Foxus

Release 0.1

Zelle Marcovicci

Feb 16, 2023

CONTENTS

Foxus is a project experimenting with passthrough Augmented Reality on existing devices. It uses a stereoscopic camera add-on that runs on the Oculus Quest 2 at 60fps, with full RGB passthrough. The Foxus is built with community support by the team at voxels.com.

Note: Here be dragons! Watch out! This project is under active development.

For now, to find out more, make sure you visit the Foxus website and Foxus store.

You can also join the Foxus discord server for discussions, contributions and support.

CHAPTER

ONE

BUILDING FOXUS

Note: Here be dragons! Not all environments are currently supported, and those which have support may not have detailed build instructions yet.

Notably, Windows builds for Foxus are not yet supported. There's still some kinks to be worked out with the OpenCV libraries and other dependencies.

Although it's still a big unknown, you can always try building Foxus under the Windows Subsystem for Linux or other virtual machine solutions!

Got Foxus built out on something not mentioned here? We'd love to hear from you!

1.1 Linux (General)

You'll first need to download the X11 build dependencies for Godot, as described in their documentation. They provide a set of one-line commands specific to your Linux distribution to get started.

You'll also want to install Android Studio. Go to https://developer.android.com/studio/index.html and find the Linux download. Go through the standard install process. When you're on the screen reading "Welcome to Android Studio", click "More Actions" and then "SDK Manager".

On this screen, you're given the install location of the Android SDK, which you'll need to know in a moment. You can also install the required NDK here. Go to the *SDK Tools* tab, then check *Show Package Details* in the bottom right. Now you'll be able to select NDK version 23.2.8568313 and install it by hitting Apply.

Set the *ANDROID_NDK_ROOT* and *ANDROID_SDK_ROOT* variables, using the install location of your Android SDK from above. For example, if your Android SDK was installed in */home/Android/Sdk*, your commands might look like this:

```
export ANDROID_SDK_ROOT=/home/Android/Sdk
export ANDROID_NDK_ROOT=/home/Android/Sdk/ndk/23.2.8568313
```

If you don't have any installed already, I also recommend getting *unzip*, *clang* and both the JRE and JDK for your preferred version of Java. (Java 11 works well.)

You can now use the console to clone and build Foxus. Each of these steps will take a while.

```
git clone --recursive https://github.com/cryptovoxels/foxus.git
cd foxus
./build_godot.sh
./build.sh
```

Finally, to run the editor:

```
./run_editor.sh
```

Go to Editor > Editor Settings in the top bar. Here you'll have to set the Android SDK path (it's the same one we found earlier) and the debug keystore (which will be in the foxus folder). After that, you're good to go and you can build this project out to your Oculus headset.

1.2 Linux (Steam Deck)

There's a few things to consider before we get started…

The Steam Deck OS ships as "read-only" which prevents us from installing the necessary packages to build Godot and Foxus. This means that we'll be changing it to read-write mode immediately, which is recommended more for advanced Linux users since we can make changes to the OS. Don't install packages willy-nilly after this or you may break something!

The Oculus app currently doesn't support Linux. This means that we don't have access to the Air Link features, and so, to build to your headset, you'll be using a wired connection. Meta recommends a very expensive Oculus Link cable, but it seems to me any USB-C cable may work. Cable management may become an interesting challenge for you. Whenever possible, try using Bluetooth connected keyboards/mice to free up some ports. I'm using a generic USB-C dock with USB 2.0 ports, with an adapter to plug the USB-C cable into one.

After you've used a wired connection, you can use ADB to switch to a wireless one temporarily. This lets you quickly iterate through builds with the Foxus cameras plugged in, which is a real plus.

As our first step we'll be setting up a password to use sudo commands. Switch to Desktop Mode if you haven't already, and open up the Konsole app.

```
passwd
```

Don't forget the password you set in this step! You'll need it again frequently.

Next up we'll disable read-only mode on the OS, and make sure pacman (which we'll be using to fetch packages) is up to date.

```
sudo steamos-readonly disable
sudo pacman-key --init
sudo pacman-key --populate
sudo pacman-key --refresh-keys
```

(That last one might take a while, and I can't tell if it's required or I'm superstitious … but it doesn't hurt.)

The Godot build page has a "one-line" command to get the required dependencies working on Arch Linux setups. However, this will break the Steam Deck's audio libraries if you use them as-is, and there's some stuff missing that we'll have to do ourselves.

```
sudo pacman -S scons gcc yasm linux-headers clang llvm pkgconf libxcursor libxinerama␣
↪libxi libxrandr mesa glu libglvnd alsa-lib libisl libmpc linux-api-headers glibc␣
↪libx11 xorgproto libxrender pavucontrol libxext systemd libpulse libxfixes
```

Yes, a lot of this is a reinstall of existing packages. Just trust me — not all of the stuff you'd expect to work out of the box will unless you reinstall them!

Let's grab the JDK and JRE for Java 11 while we're here.

```
sudo pacman -S jdk11-openjdk jre11-openjdk
```

OK, take a break from konsole commands. It's time to go install the Android SDK & NDK. I recommend using Android Studio for this. https://developer.android.com/studio will have the latest version, so you can navigate there on your Steam Deck and extract it. Go into the "bin" folder and run the studio.sh file (or run it in the konsole if you like.) Go through the standard install process. When you're on the screen reading "Welcome to Android Studio", click "More Actions" and then "SDK Manager".

On this screen, you're given the install location of the Android SDK:

```
/home/deck/Android/Sdk
```

. . . which will help you later. You can also install the required NDK here. Go to the "SDK Tools" tab, then check "Show Package Details" in the bottom right. Now you'll be able to select NDK version 23.2.8568313 and install it by hitting Apply.

We can now set these locations as environment variables for the build process to use.

```
export ANDROID_SDK_ROOT=/home/deck/Android/Sdk
export ANDROID_NDK_ROOT=/home/deck/Android/Sdk/ndk/23.2.8568313
```

Now let's try actually building our special version of Godot. Each of these steps will take a while.

```
git clone --recursive https://github.com/cryptovoxels/foxus.git
cd foxus
./build_godot.sh
./build.sh
```

Finally, to run the editor:

```
./run_editor.sh
```

You're in! Go to Editor > Editor Settings in the top bar. Here you'll have to set the Android SDK path (it's the same one we found earlier) and the debug keystore (which will be in the foxus folder). After that, you're good to go and you can build this project out to your Oculus headset.

1.3 MacOS (Intel chip only)

M1 and M2 chips are currently **not** supported for building Foxus.

Note: Here be dragons! The macOS builds of Foxus are mostly untested.

If you're running a macOS environment and want to share your experiences building out Foxus, get in touch!

You'll first need to download the macOS build dependencies for Godot, as described in their documentation. If you use Homebrew or MacPorts, installing SCons and yasm is a bit easier:

```
brew install scons yasm
```

```
sudo port install scons yasm
```

You'll also want to install Android Studio. Go to https://developer.android.com/studio/index.html and find the macOS download. Go through the standard install process. When you're on the screen reading "Welcome to Android Studio", click "More Actions" and then "SDK Manager".

On this screen, you're given the install location of the Android SDK, which you'll need to know in a moment. You can also install the required NDK here. Go to the *SDK Tools* tab, then check *Show Package Details* in the bottom right. Now you'll be able to select NDK version 23.2.8568313 and install it by hitting Apply.

Set the *ANDROID_NDK_ROOT* and *ANDROID_SDK_ROOT* variables, using the install location of your Android SDK from above. For example, if your Android SDK was installed in */home/Android/Sdk*, your commands might look like this:

```
export ANDROID_SDK_ROOT=/home/Android/Sdk
export ANDROID_NDK_ROOT=/home/Android/Sdk/ndk/23.2.8568313
```

If you don't have any installed already, I also recommend getting *unzip*, *clang* and both the JRE and JDK for your preferred version of Java. (Java 11 works well.)

You can now use the console to clone and build Foxus. Each of these steps will take a while.

```
git clone --recursive https://github.com/cryptovoxels/foxus.git
cd foxus
./build_godot.sh
./build.sh
```

Finally, to run the editor:

```
./run_editor.sh
```

Go to Editor > Editor Settings in the top bar. Here you'll have to set the Android SDK path (it's the same one we found earlier) and the debug keystore (which will be in the foxus folder). After that, you're good to go and you can build this project out to your Oculus headset.

If you're trying to run Foxus on macOS directly, due to how the USB cameras are handled in macOS, the Foxus app has to run as root (Administrator). Use the provided run.sh with sudo to run it:

```
sudo ./run.sh
```

CHAPTER

TWO

THE FOXUS ECOSYSTEM

2.1 Foxus Camera

The Foxus camera is a nifty bit of hardware – two high-quality stereoscopic cameras that go on the front of your VR device to allow full-color passthrough video.

They look great, too.

Right now, these play nice with the Godot engine, using a special plugin called gd_eiffelcam to process the video feeds. You can connect them to a Meta Quest headset using a microUSB to USB-C cable. You can also connect them directly to a computer, where the feeds can be used in other programs with a bit of hackery!

2.2 Foxus Playback

The Foxus playback engine is an app that runs on your Quest 2 headset to give you access to mixed reality experiences. It can be installed via Sidequest, or using an ADB connection to install the latest APK to your device.

This App currently allows you to adjust the colour of the video feed with a menu of LUTS, which is the first MR experience we have implemented. You can use our Foxus Creator platform powered by Godot to create your own MR content to be viewed in the Foxus App.

Rather than a fully virtual reality experience, or an augmented reality experience where digital assets are overlaid on reality, we use the term mixed reality because the Foxus is for processing and altering the optical flow feed through these cameras.

We're also working on adding networking features so that your Foxus can sync up with others in the same room, and you can share some communal MR experiences.

2.3 Foxus Creator

The Foxus Creator engine is powered by Godot, and here you'll be able to create MR experiences on your home computer and preview them inside of a headset. Foxus Creator runs on the Godot engine, but we are developing it for people who don't necessarily have experience with Godot. If you're a Godot power user, we're hoping to also provide a plugin to enhance your regular Godot environment.

2.4 Foxus Storefront (Coming soon. . .)

Made something great? Upload it to the Foxus storefront and anyone can use it inside the Foxus playback engine. Charge money or make it free. You decide if your creations are Mixable (can be edited by the purchaser) or set in stone.

2.5 Other Foxus Toys (Coming soon. . .)

Maybe you don't have a Quest . . . there are alternatives coming soon!

Note: Here be dragons! The dragons here are especially large.

The Foxus Toy is under super secret development.

CHAPTER

THREE

CREATOR TUTORIALS

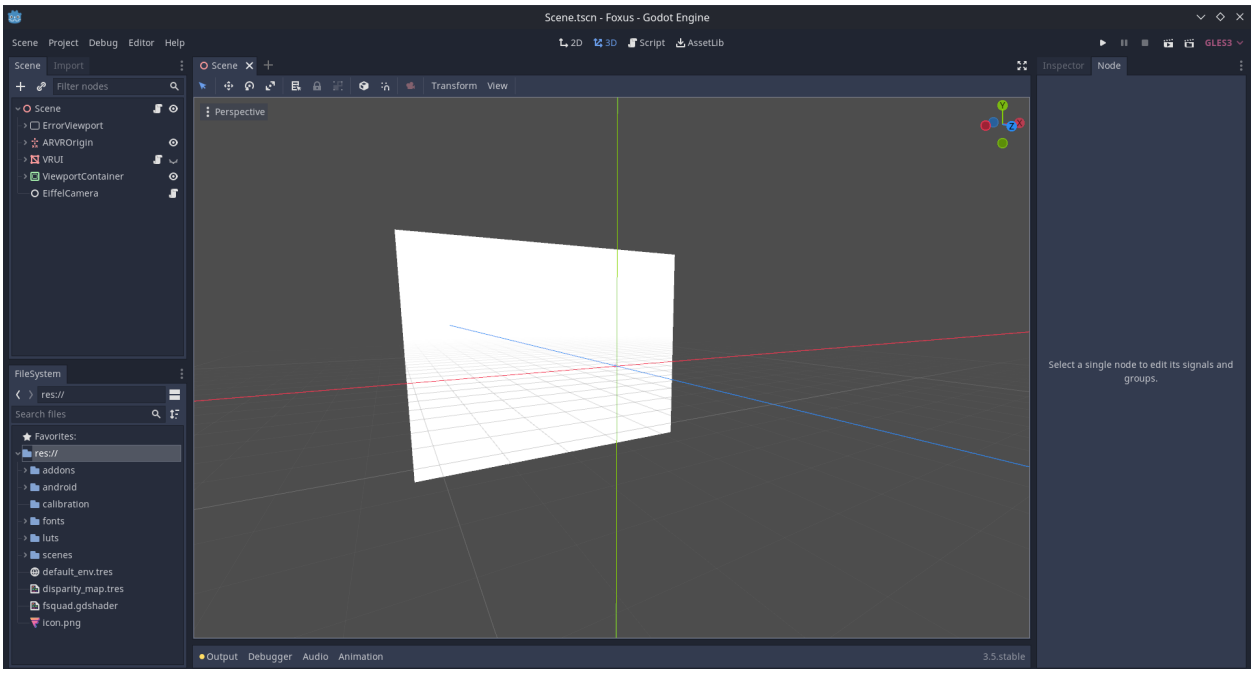
Note: Here be dragons! More tutorials are coming soon.

If you have ideas for tutorials or something to contribute, we'd love to hear from you.

These tutorials are aimed at users new to the Foxus Creator system, with a brand new project that contains nothing but the Foxus camera setup.

If you've never used Godot before, that's OK. We'll quickly run through the different parts of the Foxus project that you'll see when you open it. Otherwise, feel free to skip ahead to one of the more specific tutorials in this section.

When you open the Foxus project in the editor, you'll be greeted with the 3D Scene view. In the center of the screen, there's not much to look at – you'll see a white quad (a quadrilaterial polygon) floating in a grey void. This quad is where we display the feed from the Foxus cameras.



It will probably look something like this.

Check out the left-hand side of the screen and you can see the Scene hierarchy, which contains nested objects called nodes. The top level of this hierarchy is a node called the Scene, and represents the whole project. Inside of the Scene node are a variety of other nodes, two of which are quite important:

- ARVROrigin – this represents the headset wearer's position. Any nodes which are placed inside of (parented to) the ARVROrigin will stick to the user's field of vision, and move with their head.
- ViewportContainer is the node that contains the menu you can access by pressing B inside of the app. This includes options, LUTs, calibration information, etc. You shouldn't need to mess around with this, but it's there.

Clicking a node in the Scene hierarchy on the left will open more detailed information on the right-hand side of the screen. This will be very useful later.

In addition to that 3D view in the center of the screen, you can click on the tabs above it to switch to a few other views: 2D view (which will show you the user interface), Script (which will let you explore code on individual nodes in more detail) and AssetLib (which you can use to download or import outside assets).

Finally, in the lower-left of the screen, you'll see the FileSystem area. This shows all folders and files in your project, including those which are not currently in the Scene hierarchy.

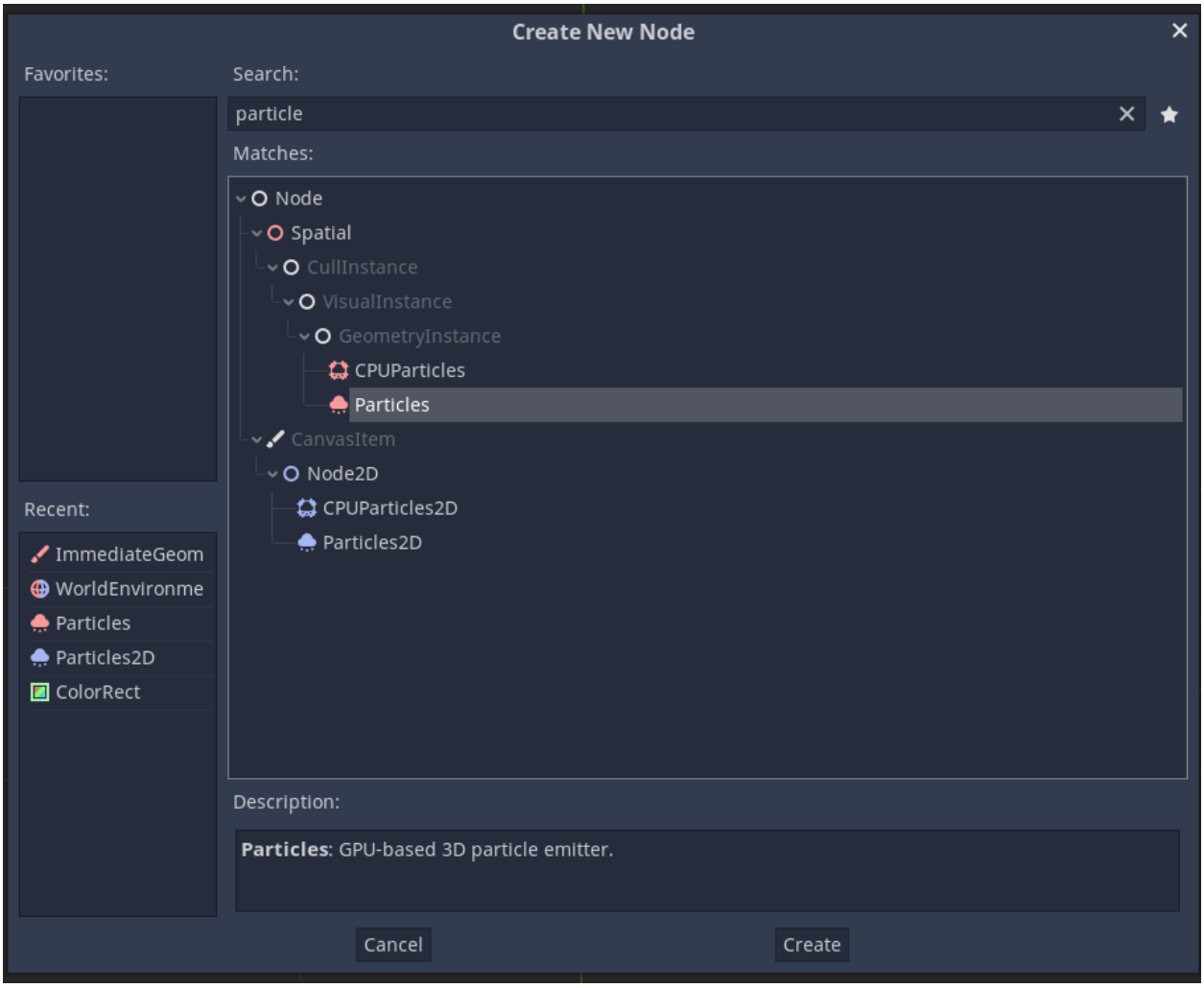
There's a lot more to explore in Godot, but for now, this is all you need to know about the basic anatomy of the project.

3.1 Particle Systems

Godot has a system for creating particle effects, which we can easily use in Foxus to great effect.

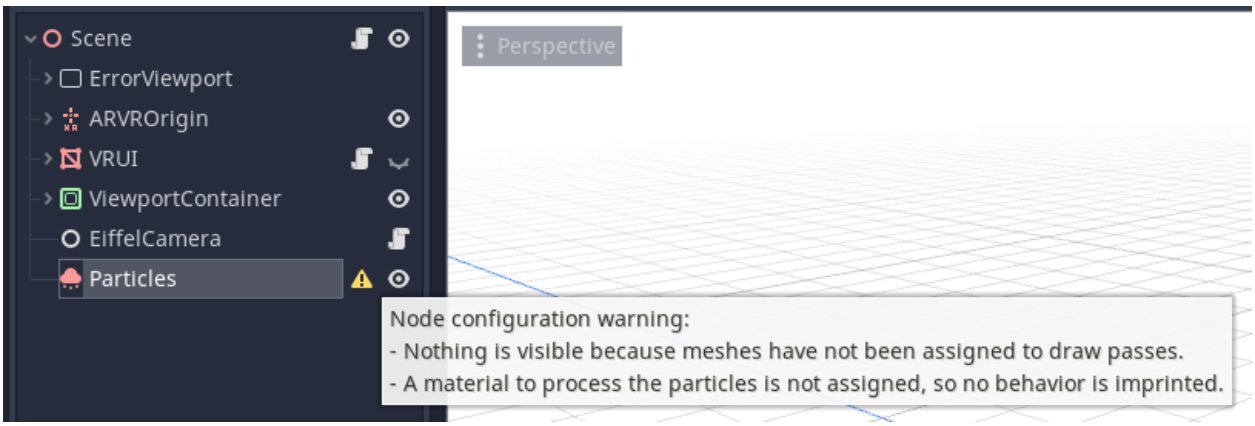
After you open the project, check the left side of the screen to see the Scene hierarchy. Directly below the tab that reads "Scene" is a + button. You can click this, or hit CTRL-A, to create a new node within your scene.

You'll be given heaps of options in the new window that appears, but just type in "Particles" and it will give you fewer to choose from.



A filtered view of the New Node window.

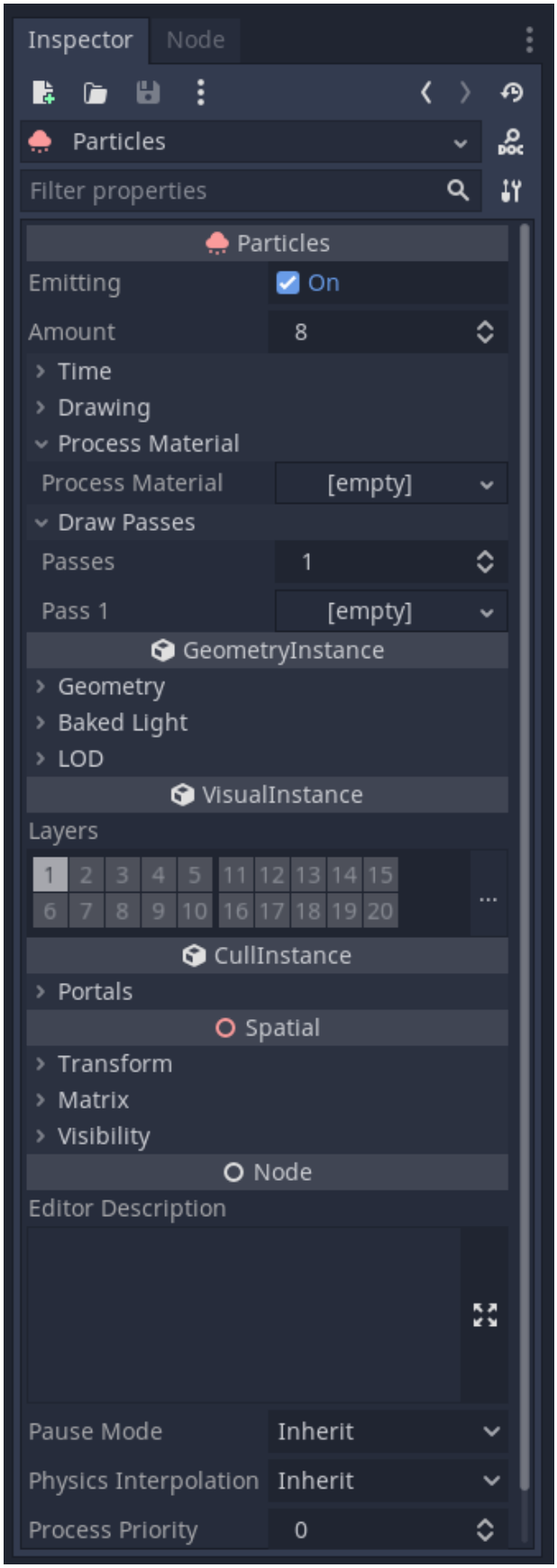
CPUParticles and the regular Particles type (which is rendered with the GPU) will have different performance on different devices. For now, let's go with the regular one. You can hit the "Create" button to add it to your scene. You'll now be able to see it in the Scene hierarchy to the left, with some warnings attached.



Check warnings by hovering your mouse over the yellow alert symbol.

If you click on this node, more information will open up on the right side of the screen. Use the tabs at the top of

this column to change from the "Node" view to the "Inspector" view. You may have noticed our warnings referred to missing *draw pass* meshes and *process materials*, so we'll be adding them here. You can click on these headings in the Inspector to expand them.


Inspector
Node
Particles
Filter properties
Particles
Emitting
On
Amount
8
Time
Drawing
Process Material
Process Material
[empty]
Draw Passes
Passes
1
Pass 1
[empty]
GeometryInstance
Geometry
Baked Light
LOD
VisualInstance
Layers
1 2 3 4 5 11 12 13 14 15
6 7 8 9 10 16 17 18 19 20
CullInstance
Portals
Spatial
Transform
Matrix
Visibility
Node
Editor Description
Pause Mode
Inherit
Physics Interpolation
Inherit
Process Priority
0

There's nothing there, but we can fix that.

By clicking on the entries reading [empty] we can assign new materials and meshes to them. In the Process Material heading, I've selected "New ParticlesMaterial", and in the Draw Passes heading I've selected "New SphereMesh" – what happens next?

The particle emitter is now pushing out endless Sphere Meshes.

If you're looking to create spheres or other simple shapes, this might be good enough. Right now, the spheres are only being fired downwards, like a machine gun. In the new Particle Material we've created (you may have to click on it to expand and edit it), open the heading for "Emission Shape." By default, this is set to a single point, which is why our spheres are all appearing at the same location. We can change the Emission Shape to a sphere, too, and increase the radius, to give a more organic falling particle effect.

Of course, you can experiment with other emission shapes and their sizes.

Let's say I want to create some particles inspired by the bokeh effect – gently falling balls of semi-transparent light. There's a few things I'll want to change here.

First, my bokeh are falling too quickly. If I open the "Gravity" sub-heading, I can see that by default a force of -9.8 is being applied to these particles along the green, vertical y axis. Changing this value to -1 makes them fall slower (and positive values would reverse their gravity).

It would be nice if the bokeh orbs were a little smaller, and if they weren't all exactly the same size. The "Scale" subheading lets me change both these values. I'll change "Scale" to 0.5, and "Scale Random" to 1, to give a lot of variation.

We're getting closer to something pretty.

I'm satisfied with the way my particles are being emitted now. I'd like to work on the particles themselves instead. If I click on my new sphere mesh inside of Draw Passes, I can expand it and edit its qualities. I need to give it its own material to change things like its color and translucency. You'll see a section that says Material is currently [empty], so click on it and select "New SpatialMaterial" so we can change some of its values.

Since I'm making bokeh, I already know a few things about how I want this new material to behave. Open the "Flags" heading. Let's check "Transparent" (so that it can render transparency on our bokeh orbs) and "Unshaded" (since they're balls of light, we don't want them to receive any sort of shading themselves). Next, open the "Albedo" heading and we can directly change its color here. I'll go for a nice, warm orange. I'll also use the "Alpha" slider to make it a bit transparent.

If you're familiar with hexadecimal color codes, this orange is #fab000 – isn't that fabulous?

There's one more really important thing we have to do to make sure our bokeh will show up nicely in the Foxus view... position the particle emitter!

Minimize the Process Material and Draw Passes headings that we've been working on. We can position the particle emitter using the "Transform" heading inside of the "Spatial" property, a little farther down. To position it more in the eyeline of a Foxus user, I've set the y value to 3.5 and the z value to -5.

If we just leave our particle emitter here, it looks pretty good in the Foxus view. However, you'll notice that it is anchored to the *scene*, which means that when we turn our head, it doesn't move with us.

If we want the bokeh particles to move with the camera, on the other hand, we can anchor this to our "head" by dragging the Particles node (in the scene hierarchy) into the ARVR origin node, and then inside of the ARVRCamera node.

For many types of particles, using a primitive shape like a sphere or cube isn't going to cut it. While you can certainly create more complex mesh shapes and use them as particles, a better solution might be to use 2D images like sprites.

To accomplish this, you can edit your existing particles, or make a new one. I've used the same settings for the Process Material section, meaning that my particles are gently falling in a sphere shape. However, instead of using a SphereMesh under Draw Passes, I've selected a QuadMesh, which is a flat quadrilateral. The new SpatialMaterial for this quad needs to have transparency checked, and to be unshaded, just like our last one. I also recommend enabling "Billboard Mode" under the "Parameters" heading. This means that the particles will always face the viewer.

Finally, under Albedo, instead of selecting a color, I've just loaded in the icon.png in the Foxus project as its texture.

It's like Foxus icon snowflakes…

You can also use an animated texture instead of a still image, if you import some into the project. The easiest way to do this is to output separate frames of an animation as still images and to drag the whole folder into the Foxus project folder. Instead of loading in an image as the Albedo texture, select "New AnimatedTexture", and load your frames in individually. You can also define a number of frames and a frames-per-second rate here.

Short animations are easy to load in frame-by-frame.

3.2 Importing 3D Models

Note: Here be dragons!

Positioning of 3D models in a mixed reality space is not an exact science.

Just placing an object in Godot will not anchor it to a "real" position in space.

Moving your head, or moving within the space, may move 3D objects in unexpected ways.

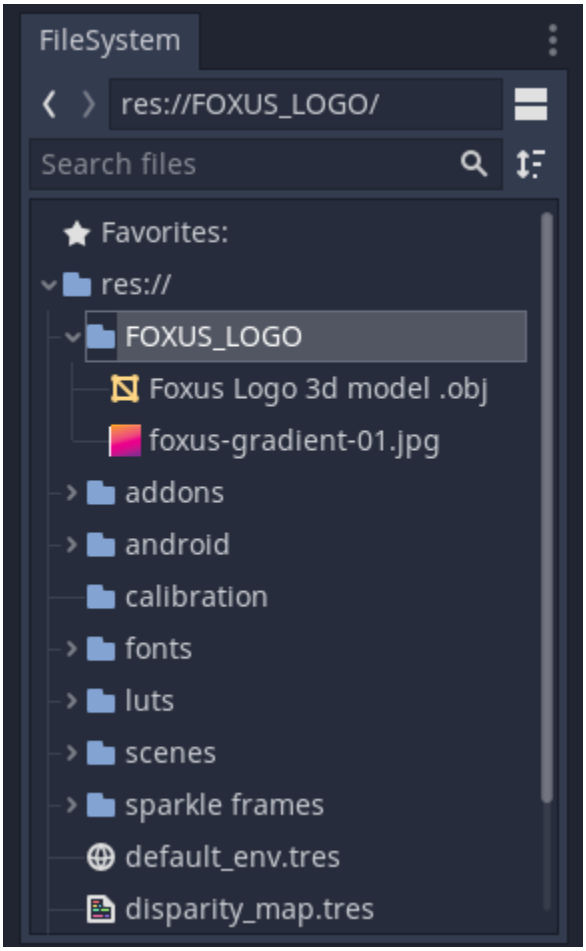
This problem could likely be solved with the use of pose estimation tools like ArUco markers.

If you've made a 3D model in an external program, or downloaded one you can use, you can probably import it into the Foxus view. Check the Godot documentation to find a list of accepted formats. For now, since we're keeping things as simple as possible, we'll be using a .obj file of the Foxus logo, with a simple gradient as its texture. You can download this file, and the texture it uses as an image, below.

`Foxus Logo 3D Model.obj`

`foxus-gradient-01.jpg`

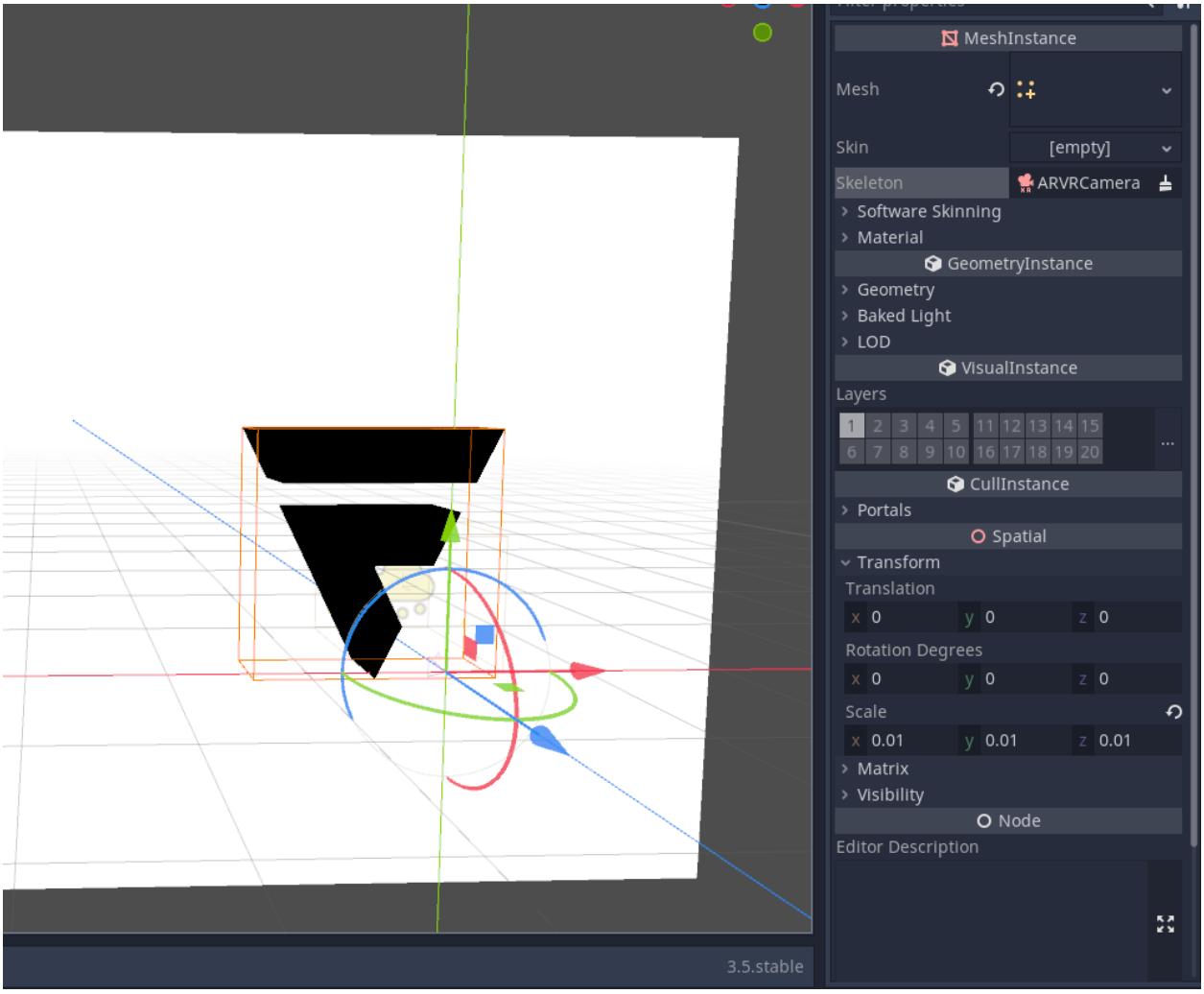
You can drag these directly into the Foxus project folder, or tuck them inside of a sub-folder to keep things more organized. Either way, they will be visible inside of the FileSystem tab inside of Godot:



Not all file types are visible in this view, but these are.

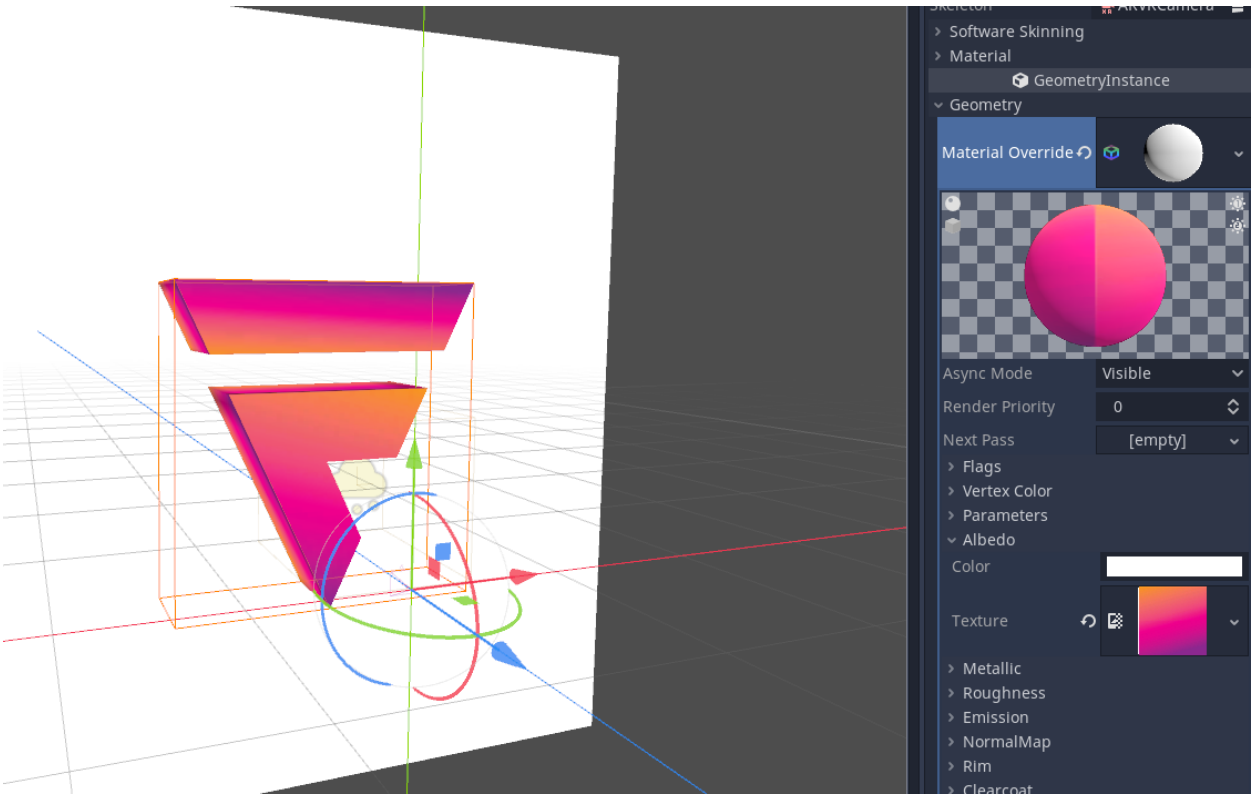
Dragging our new .obj file into the 3D Scene View in the center of the screen, or into the Scene hierarchy in the top left, will add it to the scene. However, you won't immediately be able to see it. This model is very large, so we'll need to reduce its scale.

Find the *Foxus Logo 3d model* object in the Scene hierarchy and click on it. You can adjust various properties for it on the right-hand side of the screen, under the Inspector tab.



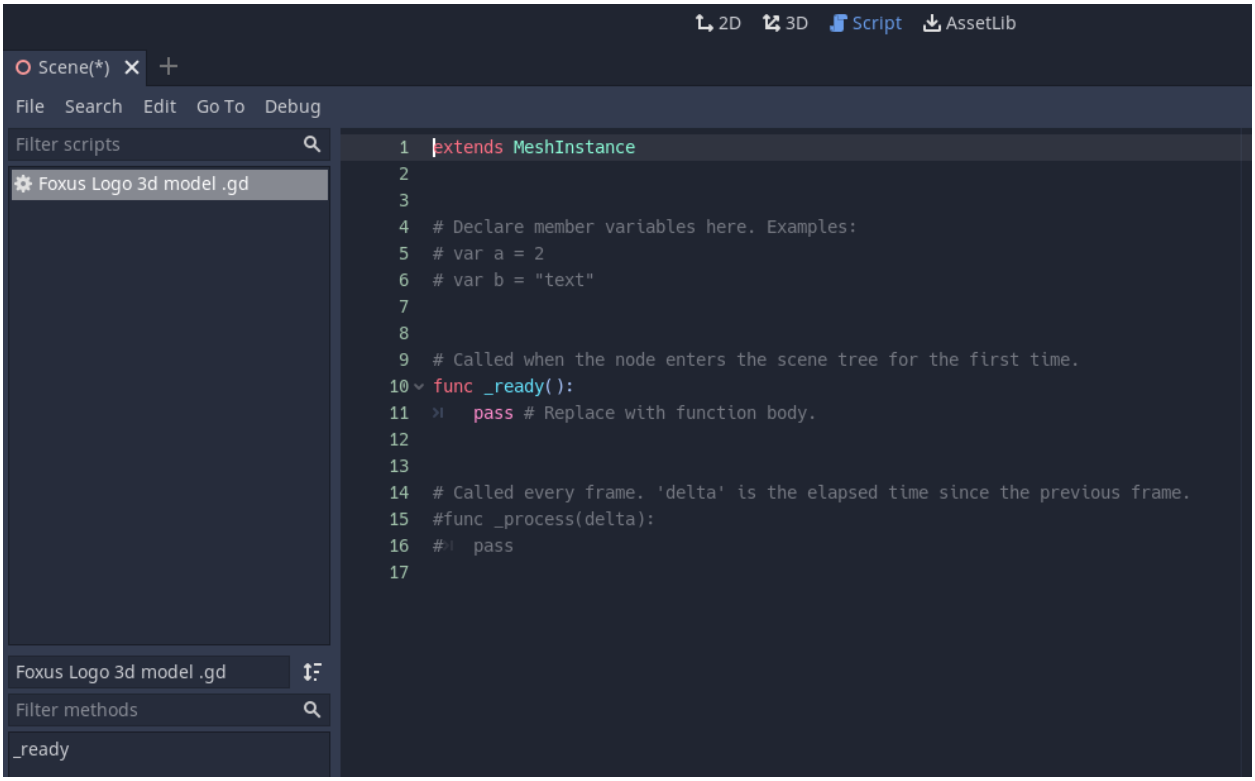
I've set the model's scale along all three axes to 0.01 here.

This model has heaps of possible surfaces you can apply textures to, but let's apply the gradient to all of them indiscriminately using the Material Override option, under "GeometryInstance".



After that, it's as easy as loading in our gradient to the Albedo texture of this material.

This looks neat, but because of the way 3D objects are anchored in the Quest's viewport, we won't get to view it from all angles like this. To compensate for this, let's add a simple script to this logo that makes it spin around by itself. Right-click the *Foxus Logo 3d model* object in the Scene hierarchy and select *Attach script*. You can leave all of the values to default in the small window that appears and create your new script.



If you've never looked at the Script view before, it may be intimidating, but we won't be doing anything frightening.

We'll be replacing this dummy script with a very simple one.

```
extends MeshInstance

# This is the rotation speed of our object, in degrees per second.
# 360 degrees is a full revolution each second. -360 spins the other direction.
var speed = 10

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
    # rotate_object_local rotates the object in local space.
    # The first argument we pass here (that strange Vector3) is just telling it
    # to rotate along the Y axis.
    # Then, multiply the time elapsed since the last frame, with our speed,
    # which we turn into "radians" before applying them to the rotation.
    rotate_object_local(Vector3(0, 1, 0),delta * deg2rad(speed))
```

Finally, as with our particle emitter tutorial, we should position this logo somewhere in space where we can definitely see it in the headset. I've dragged it inside of the *ARVRCamera* node, inside of *ARVROrigin*, so that it moves with my head. Then, in the same Spatial section where I scaled the model, I've given it a Transform position of *5, -5, -30*.