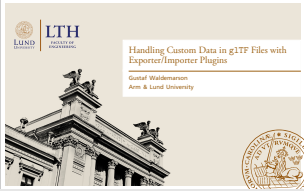
LUND UNIVERSITY | LTH FACULTY OF ENGINEERING

Handling Custom Data in g1TF Files with Exporter/Importer Plugins

Gustaf Waldemarson
Arm & Lund University

2025-01-11



Ah, I think we are just about ready to start!

Thanks everyone for coming! (TBD: And it's nice to see so many people here). I should warn you all though, this is a mostly technical, or rather, three technical topics. As such, I'm afraid that the supply of neat renders and images may run a bit sparse at time, but I hope you will all learn at least something from all this!

Who am I?

- Industrial PhD Student at the Lund University Graphics Group
- Software Engineer at Arm

- https://gustafwaldemarson.com

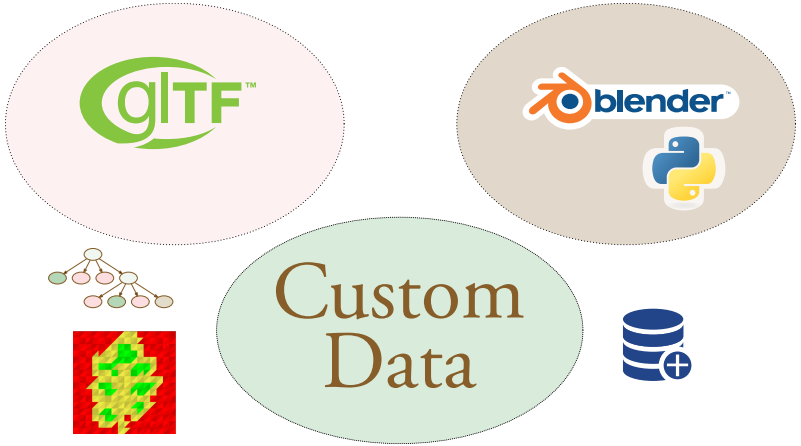
Who am I?
- Industrial PhD Student at the Lund University Graphics Group
- Software Engineer at Arm
- https://gustafwaldemarson.com
1/57

But first, I think some kind of introduction is in order: My name is Gustaf Waldemarson, a software engineer at Arm, where I have been helping out with various bits and bobs of the Mali driver stack, (especially the parts related to ray-tracing).

And to top it all off, I'm also a so-called Industrial PhD student at Lund University in Sweden where I also work with various ray-tracing related topics, some of which will make a small appearance later in this talk.

All that said, I am obliged to say that I do not represent Arm during this event, nor is the content is sponsored by Arm, and any views or opinions herein are entirely my own.

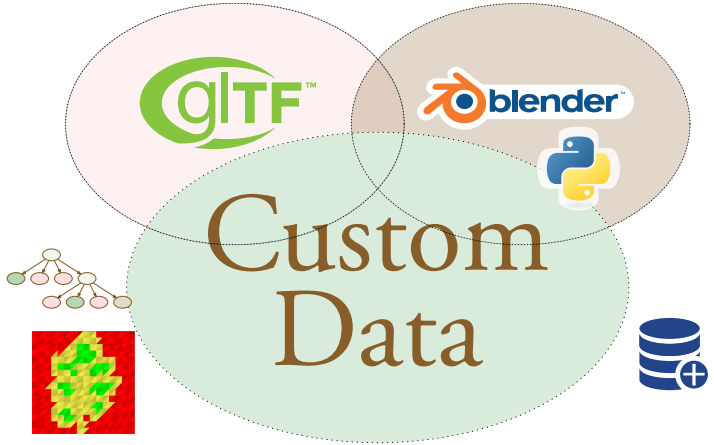
What is all this?

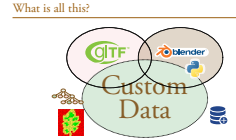


So, let's actually get into the meat of things: As I hope you can imagine, the topic I will discuss is about managing custom data with the `glTF` and Blender. As such, I imagine the audience to be something like this:

On one side, we have people who know and work with `glTF`, on the other, we have Blender developers and plugin-writers and lastly, some people who have some kind of data that isn't natively supported in there. Which could be anything from some tree data, some funky texture or data stored in a database somewhere.

What is all this?





My goal today is not to make you an expert on all these topics, but what I want is to slowly pull these groups together, just a little bit, such that there is a little bit of overlap. And thus make the work for anyone that want to do something similar to the things I have done just a bit easier.

And of course, I hope to do that with a few (hopefully) convincing examples or use-cases.

Agenda

- glTF?
  - What is it?
  - Data-models
- The glTF Importer/Exporter Addon
  - Blender
  - tinyglTF
- Extension Mechanism
- Example Plugin: Watermarking
- Customization Functions / Hooks
- Micromap Generator Plugin
- Importers

Agenda

- glTF?
  - What is it?
  - Data-models
- The glTF Importer/Exporter Addon
  - Blender
  - tinyglTF
- Extension Mechanism
- Example Plugin: Watermarking
- Customization Functions / Hooks
- Micromap Generator Plugin
- Importers

4/57

With that, we have a few different areas to cover! First, I will naturally start by talking a bit about glTF itself; what it is and what kind of data it stores and so forth.

Second, I'll briefly go over how Blender works with glTF-files, and talk a bit about the *Addon* that drives that process, as well as discussing at least one other tool you might come across for working with such files, such as when you are pulling in the data into your own rendering frameworks.

After that, we get to the custom things themselves. Here, I first, need to present the *Extension* mechanism that makes most of this possible, after which I'll show in detail how to construct a simple example plugin that simply performs a bit of watermarking on each image that is exported.

(After that I'll also show some other functions, or hooks as they called that we can modify and how these are generally structured.)

I'll then move on to my primary use of these facilities: Namely generating a thing known as micromaps during the glTF export process. And finally, I'll wrap things up by saying a few words about the *Importer* side of glTF as well.

What is glTF?

What is glTF?

5/57

So where should we start? Well, for completeness, we really should start talking about what glTF is.

In short, it is a royalty free format for transmitting (and loading) 3D scenes and models, while keeping the data as small as possible, both while in-memory and when stored on disk. Additionally, and most interesting to us, it is very extensible, which is what makes everything else I will be talking about possible.

That said, glTF is not a universal 3D modeling format. The base primitives it supports are distinctly geared towards real-time rendering use-cases. In fact, many of the constants used in the format map directly to the equivalent API values in OpenGL, all to make it as seamless as possible to load all data and send it directly to the GPU. But, the format is of course not limited to that. As an example, the material model is very general, and with the aforementioned extension system, it is easily extended.

I should mention that I will not go into detail about how to author glTF scenes, but will instead focus on some of the technical aspects we need to know about. That said, Julien Duroure, one of the primary authors of the glTF importer/exporter addon, that I've literally built everything on-top-of, will have a session after this in the class-room venue that will deal with this topic in more detail.

Types of glTF Files

- GLTF_EMBEDDED (.gltf)
- GLTF_SEPARABLE (.gltf, .bin + textures)
- GLTF_BINARY (.glb)

Types of glTF Files
• GLTF_EMBEDDED (.gltf)
• GLTF_SEPARABLE (.gltf, .bin + textures)
• GLTF_BINARY (.glb)
6/57

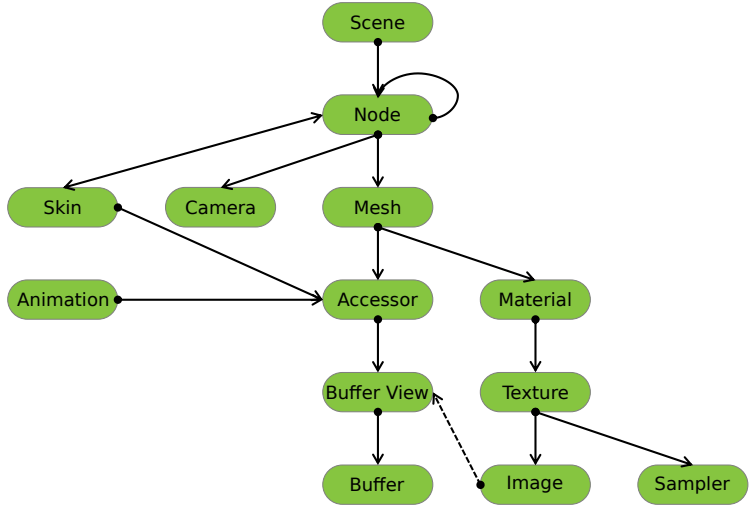
Starting with the files themselves: There are essentially three versions of glTF, which in Blender is exposed as GLTF_EMBEDDED, SEPARABLE or BINARY.

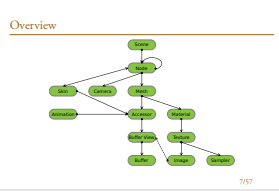
The first two are essentially two versions of a JSON file and only really differs in how some of the data is *packed*.

- For the embedded format, *all* data (even images) are packed in the same file as base-64 encoded strings.
  - This makes it easy to transport and share,
  - But harder to re-work or re-author.
- For the other, the data is instead packed in *separate* files next to the primary .gltf-file.
  - Easier to modify some data after the fact.

As for the last format, it is an entirely binary format (and the most compact version), that I like to think of as a binary equivalent to JSON, but of course, there are more nuances to it than that, but that is nothing we will get into here.

Overview



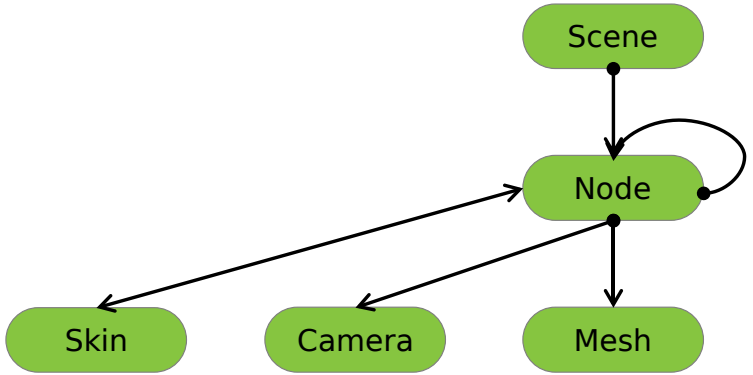


So, let us take a look at the insides of a `glTF`-file instead shall we? In brief, the entire file is actually really well encapsulated with a single relation diagram:

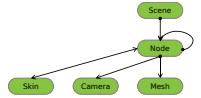
Which lists *almost* all types of objects that exists in the `glTF`-world.

And while this looks a bit complicated, if we pull it apart, we can actually identify some key components that we may care a bit more about.

Scene Graph



Scene Graph



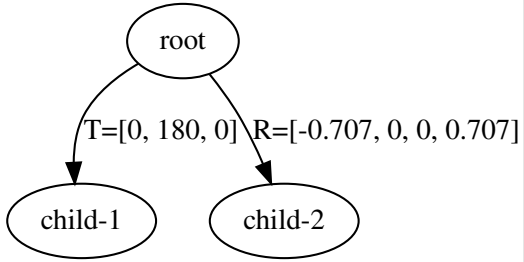
8/57

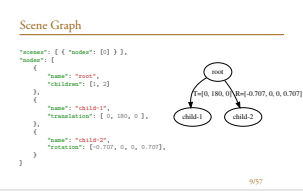
This part defines the scene graph(s) of the file.

And interestingly, it *also* contains the bone hierarchy for skinned animations, but that is another side note.

Scene Graph

```
"scenes": [ { "nodes": [0] } ],
"nodes": [
    {
        "name": "root",
        "children": [1, 2]
    },
    {
        "name": "child-1",
        "translation": [ 0, 180, 0 ],
    },
    {
        "name": "child-2",
        "rotation": [-0.707, 0, 0, 0.707],
    }
]
```



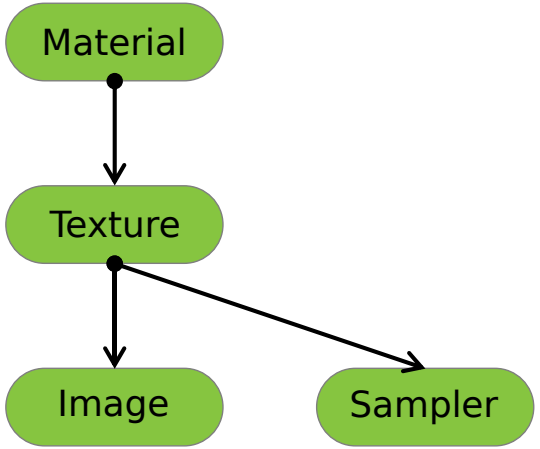


The way these are structured, is that each scene in a file contains an array of node indices that collectively create a graph (or more accurately, a collection of trees).

Such as in this trivial example: Where we only have a single scene with a root node with two child nodes, each with a different associated transform: One with a rotation, and the other with a translation.

And this use of indices is a recurring theme in `glTF`: To keep everything as compact as possible, objects are almost always stored in linear arrays which are then cross-reference with indices when necessary.

Shading Model

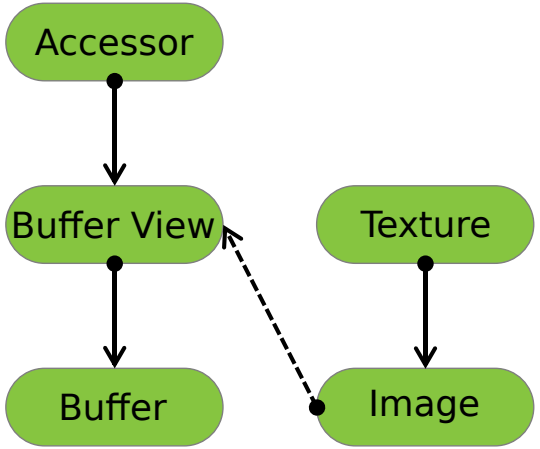


Shading Model
Material
Texture
Image
Sampler
10/57

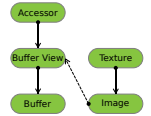
Moving on: This next part define the shading model used by the objects in the scenes.

By default, a Physically Based shading model that is called the “metallic-roughness” model is used, which essentially defines up to 3 textures that we may care about (a color, metallic-ness factor and roughness factor). Again, there are more details, but we won’t really have time to go into them.

Data Storage



Data Storage



11/57

And lastly, this part define the data acquisition model. And, as you can imagine from the topic of this talk, this part is pretty important for us!

Buffers, Views and Accessors

`Accessors`

- What to access
- How to access
- `BufferView`

`Texture`

- `Image`

`Image`

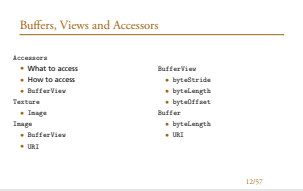
- `BufferView`
- `URI`

`BufferView`

- `byteStride`
- `byteLength`
- `byteOffset`

`Buffer`

- `byteLength`
- `URI`



So, most of the work we want to do in this talk really just boils down to using some data that you already have, and packing it into something that `glTF` understands. And, beyond using `JSON` primitives, these are our basic building blocks to do so, which we can break down into a few different levels:

At the lowest level we find the `Buffer`, which really is just a source of unstructured bytes. Just above that is a `BufferView`, which really is just a bit of metadata on *how* to index a `Buffer` using an offset, stride and length.

Above *that* we'll find the `Accessors`, `Textures` and `Images` which describes exactly what kind of data we want to access and how it should be accessed through e.g., a *BufferView*.

Buffers, Views and Accessors

Inside Blender…

- `Accessor` ⇒ `Accessor`
- `Texture` ⇒ `Texture`
- `BufferView` ⇒ `BinaryData`
- `Buffer` ⇒ `BinaryData`
- `Image` ⇒ `ImageData`

https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html
https://www.khronos.org/files/gltf20-reference-guide.pdf

Buffers, Views and Accessors
Inside Blender…
• Accessor ⇒ Accessor
• Texture ⇒ Texture
• BufferView ⇒ BinaryData
• Buffer ⇒ BinaryData
• Image ⇒ ImageData
https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html
https://www.khronos.org/files/gltf20-reference-guide.pdf
13/57

So what becomes of all of this when it reaches Blender?

This is where the `glTF` importer and exporter addon comes in. And the way it is structured in a really straight-forward way: Each object that exist in `glTF` has a corresponding class associated with it. And any entity that was cross-referenced with indices gets resolved into a proper object-reference, all to make it a lot easier to handle the data.

So thanks to Julien and the other contributors of the `glTF` addon, we can actually ignore almost all of these low level details as they are actually handled automatically for us. Even the really low level buffer and image objects gets converted the so called `BinaryData` and `ImageData` classes to make it easier to manage this type of data, something we will get into a bit more detail of a bit later.

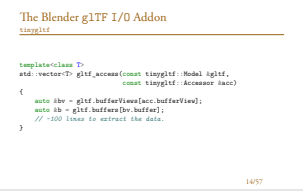
So, if you are working with the format, keeping the `glTF` specification and object reference sheet (that I have linked down here) around is very helpful to figure out exactly what data each object *actually* contains.

In summary: These classes acts as both a data-container (if you need the data) and a wrapper for reading and writing to the various types of `glTF` files.

The Blender glTF I/O Addon

tinygltf

```
template<class T>
std::vector<T> gltf_access(const tinygltf::Model &gltf,
                           const tinygltf::Accessor &acc)
{
    auto &bv = gltf.bufferViews[acc.bufferView];
    auto &b = gltf.buffers[bv.buffer];
    // ~100 lines to extract the data.
}
```

However, if you are on the receiving end of the glTF-file, you may actually care about those details. E.g., if you are exporting files from Blender using glTF, and then using the tinygltf C++ library to load the file. In that case you only get a direct struct-based representation of the glTF-file: It is up to you to actually read back the data if you want it, including the index-resolution I mentioned earlier. Which, as in this truncated snippet, can require quite a bit of extra code; which in my case, takes about another 100 lines of code.

(But, inside Blender most of this is managed for us, however, in our engines, or in my case, my research renderer, we usually don't have Blender directly available. For those cases, I recommend using tools such as tinygltf to do most of the heavy lifting for us.)

(While bare-bone, it does an excellent job of making the glTF data available to C++ applications.)

The Blender glTF I/O Addon

Blender

```python
import io_scene_gltf2.io.exp.gltf2_io_binary_data as exp
import io_scene_gltf2.io.imp.gltf2_io_binary_data as imp
indices = [0, 1, 2]
binary_data = exp.BinaryData(bytes(indices))
decoded = imp.BinaryData.decode_accessor_internal(binary_data)
```


The Blender glTF I/O Addon
Blender
import io_scene_gltf2.io.exp.gltf2_io_binary_data as exp
import io_scene_gltf2.io.imp.gltf2_io_binary_data as imp
indices = [0, 1, 2]
binary_data = exp.BinaryData(bytes(indices))
decoded = imp.BinaryData.decode_accessor_internal(binary_data)
15/57

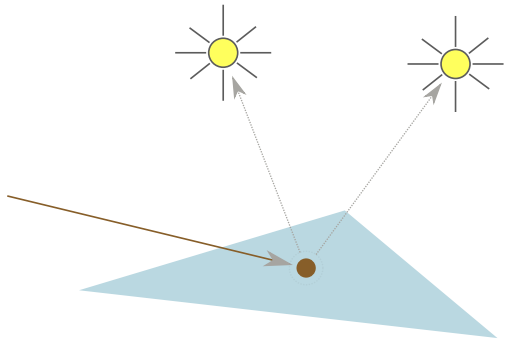
And if we compare this with the facilities available in Blender through the `glTF` addon, everything is just much nicer in comparison:

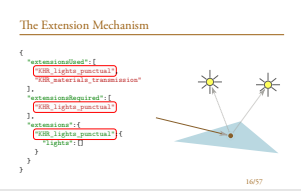
- To write a buffer (and at the same time, create a `BufferView`), we simply construct a `BinaryData` class.
- And to read that data back, we simply call a static method in the same class called `decode_accessor_internal`.

But, depending on our contexts we don't always have it this easy!

The Extension Mechanism

```
{
  "extensionsUsed":[
    "KHR_lights_punctual",
    "KHR_materials_transmission"
  ],
  "extensionsRequired":[
    "KHR_lights_punctual"
  ],
  "extensions":{
    "KHR_lights_punctual":{
      "lights":[]
    }
  }
}
```





The final thing I want to mention about `glTF` is about the *Extension* mechanism. As I mentioned earlier, the `.gltf` file is really just a kind of `JSON` file. As such, we can *technically* add whatever data we want to it. Obviously, the specification requires some stuff, but it also gives us an *official* way for adding new attributes in a structured fashion.

This is done by adding a new key-object pair under a special *extension*-object. The key in this case is the vendor prefix: Basically the letters `KHR`, for *Khronos* in this case, followed by a descriptive title. In this case, `lights_punctual` which allows us to embed various point-like light-source in the scene. For this talk though, everything is obviously assumed to be *custom*, so where applicable I will just use the prefix `NONE`.

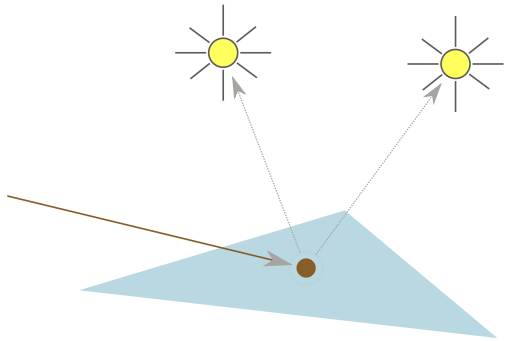
And these extensions essentially come in two flavors: *required* and *optional*. As the names suggest, if an extension is required, a *compliant* `glTF`-parser/renderer should abort if it does not support the extension. E.g., Meshes that are compressed with an unsupported algorithm, naturally cannot be handled.

Optional ones though could be things such metadata containers, or optional rendering features. I.e., if the client does not support it, it should still be able to use all other data, although renderings may be different from what the author intended.

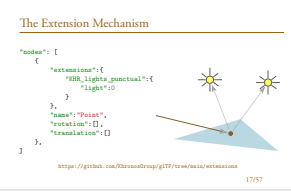
Extensions that are used are then listed in the top of the `JSON` file, like in these examples here.

The Extension Mechanism

```
"nodes": [
    {
        "extensions":{
            "KHR_lights_punctual":{
                "light":0
            }
        },
        "name":"Point",
        "rotation":[],
        "translation":[]
    },
]
```



https://github.com/KhronosGroup/glTF/tree/main/extensions



Then, depending on the actual extension itself, more *extension*-keys can be found elsewhere in the file, such as this extension here which defines a point-light attached to the scene-graph. Where and how these keys are used is entirely up to the extension itself.

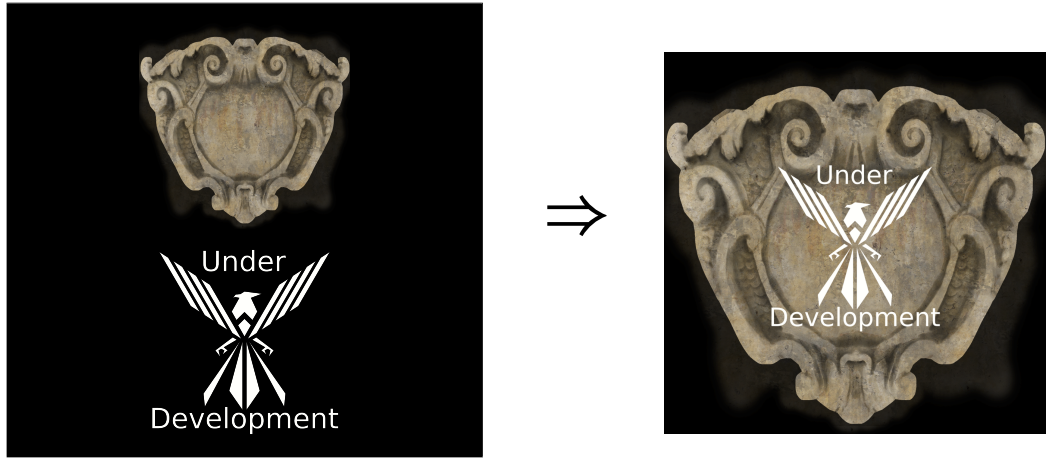
Extensions are sometimes useful for many people out there though, as such they are occasionally *standardized*, and a collections of these can be found here:

- https://github.com/KhronosGroup/glTF/tree/main/extensions

Creating of 'official' extensions in this way is well beyond the scope of this talk, but this links provides details for the interested out there.

Texture Watermarking

A First Use-Case



LUND UNIVERSITY

And then we finally we get to the actual Plugins themselves. A quick disclaimer before go on though: I do not claim to be a good Blender plugin writer. In fact, I'm pretty sure there are much better ones in this very room.

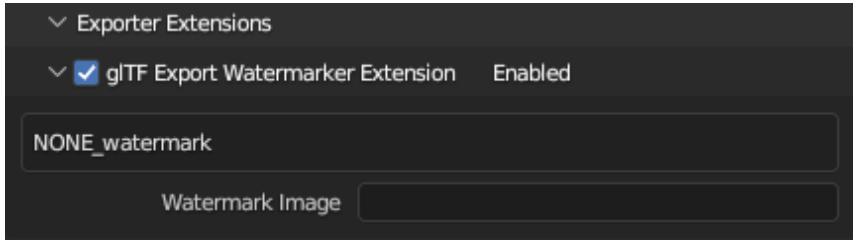
But with that, I want to present a first example of these tools in action: A plugin that watermarks all textures while exporting.

This is admittedly a bit of a facetious use-case, but it is a good, simple example to show off how to create a `glTF` plugin.

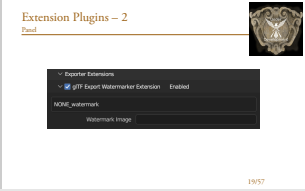
So with that, let us go over the basics that you need to create a new plugin for the `glTF` exporter.

Extension Plugins – 2

Panel



Extension Plugins – 2
Panel



So, let us start with a clear goal for this plugin:

- We know what we want the watermarker to do, we already have a nice visual cue for it up here: We want this to happen to *all* images we export.
- But obviously, we want to also be able to control the process in some fashion.
- To that end, we want to add a panel to the GLTF exporter that looks something like this:
  - Basically, we want the ability to enable or disable the plugin, and,
  - Select what image to use as a watermark.

Extension Plugins – 2

bl_info


```
bl_info = {
    "name": "glTF Export Watermarker Extension",
    "category": "Import-Export",
    "version": (1, 0, 0),
    "blender": (3, 0, 0),
    "location": "File > Export > glTF 2.0",
    "description": "Watermark any exported image texture.",
    "author": "Gustaf Waldemarson",
}
```

https://wiki.blender.org/wiki/Process/Addons/Guidelines/metainfo

Extension Plugins – 2
bl_info

But let us start from the top: First we need a special variable called `bl_info` that Blender can use to pick up metadata about the plugin, such as a name, description, current version, which Blender versions it supports and so on.

This is not unique to our type of plugin though, but is really used by all Blender addons. And after a bit of digging, I even found a link to the specification for this variable, so here you can see all options that you can (or should) add to it.

Extension Plugins – 3

Properties


```python
class WatermarkingExtensionProperties(bpy.types.PropertyGroup):

    enabled: bpy.props.BoolProperty(
        name="glTF Export Watermarker Extension",
    )
    watermark: bpy.props.StringProperty(
        name="Watermark image",
    )
```


Next we get to the property class that we use to *contains* these control options, in short, we need:

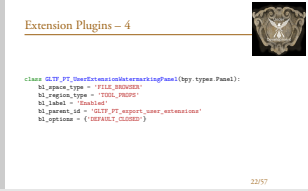
- A boolean property to enable or disable the plugin.
- A string to identify the image we want to use as a watermark.

This property class will then help also help us to create the actual buttons to change these options, which is exactly what we'll look into next…

Extension Plugins – 4


```
class GLTF_PT_UserExtensionWatermarkingPanel(bpy.types.Panel):
    bl_space_type = 'FILE_BROWSER'
    bl_region_type = 'TOOL_PROPS'
    bl_label = 'Enabled'
    bl_parent_id = 'GLTF_PT_export_user_extensions'
    bl_options = {'DEFAULT_CLOSED'}
```


Extension Plugins – 4



And to do that, we (typically) need another class to contain the *look-and-feel* of the controls.

This one uses a slew of variables and methods to define how to create buttons that set the properties, even where they should be created, and so forth. Of particular interest in this case is the `bl_parent_id` variable:

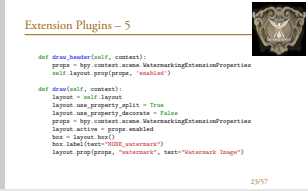
Here, it is set to the special key (`GLTF_PT_export_user_extension`) that signals that we are a child of the main `glTF` export panel.

Obviously, it could change this and place these options elsewhere, but this is a logical place for it: Right next to the other `glTF` export options.

Extension Plugins – 5


```python
def draw_header(self, context):
    props = bpy.context.scene.WatermarkingExtensionProperties
    self.layout.prop(props, 'enabled')

def draw(self, context):
    layout = self.layout
    layout.use_property_split = True
    layout.use_property_decorate = False
    props = bpy.context.scene.WatermarkingExtensionProperties
    layout.active = props.enabled
    box = layout.box()
    box.label(text="NONE_watermark")
    layout.prop(props, "watermark", text="Watermark Image")
```

Then, to actually create the buttons and allow them to set our options, we do a few things:

1. The `draw` method tells us how to well, *draw* our panel, hence, we should change the `layout` here to including any labels or decorators for the buttons that we want.
2. Then, to create buttons and connect it with our properties, we call the `prop` method here the property class in question and the *key* that identifies the option, such as the *watermark* in my case here.
3. And that is pretty much it: If done correctly, the property and panel classes will now cooperate to provide a simple menu for setting these values.

I should also note that it is strongly recommended that these two classes have unique names, otherwise, you can end up in a situation where only the last registered `glTF` plugin shows up in the menu.

Extension Plugins – 6

Registration


```python
def register():
    bpy.utils.register_class(WatermarkingExtensionProperties)
    prop = bpy.props.PointerProperty(type=WatermarkingExtensionProperties)
    bpy.types.Scene.WatermarkingExtensionProperties = prop


def register_panel():
    try:
        bpy.utils.register_class(GLTF_PT_UserExtensionWatermarkingPanel)
    except Exception:
        pass
    return unregister_panel
```


Next, we need to actually register these classes with Blender and store them somewhere in the scene context such that we can actually modify or retrieve them at a later time.

(Here we do run into a minor caveat: We must make sure to only register the panel once.

However, our plugin depend on the main `glTF` addon, hence, if that addon is not actually loaded, it we will fail to register the panel. While crude, the simplest way to handle this is simply to use `try-except` guards, even if these probably are a bit heavy for us.)

Extension Plugins – 7

Un-Registration


```
def unregister_panel():
    try:
        bpy.utils.unregister_class(GLTF_PT_UserExtensionWatermarkingPanel)
    except Exception:
        pass


def unregister():
    unregister_panel()
    bpy.utils.unregister_class(WatermarkingExtensionProperties)
    del bpy.types.Scene.WatermarkingExtensionProperties
```


Extension Plugins – 7
Un-Registration
25/57

And if we register the classes, we should probably also *un*-register them as well, with the same caveat as before.

Extension Plugins – 8

The Exporter Class


```
class glTF2ExportUserExtension:
  def __init__(self):
    from io_scene_gltf2.io.com.gltf2_io_extensions import Extension
    self.props = bpy.context.scene.WatermarkingExtensionProperties

  def gather_image_hook(self,
                        gltf2_image,
                        blender_shader_sockets,
                        export_settings):
    watermark(gltf2_image)
```


Extension Plugins – 8
The Exporter Class

class glTF2ExportUserExtension:
def __init__(self):
from io_scene_gltf2.io.com.gltf2_io_extensions import Extension
self.props = bpy.context.scene.WatermarkingExtensionProperties
def gather_image_hook(self,
gltf2_image,
blender_shader_sockets,
export_settings):
watermark(gltf2_image)
26/57

And now we finally get to the class that defines these `glTF` plugins. In your script, you should define a class with the exact name `glTF2ExportUserExtension` and add methods to it that corresponds to what we want to customize.

So in our case, we want to customize all images. Something that is done using the appropriately named `gather_image_hook`, where we simply receive the *gltf_image*, and the Blender equivalent.

(Again, we run into the small issue of dependencies: To avoid possible import errors when loading the plugins, we should place our `imports` such that they are only called when we *know* that the primary addon is present. Hence we have these `import` statements in somewhat funky locations.)

(Tentative, not sure I have time to go into this.)

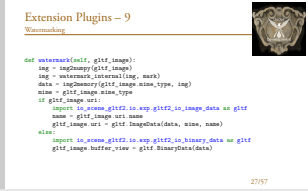
(And this is essentially the minimum that a `glTF` plugin needs to contain, however, as we will get into in a bit there are a few additions that I recommend that you add to make the code more robust, especially if you are targeting multiple Blender versions.)

(Alternatively, you can create a list called `glTF2ExportUserExtensions` (notice the *s*), which are all the classes you want to represent your plugins.)

Extension Plugins – 9

Watermarking


```python
def watermark(self, gltf_image):
    img = img2numpy(gltf_image)
    img = watermark_internal(img, mark)
    data = img2memory(gltf_image.mime_type, img)
    mime = gltf_image.mime_type
    if gltf_image.uri:
        import io_scene_gltf2.io.exp.gltf2_io_image_data as gltf
        name = gltf_image.uri.name
        gltf_image.uri = gltf.ImageData(data, mime, name)
    else:
        import io_scene_gltf2.io.exp.gltf2_io_binary_data as gltf
        gltf_image.buffer_view = gltf.BinaryData(data)
```

And for completeness, the function that performs the actual watermarking looks something like this:

1. First, we basically convert the encoded image to something we can use (which is a numpy array in this case).
2. Then, we apply the watermark,
3. And finally, we convert the image back to a binary format again.

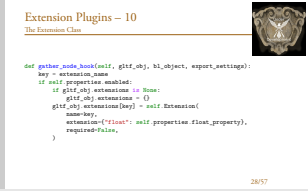
We do have to be slightly careful though: It is the user that decides what format the image is stored in depending on the embedded or separable time from before. Hence, a *good* plugin obviously should respect that choice, even if we could override it.

Thankfully, this is pretty easy to handle: We simply check the `uri` attribute of the original image, and then use either the `BinaryData` (for embedded formats) or `ImageData` class (for the separable one).

Extension Plugins – 10

The Extension Class


```
def gather_node_hook(self, gltf_obj, bl_object, export_settings):
    key = extension_name
    if self.properties.enabled:
        if gltf_obj.extensions is None:
            gltf_obj.extensions = {}
        gltf_obj.extensions[key] = self.Extension(
            name=key,
            extension={"float": self.properties.float_property},
            required=False,
        )
```

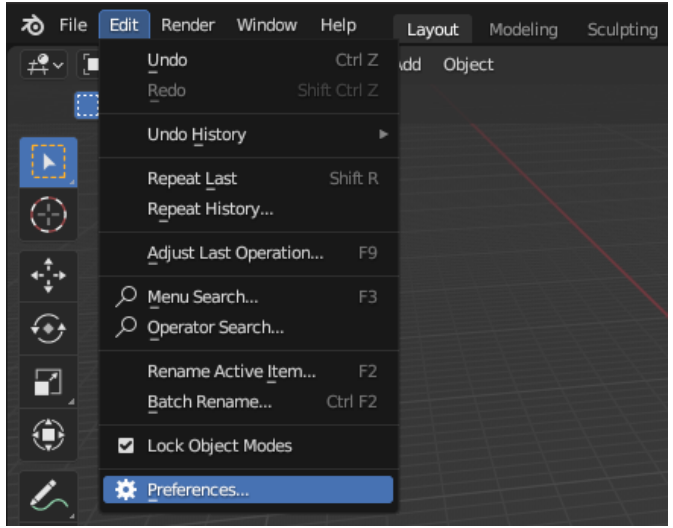



Lastly, I want to say a few words about the so-called `Extension` class, which I didn't actually have to use in this example. This is a special class used to signal that we are *adding* extension data of some kind so that the exporter can add it to the list of used Extensions. As I showed before though, we can place this class pretty much anywhere, such as in the node hierarchy in this example.

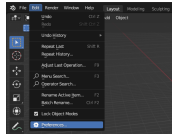
It can also contain anything that can become valid `JSON` data. And this includes any other `glTF` classes, even the special `BinaryData` classes I showed earlier.

(Which in those cases, the data is packed into *Buffers* and these links are replaced with *BufferViews*.)

Installing and Enabling Plugins



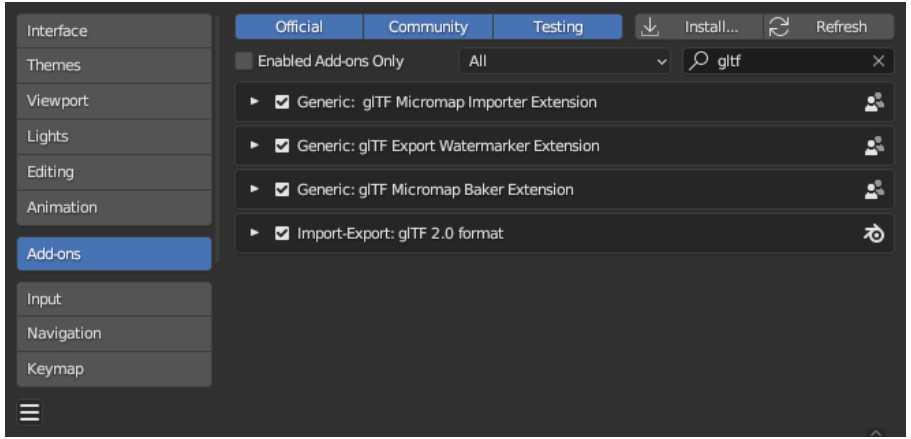
Installing and Enabling Plugins



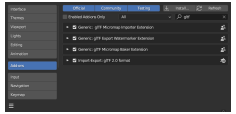
And now is a good point for this little reminder for completeness's sake: I typically install a new by dropping the file into the addon directory, but forget to enable it, and then sit around scratching my head for a bit wondering where all my stuff has gone.

So this is easily fixed done by opening the *Edit*-menu, and going to *Preferences* dialogue.

Installing and Enabling Plugins



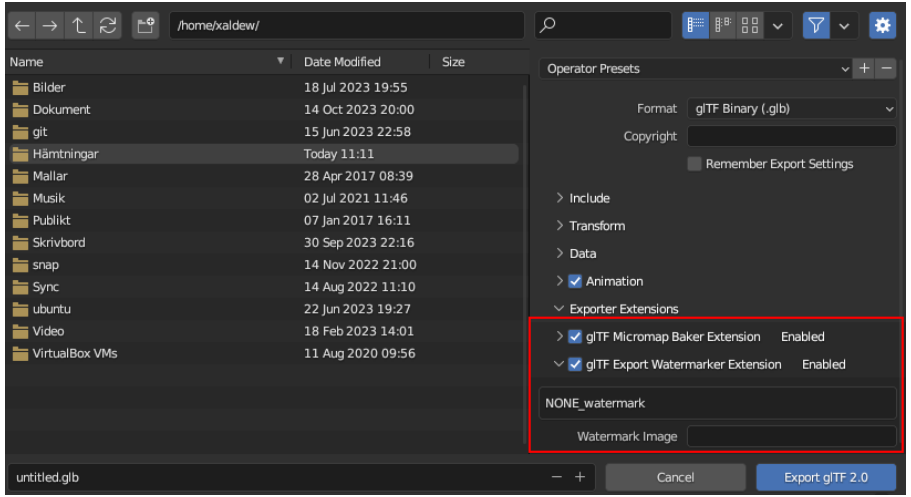
Installing and Enabling Plugins



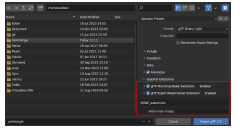
Here, we open the *Add-ons* section. Then I recommend enabling all sections and searching for "gltf". Then simply click *enable* to make the add-on usable.

I'm sure there's a way of doing this automatically, but it is a good thing to be able to find these settings, and you only have to do this once anyways.

Installing and Enabling Plugins



Installing and Enabling Plugins



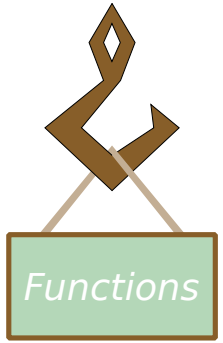
31/57

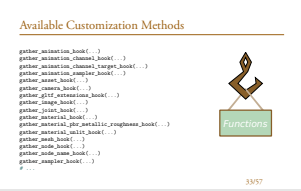
And with all that, if you have done everything correctly, you should have a new entry in the `glTF` export panel with your options.

Hooks

Available Customization Methods

```
gather_animation_hook(...)
gather_animation_channel_hook(...)
gather_animation_channel_target_hook(...)
gather_animation_sampler_hook(...)
gather_asset_hook(...)
gather_camera_hook(...)
gather_gltf_extensions_hook(...)
gather_image_hook(...)
gather_joint_hook(...)
gather_material_hook(...)
gather_material_pbr_metallic_roughness_hook(...)
gather_material_unlit_hook(...)
gather_mesh_hook(...)
gather_node_hook(...)
gather_node_name_hook(...)
gather_sampler_hook(...)
# ...
```





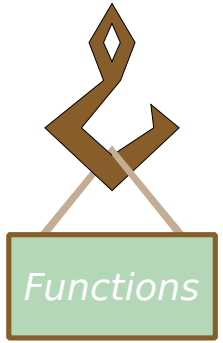
As most of you probably know, Blender has loads of primitives, and `glTF` supports many of these. And this support is exposed with a long list of methods as you can see *here*, all of which you can override. But in short: Each of these map to one of the objects that you can find in the `glTF`-world.

(And while I have worked with a good chunk of these types, I obviously cannot go into detail on all of these.)

Available Customization Methods

My Overrides

```
gather_image_hook(...)
gather_mesh_hook(...)
gather_primitive_hook(...)
```



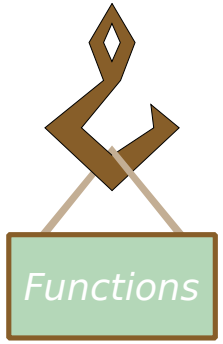
Available Customization Methods
My Overrides
gather_image_hook(...)
gather_mesh_hook(...)
gather_primitive_hook(...)
Functions
34/57

So, beyond the image hook we looked at for the watermarking plugin, We will only really talk about a few more method out of all of these: Namely the mesh and primitive-hooks. While this is pretty restrictive, I think that just the mesh alone is good example, such that it is pretty easy to infer what *could* be done in any of the other hooks.

Available Customization Methods

General Structure

```
gather_mesh_hook(self,
                 gltf2_mesh,
                 blender_mesh,
                 blender_object,
                 vertex_groups,
                 modifiers,
                 materials,
                 export_settings)
```



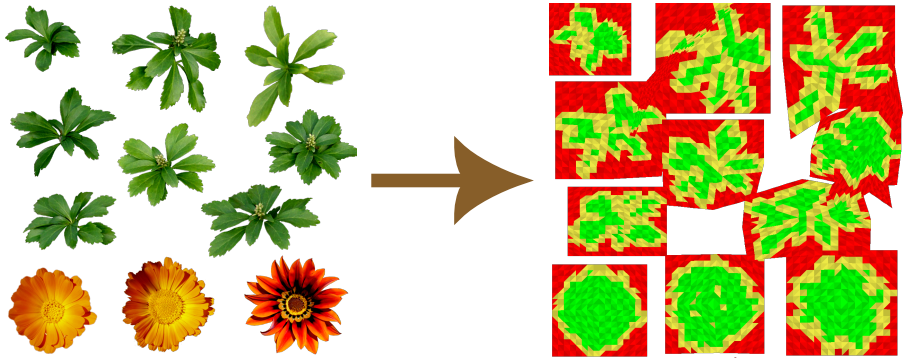
Available Customization Methods
General Structure
gather_mesh_hook(self, gltf2_mesh, blender_mesh, blender_object, vertex_groups, modifiers, materials, export_settings)
Functions
35/57

In general, I try to think of each of these methods like this: It's a method that receives both the Blender and `glTF` versions of the object that we are exporting or importing.

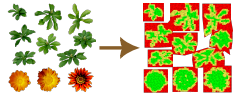
So, in the case of a mesh: We get both the Blender and `glTF` version, but also which object is belongs to, the vertex groups, modifiers and materials.

Finally, all methods also get this last argument which is a dictionary that contains global export options: Such as the output directory where any generated file should be placed, if we are exporting in `JSON` or binary mode, etc.

Micromaps – VK_EXT_opacity_micromap



Micromaps – VK_EXT_opacity_micromap



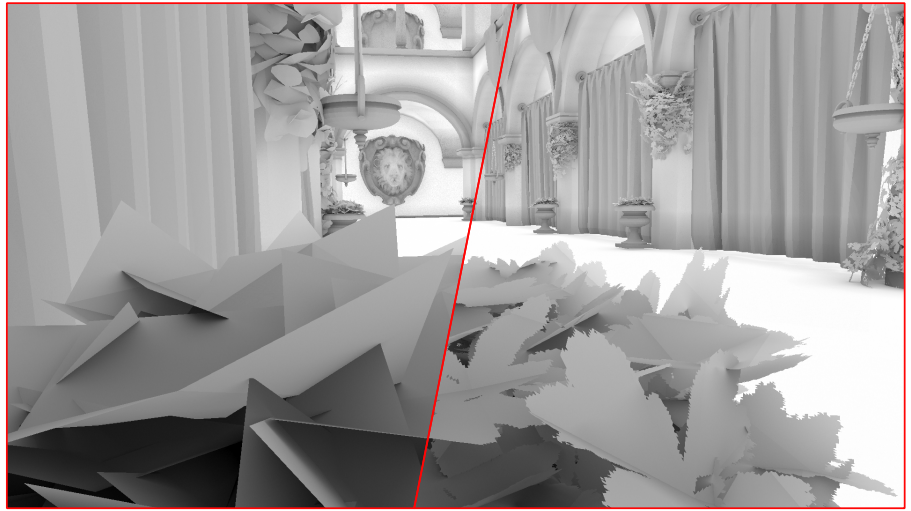
36/57

This moves us to my actual use case: As I mentioned, I use `glTF` as my primary 3D model format for my research renderer, as such, I often investigate various new techniques or types of data. However, creating new formats for each new type of data is really annoying. As such, I eventually realized that the `glTF` extension mechanism could be used to simplify this matter.

As such, allow me to present the thing that I have been working on: A relatively new type of rendering primitive in Vulkan® and DirectX® that is known as a *Micromap*.

Basically, we want to somehow generate this thing that you can see on the right.

Micromaps – VK_EXT_opacity_micromap



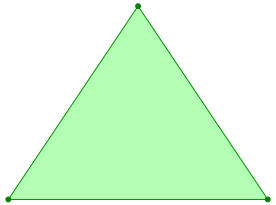
Micromaps – VK_EXT_opacity_micromap

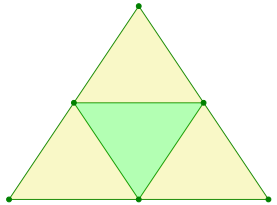
37/57

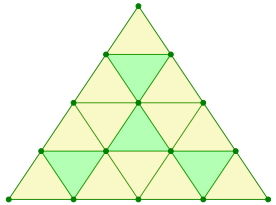
And once you apply these to a scene, such as our traditional Sponza scene, we can get an effect that looks something like *this* inside a ray-tracing pipeline.

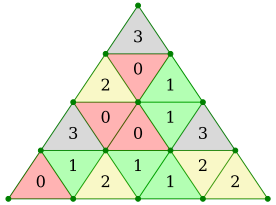
And this is entirely without using any kind of alpha-mapping: The micromaps have effectively made our geometry more granular *almost* for free!

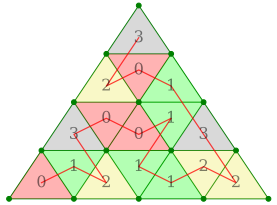
Micromaps – VK_EXT_opacity_micromap

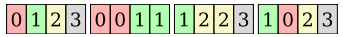


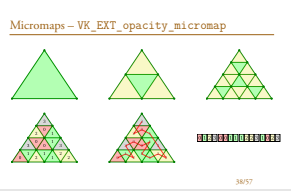












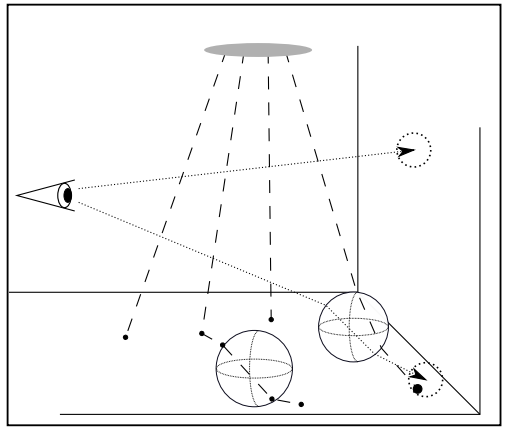
Without digging into too many of the details, (And I mean, it is a Vulkan® extension, there are *loads* of details to go around), we can still pretty easily describe it as follows:

1. Take any triangle,
2. Split each edge at their midpoint, forming 4 new, virtual *sub-triangles*,
3. Repeat this n times,
4. Next, associate a value with each of the *sub-triangles*,
5. Finally, linearize these values with a special space-filling curve, giving us an array of values: I.e., our micromap.

In short: You can view the micromap as very specialized per-triangle texture. And I want to be able to generate some realistic data for it.

Opacity Micromaps

0 Transparent
1 Opaque
2 Unknown (Transparent)
3 Unknown (Opaque)



Opacity Micromaps
0 Transparent
1 Opaque
2 Unknown (Transparent)
3 Unknown (Opaque)
39/57

Now, in Vulkan®specifically, there is currently only one official type of Micromap available: Namely the “Opacity Micromap”, that is, a map that stores opacity values at each subtriangle, and only very specialized ones at that:

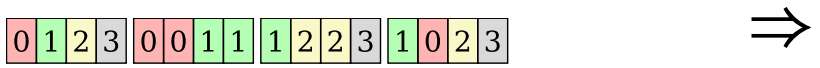
Transparent (0) Sub-triangle is 100% transparent,

Opaque (1) It is 100% opaque,

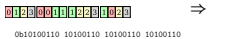
Unknown (2, 3) Unknown opacity with escape hatches for faster processing in some cases.

Naturally, this is a specialized type of opacity, and in fact, is targeted at improving the performance of hardware accelerated ray-tracing applications by reducing the number of Any-Hit shader calls, but that is a discussion for a different time.

Opacity Micromaps

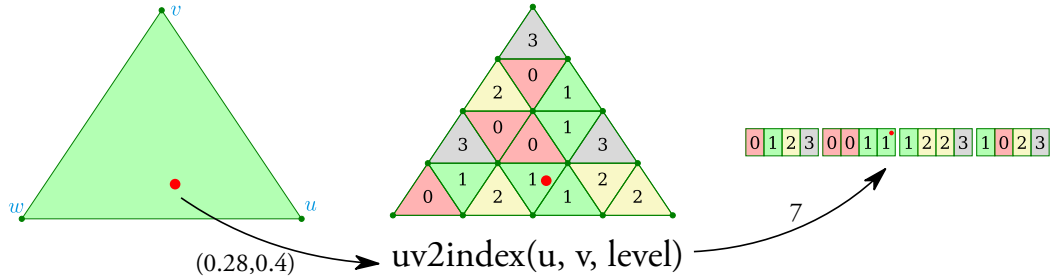


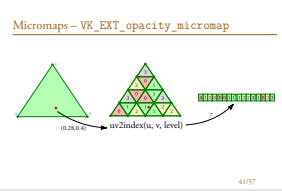
Opacity Micromaps



And while this looks like we are storing an array of values, the kicker is though that each of these elements only need two *bits* each. In fact, there is another mode where we only use a single bit per value. Hence, these micromaps can be extremely compact compared to a typical alpha-map.

Micromaps – VK_EXT_opacity_micromap

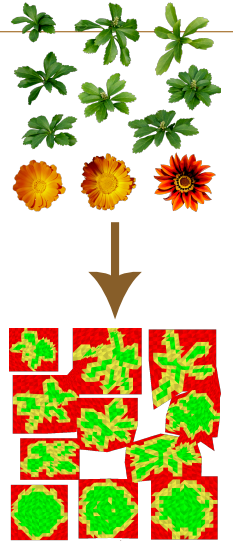




And for completeness: At rendering-time, we have a very efficient algorithm for going from barycentric coordinates to a micromap-index and consequently the micromap-value stored in that specific *sub-triangle*.

Opacity Micromaps Exporter

```
for triangle in scene:
    if tri.alpha_map:
        generate_micromaps(tri.alpha_map)
```



Opacity Micromaps Exporter

for triangle in scene:
if tri.alpha_map:
generate_micromaps(tri.alpha_map)

42/57

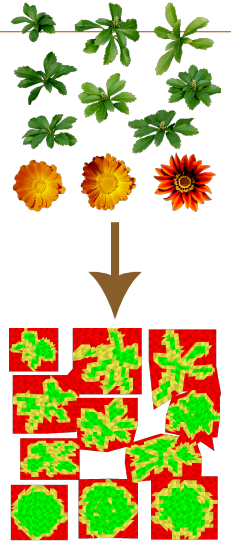
So, in my case, I want to create a plugin that can generate these micromaps and embed them as a part of the `glTF`-file. And, we also want them to be *realistic*, that is, we want our exporter to find any *alpha-texture* associated with each triangle and generate a “representative” opacity micromap for it.

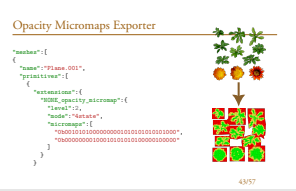
So, what we want to do is the following:

- For each triangle:
- Add micromaps if it has an alpha texture.

Opacity Micromaps Exporter

```
"meshes":[
{
  "name":"Plane.001",
  "primitives":[
    {
      "extensions":{
        "NONE_opacity_micromap":{
          "level":2,
          "mode":"4state",
          "micromaps":[
            "0b00101010000000001010101010101000",
            "0b00000000100010101010100000100000"
          ]
        }
      }
```





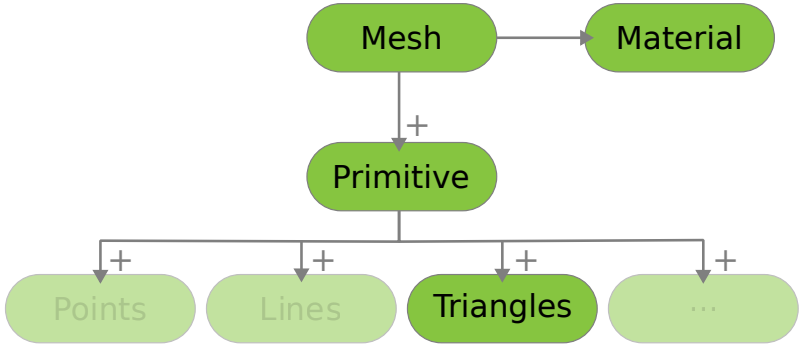
And just so we are all clear on exactly what we want in the end:

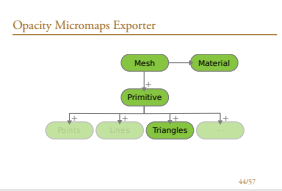
We want to be able to generate `glTF`-file that contains extension objects that look something like this:

- That is, each primitive (i.e., list of triangles) now have a list of micromaps associated with it, with any interesting metadata (such as the subdivision level).

(This is also one of the reasons we would like to generate these during the export phase: The micromaps needs to be matched with the correct vertex indices, and the Blender indices does not necessarily match the one used by `glTF`.)

Opacity Micromaps Exporter



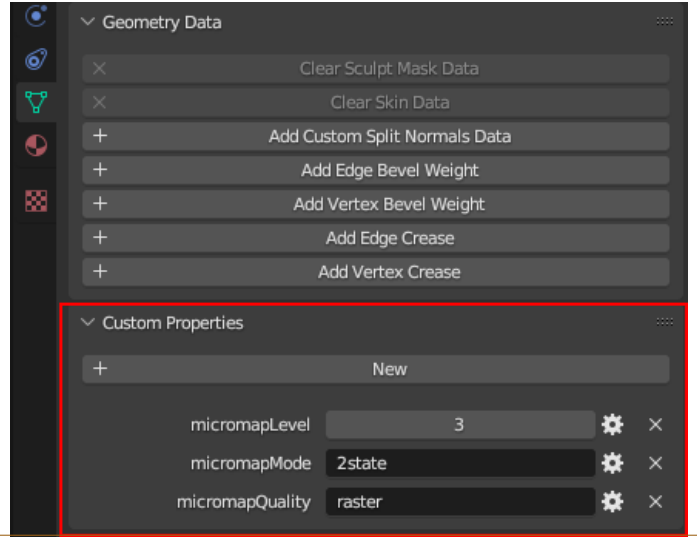


So on the face of it, it looks like we should be able to just add a customization method for primitives and just process each of them as we export.

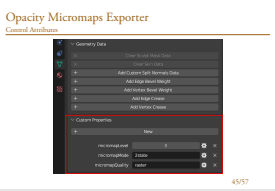
Unfortunately for us, or *me* at least, `glTF` structures primitives something like *this*. Of particular note is that materials are attached to the *mesh* rather than the primitive. So we actually need to move up a bit and customize the mesh-exporter instead.

Opacity Micromaps Exporter

Control Attributes



LUND UNIVERSITY



Another good reason for moving up to the mesh level is that we can more easily attach some custom control attributes on each mesh, that we can use to control the micromap generation process on a per-mesh-basis.

Opacity Micromaps Exporter

```
for mesh in gltf_scene:
    for primitive in mesh:
        alpha_map = mesh.material.baseColorTexture
        if alpha_map:
            indices = gltf_access(primitive.indices)
            tex_coords = gltf_access(primitive.tex_coords)
            primitive.extension = micromaps_extension(...)
```


Opacity Micromaps Exporter

```
for mesh in gltf_scene:
    for primitive in mesh:
        alpha_map = mesh.material.baseColorTexture
        if alpha_map:
            indices = gltf_access(primitive.indices)
            tex_coords = gltf_access(primitive.tex_coords)
            primitive.extension = micromaps_extension(...)
```

46/57

Thus, to adapt our loop to the `glTF` world, we will do roughly the following:

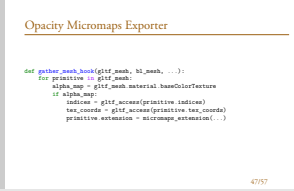
- Each time we export a mesh:
  - Check if the triangle contains alpha-textures. If so:
    1. Extract the `glTF` triangles (indices, uvs, materials and textures).
    2. Use these to map all triangles to the alpha texture(s).
    3. Generate the micromaps from that mapping.
    4. Store them in the `.gltf`-file.

Now this seems pretty easy on the face of it: And it sort of is when you have dug deeply enough into the `glTF` model:

Opacity Micromaps Exporter

```
def gather_mesh_hook(gltf_mesh, bl_mesh, ...):
    for primitive in gltf_mesh:
        alpha_map = gltf_mesh.material.baseColorTexture
        if alpha_map:
            indices = gltf_access(primitive.indices)
            tex_coords = gltf_access(primitive.tex_coords)
            primitive.extension = micromaps_extension(...)
```


Opacity Micromaps Exporter



47/57

All we had to do, was replace the top of the loop with the `gather_mesh_hook()`, and we're basically off to the races.

There are a number of caveats here that make this a bit more challenging in practice.

- Each time a micromap is divided, we quadruple the number of sub-triangle.
  - That is $4^n \cdot$ # primitives. That's a lot of triangles.
  - But each of them are computationally independent

Clearly, we want to parallelize this as much as possible, which in Pythons case means bringing in `multiprocessing`, and probably also some way of profiling the code to find any problematic snippets.

Multiprocessing

- No locals
- Avoid nested parallelism
- (Most) Arguments are *copied*
- Appropriate chunk-sizes

```
for work_item in chunk:
    run_task(work_item)

chk_sz = max(1, nitems // ncore)
```


Multiprocessing

- No locals
- Avoid nested parallelism
- (Most) Arguments are *copied*
- Appropriate chunk-sizes

```
for work_item in chunk:
    run_task(work_item)

chk_sz = max(1, nitems // ncore)
```

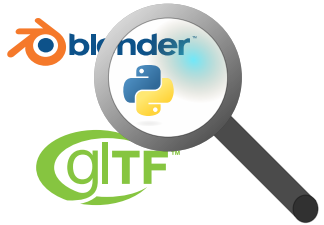
48/57

And Multiprocessing in itself is a complicated topic, with numerous corner cases, as such I won't really go into it. But I do want to give some general advice if you also end up in this situation:

- Remember that local variables and functions are typically not available,
- Avoid nested parallelism. Try to *flatten* any nested structures and provide those as arguments instead.
- Remember that practically all data we send to the processes has to be copied, which could be expensive if we are using textures as arguments.
- Set appropriate chunk-sizes: The Python default of 1 is often inadequate. Try to divide your tasks evenly between your cores!

Profiling Plugins

```
glTF2_pre_export_callback(export_settings)
glTF2_post_export_callback(export_settings)
```



Profiling Plugins
glTF2_pre_export_callback(export_settings)
glTF2_post_export_callback(export_settings)
49/57

And profile your application. Python has tools that these tasks easier, so try to use them when possible! And the `glTF` addon has two more special methods that we can use help this out even further:

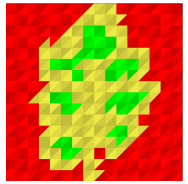
- `glTF2_pre_export_callback(export_settings)`
- `glTF2_post_export_callback(export_settings)`

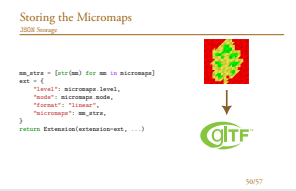
As their name suggests, these functions run before and after the export, as such they are a good place to start or stop the profiling process. Or, for printing out the actual profiling results in the end.

Storing the Micromaps

JSON Storage

```
mm_strs = [str(mm) for mm in micromaps]
ext = {
    "level": micromaps.level,
    "mode": micromaps.mode,
    "format": "linear",
    "micromaps": mm_strs,
}
return Extension(extension=ext, ...)
```





But moving back to the micromaps:

Obviously, I didn't learn everything about `glTF` (and micromaps) at once, so I actually ended up with more than one "format" to store them, hence I actually wanted to store this data in different ways for different use-cases:

- Small micromaps used for testing purposes is most easily manged in a string.
  - This makes it easier to quickly (and manually) change the micromap after exporting it.
- Large micromaps are probably not modified in such a way, so for them, we may as well store them using the existing `glTF` buffer-system.

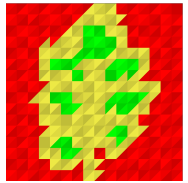
(Here is one caveat that is worth mentioning: While `glTF` is working, the entire `JSON` portion is actually kept in memory, so if you store too much in there, we can actually run out of memory during the export.

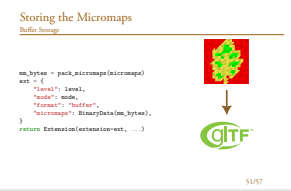
As an example, I once attempted to export a gigantic scene with large micromaps in the string format. And while everything seemed to work in the beginning, I was left scratching my head for a while until I realized what was happening.)

Storing the Micromaps

Buffer Storage

```
mm_bytes = pack_micromaps(micromaps)
ext = {
    "level": level,
    "mode": mode,
    "format": "buffer",
    "micromaps": BinaryData(mm_bytes),
}
return Extension(extension=ext, ...)
```



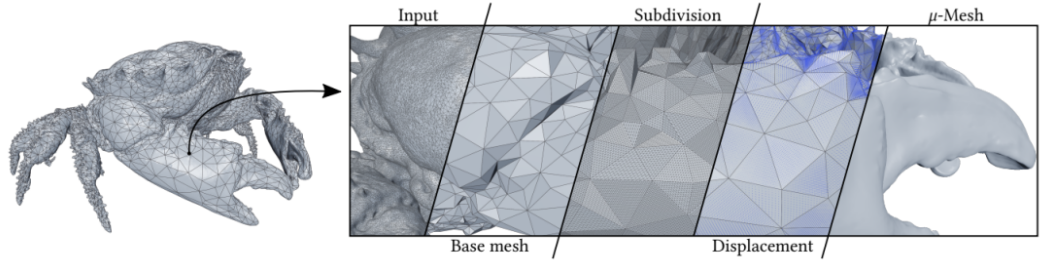


So, in the previous example we only stored the micromaps as strings in the JSON format, which of course, is terribly inefficient, effectively using eight times as much storage as necessary.

Instead, the better thing to do in this case is to use the existing `glTF` buffer system, which we can easily do by just using the `BinaryData` class from before:

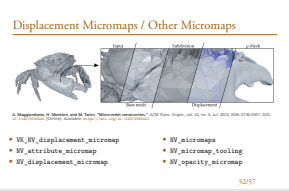
We just convert the micromap data directly to flat buffer of bytes and give that to the `BinaryData` class, then `glTF` will handle the rest.

Displacement Micromaps / Other Micromaps



A. Maggiordomo, H. Moreton, and M. Tarini, "Micro-mesh construction," *ACM Trans. Graph.*, vol. 42, no. 4, Jul. 2023, ISSN: 0730-0301. DOI: 10.1145/3592440. [Online]. Available: https://doi.org/10.1145/3592440

- `VK_NV_displacement_micromap`
- `NV_attribute_micromap`
- `NV_displacement_micromap`
- `NV_micromaps`
- `NV_micromap_tooling`
- `NV_opacity_micromap`

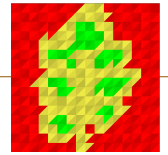


Obviously, in this talk I only talked about *Opacity* Micromaps. However, Nvidia have defined at least one other type known as a *Displacement* Micromap which is already available in Vulkan® and DirectX® as a provisional extension. On top of that, it seems like they are also working more types of micromaps, as they have defined `glTF` extensions for micromaps with more general attributes on each sub-triangle.

This is not something I've had time to explore in detail, but modifying my plugin to output data in the same format seems like a good idea for the future.

Additionally, displacement micromaps might be something to keep an eye on in the future. As this could be something that may be nice to somehow fit into the sculpting baking process. But, this is just me speculating; I don't know enough about this extension or the baking process just yet to have a valid opinion.

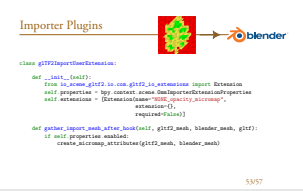
Importer Plugins



```
class glTF2ImportUserExtension:

    def __init__(self):
        from io_scene_gltf2.io.com.gltf2_io_extensions import Extension
        self.properties = bpy.context.scene.OmmImporterExtensionProperties
        self.extensions = [Extension(name="NONE_opacity_micromap",
                                     extension={},
                                     required=False)]

    def gather_import_mesh_after_hook(self, gltf2_mesh, blender_mesh, gltf):
        if self.properties.enabled:
            create_micromap_attributes(gltf2_mesh, blender_mesh)
```

Lastly, I want to talk briefly about *importers*. Most of this talk has obviously focused on exporters, which makes sense, it is what I've primarily worked with. But, it is also the most common use for `glTF`; typically you have a detailed model with a lot of data (in Blender or some other framework) that you effectively are filtering as you export it.

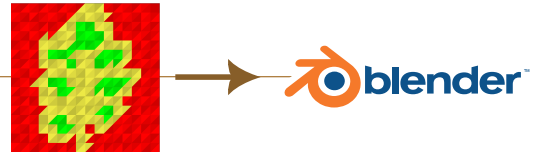
There are of course exceptions, and the most compelling use case is probably compression. Say that we have developed a new compression algorithm, then we might use an importer addon to test the decoding process.

And really, the structure of these addons are very similar to the exporter, the main thing that *can* differ is that we sometimes have `before` and `after` hook: Since we cannot really work on compressed data without actually decoding it!

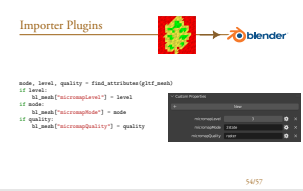
Obviously, I'm not skilled enough to develop something like that for this talk, so we'll settle for something much easier:

This simple importer that looks for the presence of micromaps in the input gltf-mesh, and if so, adds the appropriate attributes such that they will be generated during the next export.

Importer Plugins



```
mode, level, quality = find_attributes(gltf_mesh)
if level:
    bl_mesh["micromapLevel"] = level
if mode:
    bl_mesh["micromapMode"] = mode
if quality:
    bl_mesh["micromapQuality"] = quality
```



And of course, setting attributes such as these is very easy:

- We simply need to find some attributes such as the mode, level and quality in this case which I do by scanning the mesh for the *extension* key, (but omitted here for brevity).
- Then we simply set this attributes on the blender mesh.
- A tool like that could be useful if you're importing old models or are compositing several models together.

Conclusion and Summary

- What is `glTF`?
- The `glTF` Addon
- Creating Plugins
  - Watermarking
  - Micromaps
- Tips and Tricks
  - Profiling
  - Multiprocessing

Conclusion and Summary

- What is glTF?
- The glTF Addon
- Creating Plugins
  - Watermarking
  - Micromaps
- Tips and Tricks
  - Profiling
  - Multiprocessing

55/57

And just to quickly summarize things: In this talk we've talked about…

- `glTF`: what it is and how it is structured,
- the `glTF` addon that Blender uses to import and export `glTF` files.
- "Plugins" for this addon that allows us to customize the export and import.
  - Along with a few examples such as the watermarker, and
  - micromap plugins.
- And finally, I also provided a small collection of tips and tricks for handling Multiprocessing or profiling as well as how some hints at effective use of in Blender.

Thanks for Listening

Questions

- Thanks for listening!
- Questions and Answers

Thanks for Listening
Questions
• Thanks for listening!
• Questions and Answers
56/57

And with all that, I think it's about time to open up for questions, should you have any!

The End