do something that just works on whatever is there. I think if they have more freedom, they can actually modify the engine to fix what they need. Maybe just do a quick hack that works for them and we wait until we can implement a proper solution. Yeah, it's a very interesting way of working in think. There's a lot, an ecosystem of companies contributing goal, benefiting each other from it like through open source project that you won't really see much in commercial software. It's very interesting.

[00:47:28] JM: Godot can run on iOS or Android. It can run on all the different desktop platforms, Windows, Mac, Linux. Tell me what you've needed to build to allow it to run in all of these different environments.

[00:47:45] JL: Yeah. Godot works pretty much where we can make it work. Works on, yeah, Linux, Mac and Windows for the editor. You can export to the same platform, like desktop. You can run on Raspberry Pi. You can export to mobile both iOS and Android, and you can export to web via WebAssembly. Works pretty well in most platforms. What we can't do is console support, like Playstation or Nintendo Switch because those platforms are completely closed. There is no way we can have any source code publicly using those platforms. The only way you can actually get access to their APIs is by singing a very [inaudible 00:48:24] I guess NDAs that won't allow you to give any acknowledge of whatever they are doing.

We don't support consoles, but since the license of the engine is MIT, there are companies already that really the ports to console. There are two companies that specialize in console porting, which are Pineapple Works and Lone Wolf. You can license, if you're making a game and you want to publish in console, you just ask them to license it. Usually, most people think that running on console is just expressing export that it runs in the console. Not really so much that. You need to comply with a lot of technical requirements for console games. Usually, most games that publish in console need companies that will port it anyway. You just contact any of those companies and they get it running there. Yeah, we can't officially give away any of the – How could you say? We can't really license for consoles and publish a source code because it's completely forbidden by the NDAs.

[00:49:24] JM: Let's say I want to collaborate on a game that I've built. I want to higher designers and engineers and storywriters. I want all of us to work together. I know some of this collaboration can take place in the Godot editor itself. But can you tell me a bit about how collaboration works on an average game?

[00:49:50] JL: Yes. Usually, when you – I think you probably remember that at the beginning we discussed that Godot uses scenes and nodes and scripts. When you save those, they're just text files pretty much. You just save them as text files to the file system. Your game is pretty much mostly text. You can say binary scenes if you want, but mostly it will be text. Then [inaudible 00:50:11] images and running is everything is binary and you just use something like git. You can just use git and use a repository for developing with others. Godot is mostly text. All the asset like scenes and those are texts. It's a very readable text format. If you just change something, it will be manageable and fine like any source code. Collaboration is mostly done

with existing version control tools. That works very well actually. If you use a lot of binary files, you can use [inaudible 00:50:42] support for binaries. Yeah, usually you just use version control normally. Godot is assigned. If you merge other branches that's going to like keep working, it will try to – It's designed so it generates as little conflicts as possible between files working on different machines. Yeah, it's just optimized for version control pretty much.

[00:51:03] JM: All right. To close off, I just love to know why have you focused your career on games? Why have games seem so appealing to you?

[00:51:12] JL: I guess I'm a person that likes the technical challenges. Also, I am a person that is very creative. I like like music and making 3D stuff. [inaudible 00:51:22] like the technical stuff on the creative side. Honestly, I think games is the most – One of the most fitting areas for these because you can do really complex math and complex algorithms and optimizations, and I really enjoy that. But it's all for the sake of creating something nice. It's not just like maybe improving a database or something like that. This is something artistic. People create amazing games with your tools and that is very rewarding.

Yeah, I guess that makes this really enjoyable for me and probably for everybody working on these, because most them, like technical guys who work on games and game technology, we play games, but we just don't enjoy the game experience, but you also enjoy the technical parts of games. We can see a sky and say, "Wow! That shader is amazing. I wonder how they did the sky shader," or you can see an explosion and you're like, "Wow! What's that VFX? How did they do those particular effects?" We try to figure how they did it or maybe we can figure out how to do something better. This [inaudible 00:52:23] artistic and technical thing is something that game technologies is great for.

[00:52:30] JM: Well, Juan, thank you for coming on the show. It's been real pleasure talking to you about Godot.

[00:52:33] JL: Okay. It's been a pleasure.

[END OF INTERVIEW]

[00:52:43] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how ExtraHop delivers cloud-native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily, if you want to check out ExtraHop and support the show, go to extrahop.com/cloud.

[END]