Spice up your app with 3D

Alexander Larsson

>> All right. Can People hear me?

>> Yes, we can hear you.

>> All right. Cool. Wait a bit until we start, then.

>> Yes.

>> I think we can start. So, this is GUADEC 2020, day three, track 1. And we start with Alexander Larsson and spice up your app with 3D. Please.

Alexander Larsson: All right. Thank you all for coming. Even though it's the morning slot. Although it's not morning for me. So, it's kind of weird. I'm here to talk about spicing up your app with 3D. Using a library I have been writing or porting recently called Gthree. Let me see here.

So, Gthree, it's basically a library that you can use to render 3D scenes. It's not dependent on GTK, it's meant to work in the GTK and the GNOME frameworks. It's really a port of the JavaScript library Three.js. Which I guess some people are already aware of. It ports all the JavaScript from -- to C. And then JavaScript uses WebGL. But we port it to OpenGL which is more or less the same, right? So, but huge chunks of Three.js is actually written in GLSL which is the GPU programming language that uses -- it's used the same in WebGL and OpenGL. So, we can basically reuse all of that. A large part of Three.js we can just lift in entirely.

And design-wise, it's not a game engine. I mean, you can write games in it. But it doesn't center all the infrastructure that a game engine has. It's a low-level bag of bits that you can connect to make things happen in a much more flexible, but also less-finished way. So, it lives in the -- in the GNOME ecosystem, as I said. We have GObject as a base. It should be usable from whatever language that you have bindings. GPU for rendering, and it uses the Graphene library for 3D maths, vectors, all this stuff. It's used by GTK4 now. These are all core GNOME dependencies. It doesn't depend on GTK, but there's an extra library for GTK. We support both GTK3 and GTK4.

In theory, you can use it without GTK, but you have to do some hookup to get the basics in GL working. 3D is all about rendering something called a scene graph. The scene graph describes the hierarchy of objects in 3D. Three dimensional objects put into two dimensions. There is like the widget library in the GTK layout, only it's in 3D. And the equivalent of a widget is an object. A GthreeObject which is a base class that, you know, defines the child/parent hierarchy and it also has a transform. So, every object transforms itself and its children. Just recursively. So, like the outer most thing affect all the things inside it which can then have further transforms.

There are also some structural types like GthreeGroup. Which is kind of like GTKcontainer in that it doesn't really do anything. It just groups things together so you can transform them and refer to them as one. GthreeScene object, which is basically a group, but it has some extra properties that are used for the root object such as background color and things like that.

You can actually have a scene wherever you want. But it's only useful -- the scene-specific properties are only used if it's the root and the hierarchy.

Then you need some kind of visible object, something interesting to render in your -- in your scene graph. And these are the things we have. Mesh is the most commonly used one. It's the set of trying the render in some way. That's the most common way to render in 3D stuff. It's meshes of triangles. But we have lines and sprites and points. Sprites are being like two-dimensional billboard-like things in three dimensions.

Once you have all the stuff you want to render in a scene graph, you want to use a camera to specify a view into the scene. This is where things differ from like a GTK hierarchy that you have one way to view the thing. Here you can have multiple cameras pointing in different directions inside scene. And a camera object, or a camera -- the camera type is a GTK3 object. You can put it in the scene. Which can be useful if you have like a car in your scene, you can put the camera inside the car and as you drive the car around, the camera will see what the person inside the car would see, right? And the camera is a base class and we have different types of cameras.

The perspective one is the most commonly used one because it has like a traditional perspective view where things that are far from the camera are small and the things that are close to you are big. Whereas the orthographic one is an engineering style, a non-perspective correct view.

Most objects render as dark if you don't do anything by default because they need light to be visible. So, that we have light objects that you need to put in your scene, in the places where you want them to be. And this will affect the things that are close to them. But by lighting up a nearby object. And also, optionally, lights can cause objects to sort of throw shadows on other objects. But that is kind of expensive. So, it's not on by default. You need to set up which objects throw shadow and which lights throw shadow.

But we have a base class here. GthreeLight. Which defines some basic property like color and intensity. And then we have different types of subclasses where ambient is a global lighting that affects everything equally. Which you probably want -- you want so far some ambient light so that things aren't completely dark. And then you add point of spotlight for specific lighting. A point light is kind of like lightbulb. Whereas a spotlight is like a spotlight, right? It has a direction and like a cone of light. And these are more like sunlight and sky/ground kind of lighting.

And once you have these, a camera and a scene, you put them in a GthreeArea. Which is actually a subclass of GtkGL Area. The GTK open GL widget. And it creates the rendered object which is the main rendered object. It's more like an internal object. But it's what tracks all the state stuff that we need to do in OpenGL. And the full implementation with the render signal in the area uses this renderer to draw this particular scene from this particular camera. And most of the time, that's all you need. But if you want to do something else, maybe you have a multi-scene rendering or maybe you have some kind of effect you want to apply, then you can override this render signal and do something else.

Another important thing to have is textures. Textures is the Gthree image data. It's kind of like a PixBuf. But refers to the CPU. We have 2D like cube textures where you basically define six sides of a cube and that defines like a three dimensional thing. So, if you're in the middle of the cube, you can look in any direction and there's a color that's a 3D kind of texturing. From a point, in any direction there's a color value. And the most common thing to do this with the textures is color mapping. Using UV mapping. You define UV coordinates for every point and interpolates the coordinates over the triangles and we read the color from the texture and, you know, use that as the color that have pixel.

But we can also do all sorts of more complicated mappings. Like normal mappings where we have -- we interpret the -- the image data as a vector that perturb the normal of the surface so we can create, like, small scale changes in surface geometry kind of like sometimes called bump mapping where you can add geometry where the actual triangle doesn't have it. Which, you know, it doesn't affect the geometry really, but it affect how it interacts with lighting, for instance.

And there are other kinds of -- I didn't write down them all here. There are different types. But environment mapping is a common one where you can fake reflections and refraction. We also use textures for sprites where you want to have -- render some image in the middle of your scene, for instance.

And we have a type called materials that specifies how primitives are drawn. So, if you have something with a particular material instance, that instance has a bunch of properties. You can set colors on it. You know, textures, various types of textures. And the type of material you use affect how the objects respond to things like lighting and shadow. And the most typical material you would use is the meshStandardMaterial that implements something called physically based rendering. Which is a very modern rendering model for basing -- based on, you know, physical -- physically-correct lighting model that is what normal games use these days. It's a bit more computationally intensive than some of the simpler models like the Phong lighting models that we have. These are kind of old-school, artificial, a bit plasticky-looking lighting styles.

But they have their place. If you want that look, those are nice to have.

And then we have geometry which is basically, you know, vectors of position. So, we have a geometry defines a set of points. And each point also has additional information that like apply to that point. So, you can have a color for each point, you can have a normal at each point. You can have coordinates for texture mapping. And then you take those data and upload it to the GPU. And then you interpret it, depending on what kind of object you're rendering.

The most typical interpretation would be every three pixels -- or every three vertices are the corners of a triangle and then you draw a triangle. And the -- the geometry data is actually really extensible. You can add kind of attributes that you can write shaders to access your own properties for whatever thing you want to do with it.

And the most common thing to use to render these is the mesh. As I said before, it's basically a geometry plus a material. And it automatically interprets the geometry as a bunch of triangles. And it can fill them or draw them in wire-frame. And in most cases, mesh is all you have, right? And they go from really simple things like we have a bunch of primitives for generating, you know, box geometries or sphere geometries. You can put those in the mesh with a material and then you render a simple primitive. And you can add multiple of those. But it's more common to have a more complex mesh like an entire enemy in a game or the player or something. Or even as large as an entire level. It's actually nice to have as large meshes as possible rather than having multiple meshes. Because you will be able to upload the entire thing to the GPU and then have the GPU render it in parallel, which is much faster than doing it on the CPU. You should strive to have like large meshes. A few large meshes, basically.

And there are, as I said, functions to create these really simple primitives. But normally what you do is you use a 3D editor like Blender to describe your model and do all sorts of modeling and editing textures. And export to this file format called glTF, of which we have a loader. The Khronos group, the group who defined the OpenGL spec. This is a widely-supported format. It's in JSON for the structure and binary arrays for the vertex data. We can have textures as separate JPEG files on the side or they can be embedded in the glTF format. It's a very feature-full format. Has everything including animations and stuff. It really can touch almost all the features of Gthree from a single file. It's very flexible. And as I said, widely available. There's a built-in plugin for there's built-in exporter in Blender and I guess every other 3D tool you can imagine.

And it's kind of boring to list all the APIs. So, I'm gonna do a bunch of demos instead. Because I think that's a much better way to -- to describe these things. But we have all sorts of features. There's an animation framework where we can define keyframes and it will automatically animate or interpret between the keyframes. And we can trigger actions. And when the actions are done, they can stop or trigger other actions or loop. And it's very flexible.

We have a skeletal animation system where we can define a skeleton for your mesh. Which kind of did issue it's a simplified version of your mesh that is connected to it. So, when you move the skeleton, you move the -- you move the mesh around. Which is a simple way to do -- to make our character animations and things like that. We have an effects stack with some built-in simple effects. But also very easy way to add your own if you write some GSL code. And we can plug those into the rendering system and it applies an effect. And we have shader materials where you can basically dump some GSL data and we texture based on the output of that. Which is very flexible.

If you have ever played with the shadertoy website, it's kinds of like that. We can just do all sorts of stuff in a shader. It also has raycasting support. It's used for if you want to click on something and you throw a ray into your scene and see which objects are hit. It's the basis for basically mapping the mouse into the 3D space in a way that allows you to modify or interact with the 3D scene. And more complicated things like we can render scenes to textures. And you can use those textures when you render the scene again. So, you can like fake mirrors and all these other kind of tricks you do in 3D.

But as I said before, much more interesting if we do some demos. I think I've got to stop that and... there. It's a bit janky, I guess. Over this stream. But here is the simple demo of a cube. It's got semi-transparent triangles and some lines. And you could, probably, if you wanted, implement it in some kind of role OpenGL. But it wouldn't be fun. Whereas this is just a couple of lines of code in Gthree. And if you wanted just a spinning cube in the about or something, the level of dependencies this adds on top of OpenGL itself is very small. It's not like importing an entire game engine into your app. It's just a small dependency and you can easily get something like this. Not super-interesting, though.

get to the more complex stuff. Here's an example of what you can do. These are all the same geometry. But they apply different materials. And some of the materials have like textures. And some are wire-frame or had per vertex or per-face colors in the geometry. Oh, it's more like a demo of what you can do.

Here's a demo of environment variable. Environment mapping. Which is a way to fake reflections and refractions. The middle one is a refraction. So, it's like a -- it's like a glass sphere, magnifying glass or something. And you can tell these are fake because the reflections aren't actually like -- they're not reflecting the other objects. They're just reflecting from the reflection map. Which is also set as the background as a 3D texture in the background. But, I mean, for certain things, this is very much what you need. And if you really wanted, you could render the entire scene into a cube map first and then apply that if you want to have the real reflections. But that's a lot more work.

Here are some of the primitives we have for simple primitives that you might want to use. The white things are the normals and you can see from the texture the UV mapping of the texture. Here's a single scene with three cameras. So, you can sort of view the same scene in multiple ways that you want to do a 3D editor or something like that.

Here is a sample glTF model from the glTF sample. They have a GitHub repository with a bunch of sample models. You can see all sorts of cool things like partially reflect things, metallic rusty things. These all reflect the -- the environment back. You set different maps, you get different kinds of reflections. Pretty cool. So, this is -- this is -- this looks pretty cool. But it's really a simple model.

You can have more complicated ones too. For example, here, say, more cartoonish style where we have keyframe animations in it. And so, this is still using the -- the physically-based modeling material. But you just done in a way that looks more cartoonish. It's pretty cool. This is just something I downloaded off the Internet. But it's a pretty nice model.

We have support for something called decals, or whatever. Which is where you project a texture map on top of an existing geometry to kind of make a project -- a decal on top of something. It's commonly used in games, for instance, if you want to have bullet holes in your walls or you just want to add extra -- if you have a large-scale texture and you want to add extra like small scale details, you can use these. It's pretty cool.

An example of the skeleton animation. So, this model -- so, it really looks like this. The model. And the green and blue things are the bones and the skeleton. And then there is a -- there's a set of animations that apply to the skeleton. So, we can enable --

>> 5 minutes left.

Alexander Larsson: Animation, for instance, and it will -- it will slowly move and then we can cross-fade to a different animation to get smooth animations. Between animations. Smooth -- it's probably not smooth on this stream, but it is if you run it locally.

We have shadows, colored lights. Shaders, the run and texture stuff. This is just a bunch of -- each one has -- runs a get of GLS code that generates the textures. Sprites are more like 2D billboards in 3D space. And we have a point primitive, which is a cloud of points. It's more interesting if you use effects. Like you can use it to create fire and smoke and things like that.

We have an effects system where you can stack various affects on top of each other. And enable and disable them.

>> One minute left.

Alexander Larsson: Interactions where you can point at things. We have a cartoon material if you want to have a more non-physical thing. wrote a game -- or I ported an existing game. And I created Wanda I want to add somewhere. So, I guess that was all the slides. And I really think people should look at this and see if you can add some cool 3D stuff in GNOME. Maybe we could add some Easter eggs. Or maybe we can have some natural place where we can add 3D effects in the desktop. That would be cool. But I guess that's the entire slide. And I'm gonna look at the questions and see if anyone has any questions.

All right. Do there exist tools to convert to glTF? My old code that I'm updating reads from .mdl and .md3 files, for instance. I'm not aware of exactly what those formats are. But I'm pretty sure that you can find converters for almost everything into glTF. Because it's such a widely deployed format. If you can import your stuff into any -- if you can import it into Blender, for instance, or, you know, Tree Studio or whatever kind of 3D editor, I'm sure you can export to glTF. It's really common.

And can we have some 3D Easter eggs? Yes, I think so. I want to add the Wanda one somewhere. But we can have the two. They would be pretty cool. I want to have some other kind of use -- of 3D in the desktop. But I couldn't really find any good places where it would make sense. And like... in the core desktop. There wasn't really any places where we could put 3D stuff.

Yeah, so, there's a question. Wasn't there a doom kill thing at some point? Yeah. I remember someone actually modeled the real doom, I think. Where they imported all the processes as enemies. And if you shoot an enemy, it like killed that process. So, you -- I don't think that's necessarily a great UI for killing processes. But it was probably fun. Yeah, I mean, there's another question of using this for the shell itself. For like 3D effects in the shells. I don't know if that's really possible. Because the shell renders in a completely different way. And it's kind of hard to inject Gthree into the shell.

And there's a question about language bindings. I mean, it's -- I wouldn't say perfectly introspectable. We probably lack some properties for some things. So, but if someone starts using it for any object introspectable language and finds some things that they would like to set that isn't possible, it should be easy to just add properties that are required for it.

>> So, it looks like that's all the questions. Let's say thank you to Alexander. And we will be back at 17:45 with the next presentation. Thank you, Alexander.

Alexander Larsson: Thank you.