LUND UNIVERSITY | LTH FACULTY OF ENGINEERING

PBRT: Create your own Importers and Exporters

Gustaf Waldemarson
Arm & Lund University

BCON24 BLENDER CONFERENCE OCTOBER 23 – 24 – 25 AMSTERDAM

2024-10-22

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

Thanks everyone for coming!

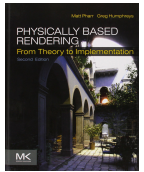
(And it's nice to see so many people here again this year!).

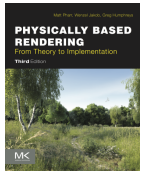
Though, I should warn you all: This will be a mostly technical talk. As such, I'm afraid that the supply of neat renders and images may run a bit sparse at times, but I hope you will all learn something from this at least!

Who am I?

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

arm

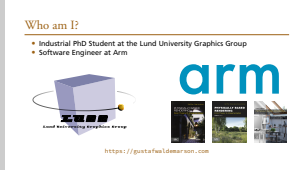




https://gustafwaldemarson.com

LUND UNIVERSITY

ECON24



And this is actually my second year of presenting at BlenderCon, but I still think some kind of self-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 graphics driver stack, and particularly the parts related to hardware 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 may 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 sponsored by Arm, and any views or opinions herein are entirely my own.

And, in case you're wondering, I am also not affiliated with the people behind PBRT in any way. It just so happens that I've used it quite a bit over the years.

Agenda

1. What is PBRT?
   - Why use it?
2. The Importer
   - Earlier Work
3. Proxy Objects and Renderer Settings
4. The Exporter

Agenda
1. What is PBRT?
   - Why use it?
2. The Importer
   - Earlier Work
3. Proxy Objects and Renderer Settings
4. The Exporter

But with that, let's jump into the meat of things: In this talk I will primarily discuss three things:

1. PBRT, the rendering framework and file format,
2. How to create a custom importer in Blender, and,
3. How to create a custom exporter.

I will also briefly touch upon some aspects about custom renderers, but more on that in a bit.

First though, I hope that I haven't lured anyone here with the hopes of learning about *PBR* textures and techniques. While the letters are the same, and actually stand for the same thing, that is actually a different topic that is out-of-scope for this talk!

Blender Renderers

Blender

Cycles	Unidirectional volumetric path tracer
Eevee	Real-time *rasterization* based rendering engine
Workbench	Real-time *preview* rendering engine

Blender Renderers

Blender

Cycles Unidirectional volumetric path tracer
Eevee Real-time *rasterization* based rendering engine
Workbench Real-time *preview* rendering engine

First though, let us briefly summarize what we already have in Blender in terms of rendering engines. By default, we have:

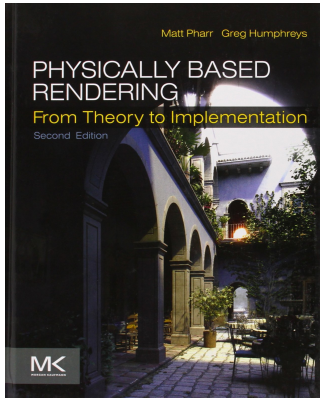
- Cycles, the unidirectional volumetric path tracer, that we use to create our nicer images, and,
- Eevee, the real-time, raster-based engine, for a bit faster rendering, and, of course:
- The workbench renderer, which we typically only use for previewing our work.

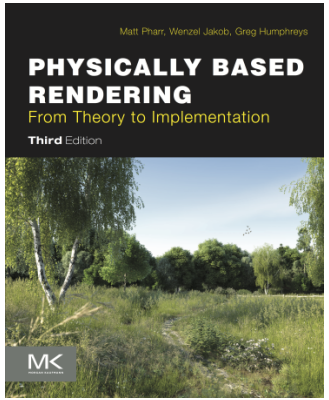
I must say, that I don't *really* know a lot about the internals of these renderers, but from what I've seen so far, both *here* and elsewhere, they do appear to be very flexible and feature-full frameworks.

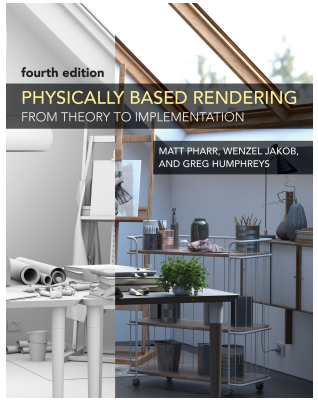
One could also argue, that some features, such as view layers with clever compositing make these count as *multiple* different rendering engines. *Still*, sometimes it is desirably to use something different, something which may be more useful for some specialized task.

PBRT

Physically Based Rendering – From Theory to Implementation







https://pbr-book.org/3ed-2018/contents
https://pbr-book.org/4ed/contents

ECCV24

LUND UNIVERSITY

And this is kind of where PBRT comes in. In full, it stands for, Physically Based Rendering, from Theory to Implementation, and it is both an award-winning text-book and a research and educational framework for developing new, or improving existing rendering algorithms.

It is also open-source under a liberal license, and as such it has been quite influential over the last few decades. And in the last two editions have even been made freely available online, which you can find them through these links, making them an excellent place for learning about all things rendering.

PBRT – The Renderer(s)

PBRT-v2

- Ambient Occlusion
- Instant Global Illumination
- Irradiance Cache
- Path Tracer
- Light Probes
- Whitted RT
- …

PBRT-v3

- Bidirectional Path Tracing
- Direct Lighting
- Whitted RT
- Path Tracing
- Volumetric Path Tracing
- Metropolis Light Transport
- Stochastic Progressive Photon Mapping
- …

PBRT-v4

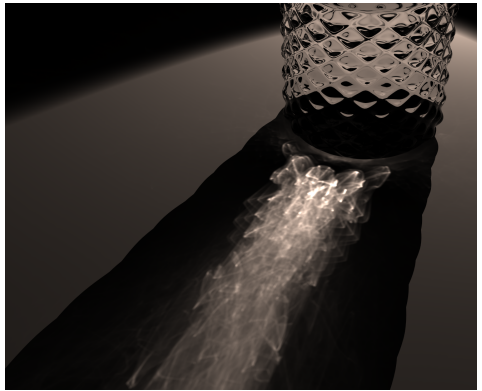
- Bidirectional Path Tracing
- Direct Lighting
- Whitted RT
- (Volumetric) Path Tracing
- Metropolis Light Transport
- Light Path Tracing
- Simple (Volumetric) Path Tracing
- Stochastic Progressive Photon Mapping
- …

As for the renderers that already exist in PBRT, they are plentiful, to say the least, as I hope these list illustrates, even if it is a bit disingenuous: As I said before, with composition we can easily get some things, such *Ambient Occlusion* rendering in cycles and Eevee. Here though, that would correspond to a different renderer.

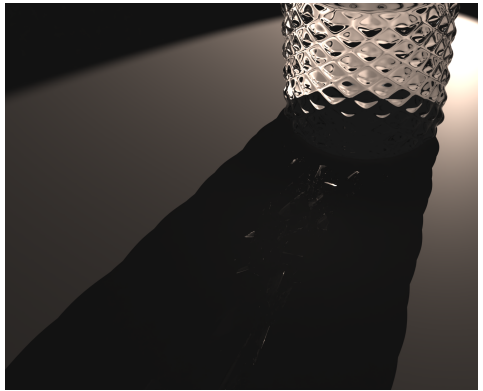
(Note that as this is in part a research framework, the editions have consequently followed various trends in computer graphics research, hence the varying availability of rendering algorithms.)

PBRT – Why?

Photon Mapping Example

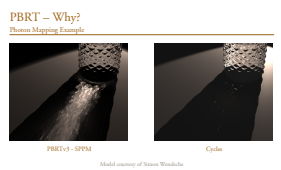


PBRTv3 - SPPM



Cycles

Model courtesy of Simon Wendsche



And just to show you an example: This is one such case where it makes sense to have a specialized renderer at hand, as scenes with a lot of caustics is still an extremely challenging thing to render:

To the left, I rendered this caustics-heavy example scene with the Stochastic Progressive Photon Mapping renderer in PBRT and to the right I did my best to recreate the same kind of caustics in Cycles, but even after digging through lots of renderings setting I just *didn't* manage to get much of anything, as I hope you can see.

Although, I fully expect that this could be significantly improved by someone who knows *Cycles* better than I do. Still, the fact remains that if you know what your scene is going to contain, it sometimes may make more sense to choose a renderer that is more suited for that task.

HBO Miniseries: Chernobyl

Cherenkov Radiation

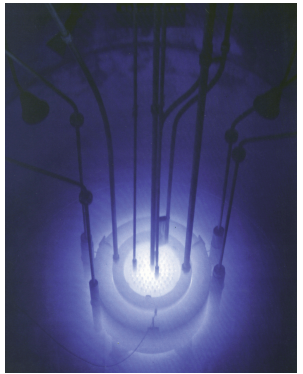
– Images from “Chernobyl” Episode 1-2 © HBO

HBO Miniseries: Chernobyl
Cherenkov Radiation

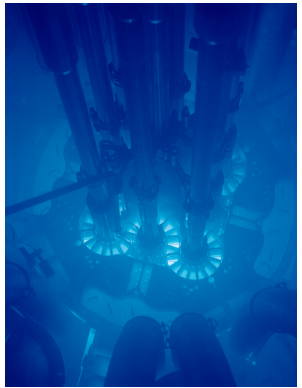
– Images from “Chernobyl” Episode 1-2 © HBO

But, as I said, PBRT is also an ideal framework for developing *new* rendering algorithms. *So*, way-back-when, in 2019, HBO aired their excellent Chernobyl miniseries, and in the first few episodes, some of the eerie lighting effects where attributed to *Cherenkov Radiation*, which caught my interest. So, I did a bit of research on that topic.

Nuclear Reactors



Reed Research Reactor



Advanced Test Reactor

Nuclear Reactors

Reed Research Reactor

Advanced Test Reactor

Typically, it is a phenomenon that can be seen in the proximity of active nuclear reactors, as you can see in these photographs.

But, when I looked for some, let's say, *neater* renderings of the effect in a more controlled setting, I was unable to find much.

Superluminal Photon Mapping

Cherenkov Radiation

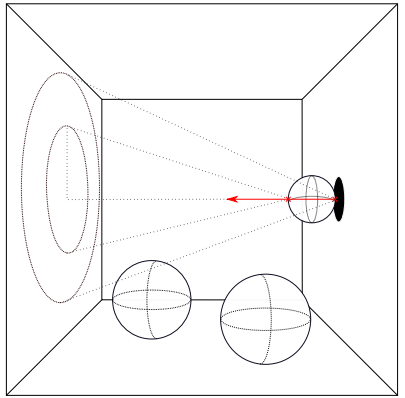
Superluminal Photon Mapping
Cherenkov Radiation

And this video simulating a single particle in 2D was among the few things I found, showing how photons are emitted in a cone from the path of a charged particle.

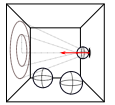
While it is neat, and describes the phenomenon quite well, it was not exactly the neat looking rendering I was after.

That got me thinking: Could I modify the photon mapping algorithm somewhat to generate something similar, but in 3D, and with more accurate lighting?

Superluminal Photon Mapping



Superluminal Photon Mapping



To that end, I created a simple variation of the so-called Cornell Box like this, letting a charged particle enter the scene along the red arrow, which should then, in theory, cast light in a cone like *this*.

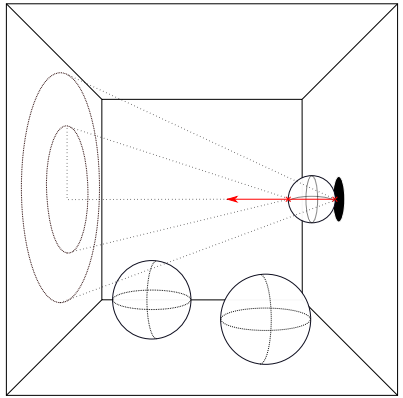
And, after a bit of implementation and experimentation in my homegrown renderer, I got something that looked like *this*. **(Next Slide)**

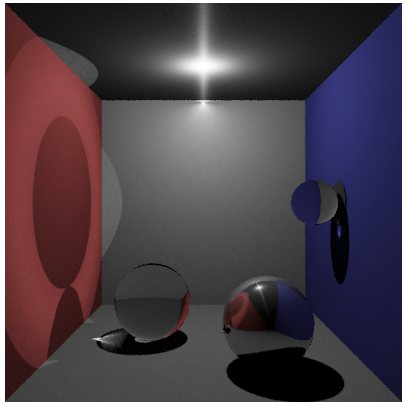
Doesn't look like much, but it was a good proof of concept.

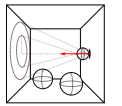
So, after a bit of porting work to PBRT, I was able to get a lot of features almost for free:

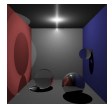
Next Slide

Superluminal Photon Mapping









To that end, I created a simple variation of the so-called Cornell Box like this, letting a charged particle enter the scene along the red arrow, which should then, in theory, cast light in a cone like *this*.

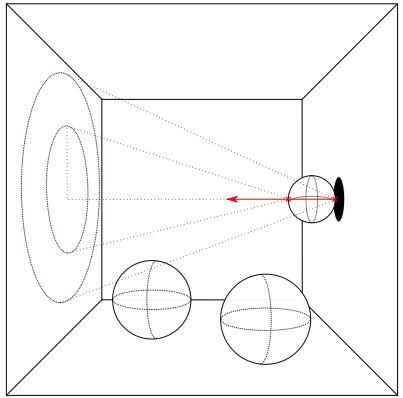
And, after a bit of implementation and experimentation in my homegrown renderer, I got something that looked like *this*. **(Next Slide)**

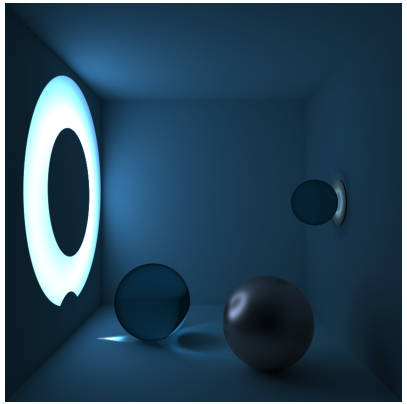
Doesn't look like much, but it was a good proof of concept.

So, after a bit of porting work to PBRT, I was able to get a lot of features almost for free:

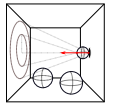
Next Slide

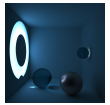
Superluminal Photon Mapping





Superluminal Photon Mapping

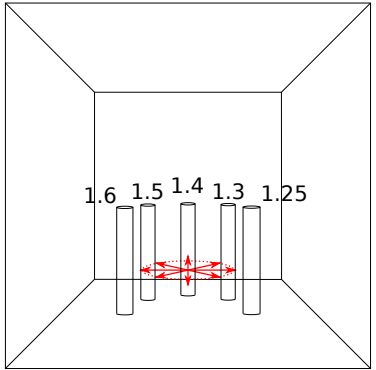


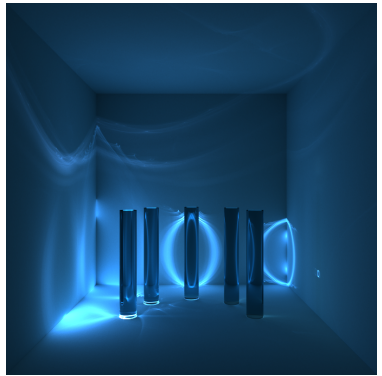


Which gave me a rendering that looked like *this*, which is much better than what I could easily create in my own experimental framework.

Basically, in PBRT I got a lot of **extra** features for free, such as accurate handling of spectral properties, which really were necessary for an effect like this.

Superluminal Photon Mapping



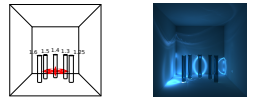


Gustaf Waldemarson and Michael Doggett. "Photon Mapping Superluminal Particles". In: *Eurographics 2020 - Short Papers*. Ed. by Alexander Wilkie and Francesco Banterle. The Eurographics Association, 2020. ISBN: 978-3-03868-101-4. DOI: 10.2312/egs.20201004

ECGN24

LUND UNIVERSITY

Superluminal Photon Mapping



And, this phenomenon depends quite a lot on the density of the material the particle passes through, so to see how that would affect things, we created another scene with rods of various index-of-refraction, giving us a rendering like *this*.

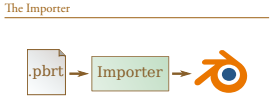
This, we wrote up in a paper a number of years ago now, so if you are interested, you can find more details down here, or on my personal web-page along with the code for it.

But with that, I hope I have given you a sufficient motivation why we may want to use different framework: Either to improve some specific effect, or, to make it easier to try out things during research.

Now though, it's high time to start talking about how we can use Blender to help with this matter by creating an importer to get things in, and exporters to move Blender objects to our renderer of choice!

So today, in keeping with this years BlenderCon theme, we are taking the first steps in "Making Blender *Love* PBRT!"

The Importer

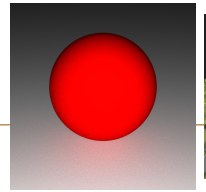


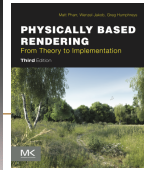
So, I want to start with talking about the importer, since it is sometimes desirable to be backwards compatible to some degree, and thus being able to *import* existing scenes directly into Blender could be a huge boon.

Furthermore, for PBRT in particular, it is *rumored* that it has been used as the base for creating various production renderers (LuxCoreRenderer was one at some point). So getting some of those scenes into Blender for comparison could be very interesting.

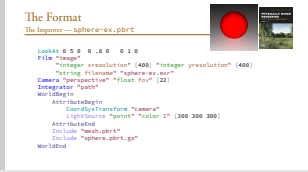
The Format

The Importer — sphere-ex.pbrt






```
LookAt 0 5 8  0 .8 0   0 1 0
Film "image"
     "integer xresolution" [400] "integer yresolution" [400]
     "string filename" "sphere-ex.exr"
Camera "perspective" "float fov" [22]
Integrator "path"
WorldBegin
    AttributeBegin
        CoordSysTransform "camera"
        LightSource "point" "color I" [300 300 300]
    AttributeEnd
    Include "mesh.pbrt"
    Include "sphere.pbrt.gz"
WorldEnd
```

But first, just so we are all on the same page, let us take a look at the format of a PBRT file...

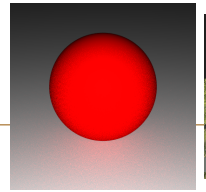
In short, you can think of it is being separated in two parts: In the first of which, the top-portion up here, we describe *how* we want to render the image, setting various parameters about how the output image should be stored, where the camera should be located, and so forth.

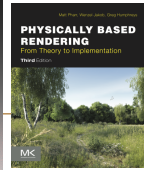
After that, we describe the contents of the scene, or *world* itself, placing light-sources and objects with the help of various transform directives as well as dressing them up with materials.

It is also possible to *Include* other files, which *may* also be compressed using `gzip` compression as you can see from the file-extension of the last file down here.

The Format

The Importer — mesh.pbrt / sphere.pbrt.gz





mesh.pbrt

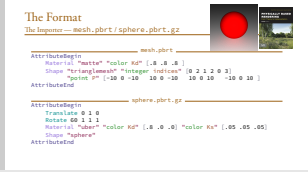
```
AttributeBegin
    Material "matte" "color Kd" [.8 .8 .8 ]
    Shape "trianglemesh" "integer indices" [0 2 1 2 0 3]
          "point P" [-10 0 -10   10 0 -10   10 0 10   -10 0 10 ]
AttributeEnd
```

sphere.pbrt.gz

```
AttributeBegin
    Translate 0 1 0
    Rotate 60 1 1 1
    Material "uber" "color Kd" [.8 .0 .0] "color Ks" [.05 .05 .05]
    Shape "sphere"
AttributeEnd
```


ECCN24

LUND UNIVERSITY



And each of these separate files simply contains more statements of the same kind, as you can see here with a description of the base-plane mesh and a slightly transformed red sphere.

And, once you run a scene like this through PBRT, you get an image that looks like this one up in corner. Nothing *too* exciting, but a good starting point to get an idea of what the format looks like.

Earlier Work

The Importer

- https://github.com/mxpv/pbrt4 (Rust)
- https://github.com/vilya/minipbrt (C++)
- https://github.com/ingowald/pbrt-parser (C++)

Earlier Work
The Importer
• https://github.com/mxpv/pbrt4 (Rust)
• https://github.com/vilya/minipbrt (C++)
• https://github.com/ingowald/pbrt-parser (C++)

To my knowledge, there is no PBRT importer out there for Blender, but there are a few libraries out there for parsing the file format, which makes sense, since it *is* a relatively easy format to work with. I wouldn't be surprised if there are many more in-house parsers out here; as even I have had my own C++ parser for it since quite a few years back now.

It probably would be possible to use one of these libraries with Blender, but doing so would complicate matters quite a bit as we would need to integrate and distribute third-party binary libraries, which felt rather tough, and would make a quite different talk. So here, everything will be done in plain Python.

This means it probably won't win any performance drag-races, but for what is probably only going to be a one-time cost for importing the file into Blender, being more general and portable feels like the right choice.

An Earlier Attempt: PBRT → glTF → Blender

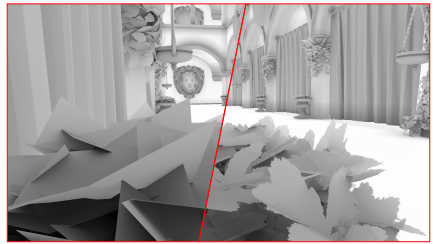
The Importer

An Earlier Attempt: PBRT → glTF → Blender
The Importer

As I mentioned, I technically already had a PBRT parser, and I even used it quite a bit this year to generate test data for a different project by first converting the PBRT format to glTF before using that importer to bring it into Blender.

Issues

- Most PBRT objects cannot be represented
- We lose the material and texture graphs



LUND UNIVERSITY

Issues
Most PBRT objects cannot be represented
We lose the material and texture graphs

This works, but is a bit problematic for a few reasons:

Firstly, PBRT has quite a number of primitives, most of which are not supported by `glTF`, but most crucially, while `glTF` has a great material model, the one available in PBRT is substantially deeper, with a full graph to describe mixing of both materials and textures. As such, in my converter I was forced to adjust all materials or simply drop that information.

Luckily, for my earlier project I was only interested in ambient occlusion rendering, so I did not *actually* have to care about this loss. However, going forward it is something I am going to need, hence this effort!

(Obviously, it should be possible to add quite a bit of this as custom `glTF` extensions, but as there was so many things that were needed, that felt like the wrong thing to do!)

The Importer

What do we need to do?

- An Import *Operator*[1]
- An Import Panel[2] (optional)
- An Import Menu Entry[3] (optional)

[1] https://docs.blender.org/manual/en/latest/interface/operators.html
[2] https://docs.blender.org/api/current/bpy.types.Panel.html
[3] https://docs.blender.org/api/current/bpy.types.Menu.html

So, what do we *actually* need to do from Blenders point of view?

In short, we only *have* to create a so-called `Operator`, i.e., something Blender can identify as modifying something in the scene.

And of course, there are various flavors of these, but we want to create Import and Export operators.

But, for convenience we probably also want to add some extra things, such as a panel for modifying some properties of the import and a menu entry to start the process from the user interface.

The Operator

The Importer


```python
class ImportPBRT(bpy.types.Operator, bpy_extras.io_utils.ImportHelper):
    """Blender Importer class for PBRT scenes."""
    # ... Config variables...

    def draw(self, context):
        """Specify how to draw the Blender panel UI."""

    def execute(self, context):
        """Execute the PBRT import process."""
```


Starting with the mandatory: To create an import operator, we need to create a class that inherits from the Blender `bpy.types.Operator` class, and in our case, we also want to inherit from the `ImportHelper`, since that will add a couple of conveniences, such as helping with the file-selection process.

The Operator

The Importer – Configuration Variables


```
class ImportPBRT(bpy.types.Operator, bpy_extras.io_utils.ImportHelper):
    """Blender Importer class for PBRT scenes."""
    bl_idname = "import_scene.pbrt"
    bl_label = "Import PBRT"
    bl_options = {"REGISTER", "UNDO"}

    filter_glob: StringProperty(default="*.pbrt;*.pbrt.gz")
    files: CollectionProperty(name="File Path",
                              type=bpy.types.OperatorFileListElement)
    mode: EnumProperty()
    parse_only: BoolProperty()
```


ECCN24

LUND UNIVERSITY

The Operator
The Importer – Configuration Variables
.pbrt Importer
class ImportPBRT(bpy.types.Operator, bpy_extras.io_utils.ImportHelper):
"""Blender Importer class for PBRT scenes."""
bl_idname = "import_scene.pbrt"
bl_label = "Import PBRT"
bl_options = {"REGISTER", "UNDO"}
filter_glob: StringProperty(default="*.pbrt;*.pbrt.gz")
files: CollectionProperty(name="File Path",
type=bpy.types.OperatorFileListElement)
mode: EnumProperty()
parse_only: BoolProperty()

As with a lot of Blender classes, we need a set of configuration variables and properties for it to function.

The Operator

The Importer – Configuration Variables


```
class ImportPBRT(bpy.types.Operator, bpy_extras.io_utils.ImportHelper):
    """Blender Importer class for PBRT scenes."""
    bl_idname = "import_scene.pbrt"
    bl_label = "Import PBRT"
    bl_options = {"REGISTER", "UNDO"}

    filter_glob: StringProperty(default="*.pbrt;*.pbrt.gz")
    files: CollectionProperty(name="File Path",
                              type=bpy.types.OperatorFileListElement)
    mode: EnumProperty()
    parse_only: BoolProperty()
```

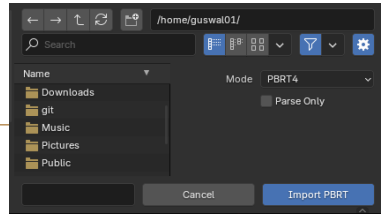

In this case, I would say that the most important ones would be these ones:

- ID-name, That give the operator a function to be called by, and,
- The `files` property, which will contain the files to be imported when the operator runs.

Any other properties that you would like, such as which parsing mode, etc, can also be added here.

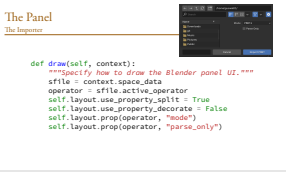
The Panel

The Importer




```python
def draw(self, context):
    """Specify how to draw the Blender panel UI."""
    sfile = context.space_data
    operator = sfile.active_operator
    self.layout.use_property_split = True
    self.layout.use_property_decorate = False
    self.layout.prop(operator, "mode")
    self.layout.prop(operator, "parse_only")
```


The Panel
The Importer



def draw(self, context):
"""Specify how to draw the Blender panel UI."""
sfile = context.space_data
operator = sfile.active_operator
self.layout.use_property_split = True
self.layout.use_property_decorate = False
self.layout.prop(operator, "mode")
self.layout.prop(operator, "parse_only")

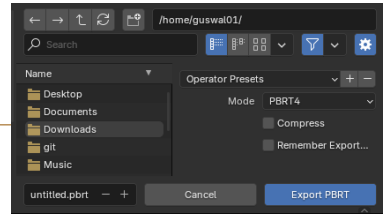
As for the panel: We have two options:

- We can either create a `draw` method directly in the operator class, like this,
- Or create a separate class to specify how and where to place the panel.

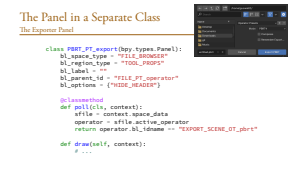
Doing it like this would then give you these *extra* options that you can set in your file-picker.

The Panel in a Separate Class

The Exporter Panel



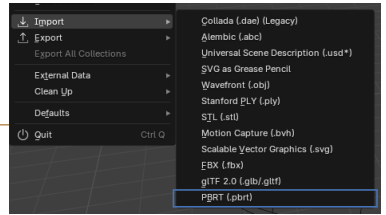

```python
class PBRT_PT_export(bpy.types.Panel):
    bl_space_type = "FILE_BROWSER"
    bl_region_type = "TOOL_PROPS"
    bl_label = ""
    bl_parent_id = "FILE_PT_operator"
    bl_options = {"HIDE_HEADER"}

    @classmethod
    def poll(cls, context):
        sfile = context.space_data
        operator = sfile.active_operator
        return operator.bl_idname == "EXPORT_SCENE_OT_pbrt"

    def draw(self, context):
        # ...
```

Creating a panel class gives a bit more control, as shown here for the export operator. Although, it is a bit redundant in this case: As far as I understand, the current default is to place the panel in the `FileBrowser`. So this example simply places it in the default place, as sometimes, more control is desired.

The Menu Entry

The Importer




```python
def menu_func_import(self, context):
    """Function to run when executing the import menu item."""
    self.layout.operator(ImportPBRT.bl_idname, text='PBRT (.pbrt)')

def register():
    """Register the addon with Blender."""
    bpy.types.TOPBAR_MT_file_import.append(menu_func_import)

    # ...
```


The Menu Entry
The Importer

def menu_func_import(self, context):
    """Function to run when executing the import menu item."""
    self.layout.operator(ImportPBRT.bl_idname, text='PBRT (.pbrt)')

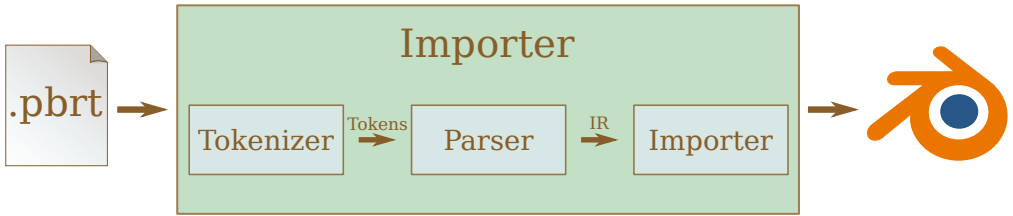
def register():
    """Register the addon with Blender."""
    bpy.types.TOPBAR_MT_file_import.append(menu_func_import)

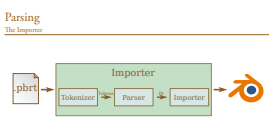
    # ...

As for the menu entry, It is actually one of the easier things to add:

When registering the extension, we simply append an operator to this Blender object, which is called when we click the chosen menu entry.

Parsing
The Importer

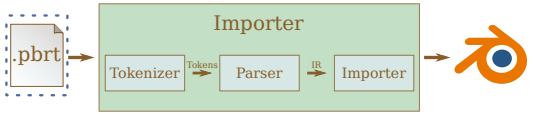




Moving on to the actual importer: Here we actually start digging into the files themselves, and for that, I typically break the importing into a couple of different steps.

The Operator

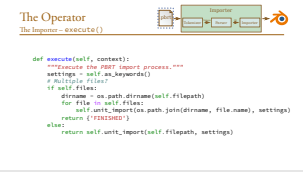
The Importer – `execute()`




```python
def execute(self, context):
    """Execute the PBRT import process."""
    settings = self.as_keywords()
    # Multiple files?
    if self.files:
        dirname = os.path.dirname(self.filepath)
        for file in self.files:
            self.unit_import(os.path.join(dirname, file.name), settings)
        return {'FINISHED'}
    else:
        return self.unit_import(self.filepath, settings)
```


ECON24

LUND UNIVERSITY



The first thing we want to do is simply to ask Blender to start the process by implementing the `execute` function.

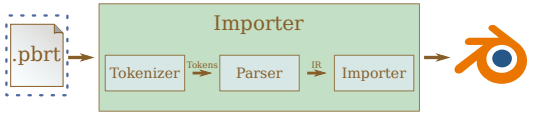
Here, we do run into a bit of a caveat though: Blender will behave slightly differently depending on if we only select one file, or whether we select several:

- For one file, the full path to it ends up in the `filepath` property.
- Otherwise, the basename for each file ends up in the `files` property list.

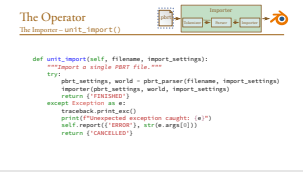
In either case though, we typically use the same parsing method for each file: called `unit_import` in this case.

The Operator

The Importer – `unit_import()`




```python
def unit_import(self, filename, import_settings):
    """Import a single PBRT file."""
    try:
        pbrt_settings, world = pbrt_parser(filename, import_settings)
        importer(pbrt_settings, world, import_settings)
        return {'FINISHED'}
    except Exception as e:
        traceback.print_exc()
        print(f"Unexpected exception caught: {e}")
        self.report({'ERROR'}, str(e.args[0]))
        return {'CANCELLED'}
```

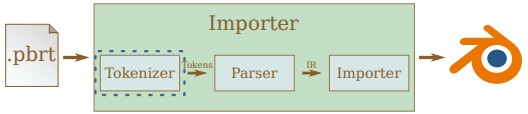



As for the singular import, it really only does two things: parse the file, and create Blender objects from the intermediate representation.

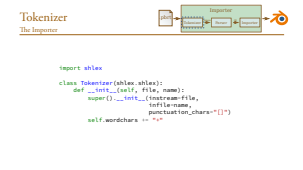
I did however find that it was useful to create a catch-all exception handler here. Blender will typically catch stray exception to avoid crashing the interface, but I kept losing the full stack-trace anyhow. Thus, with this `traceback` call I ended up saving myself from a lot of debugging time.

Tokenizer

The Importer




```python
import shlex

class Tokenizer(shlex.shlex):
    def __init__(self, file, name):
        super().__init__(instream=file,
                         infile=name,
                         punctuation_chars="[]")
        self.wordchars += "+"
```

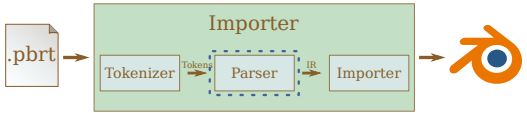



With that were inside the actual parsing infrastructure, starting with the `Tokenizer`. Typically, I wouldn't go into details about the parsing stuff, but I am particularly happy about how easy Python makes this: You only really need 5 lines to get a complete one, depending on how you want to count *this*, and this includes things such as error reporting and the ability to add new sub-files to the token-stream, which we need for the `Include` statements!

This is of course powered by the built-in Python module: `shlex`, which is typically used for parsing command-line arguments but also works really well for more general parsing tasks.

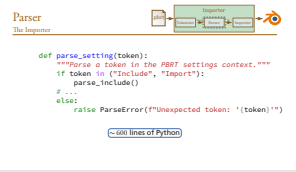
Parser

The Importer




```python
def parse_setting(token):
    """Parse a token in the PBRT settings context."""
    if token in ("Include", "Import"):
        parse_include()
    # ...
    else:
        raise ParseError(f"Unexpected token: '{token}'")
```

~ 600 lines of Python

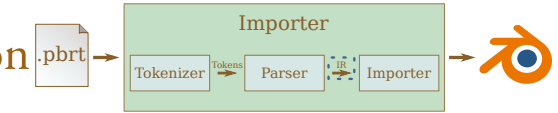


Then, from the tokens the parser builds an intermediate representation, although, I hope I will not disappoint too much, as I will omit basically everything of the parser itself.

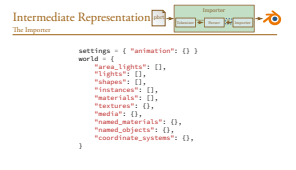
Suffice to say, it is not much more than a bunch of functions with a lot of `if-else` statements to handle all possible tokens and states that can be encountered in a file, totaling about 600 lines Python. Not too bad for a parser that is able to parse all PBRT example scenes from all versions of PBRT!

Intermediate Representation

The Importer




```
settings = { "animation": {} }
world = {
    "area_lights": [],
    "lights": [],
    "shapes": [],
    "instances": [],
    "materials": [],
    "textures": {},
    "media": {},
    "named_materials": {},
    "named_objects": {},
    "coordinate_systems": {},
}
```

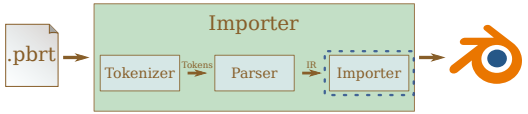



When the parser is done the `intermediate representation` that looks something like this:

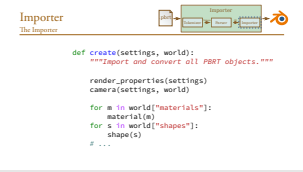
- Higher level objects, such as rendering parameters are gathered in the settings dictionary, and,
- The world objects are collected in another one, separating out shapes, instances, lights, materials, and any other *scene* specific object.

Importer

The Importer




```python
def create(settings, world):
    """Import and convert all PBRT objects."""

    render_properties(settings)
    camera(settings, world)

    for m in world["materials"]:
        material(m)
    for s in world["shapes"]:
        shape(s)
    # ...
```

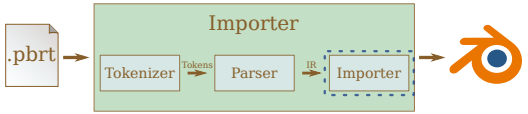



And the last thing we need to do then is simply to convert this representation to Blender objects, which is what happens in the *actual* importer part.

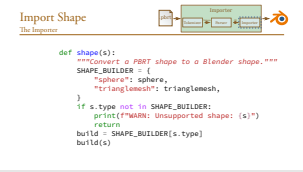
And that part is not particularly fancy: It applies the appropriate rendering and camera properties, then simply loops over each of the quantities that should be converted from PBRT to Blender objects.

Import Shape

The Importer




```python
def shape(s):
    """Convert a PBRT shape to a Blender shape."""
    SHAPE_BUILDER = {
        "sphere": sphere,
        "trianglemesh": trianglemesh,
    }
    if s.type not in SHAPE_BUILDER:
        print(f"WARN: Unsupported shape: {s}")
        return
    build = SHAPE_BUILDER[s.type]
    build(s)
```

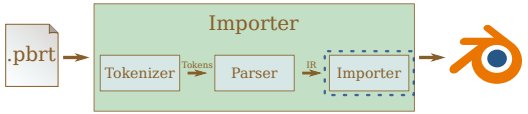



And just to show an example, when we import a single shape I typically do it like *this*. In short, I create a mapping over all shapes currently supported by the importer to a dedicated function that create that specific shape.

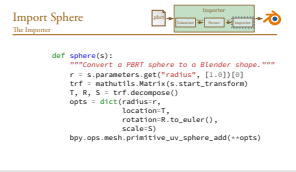
This way I can quite easily add support for new shapes as necessary by simply adding new import functions.

Import Sphere

The Importer




```python
def sphere(s):
    """Convert a PBRT sphere to a Blender shape."""
    r = s.parameters.get("radius", [1.0])[0]
    trf = mathutils.Matrix(s.start_transform)
    T, R, S = trf.decompose()
    opts = dict(radius=r,
                location=T,
                rotation=R.to_euler(),
                scale=S)
    bpy.ops.mesh.primitive_uv_sphere_add(**opts)
```

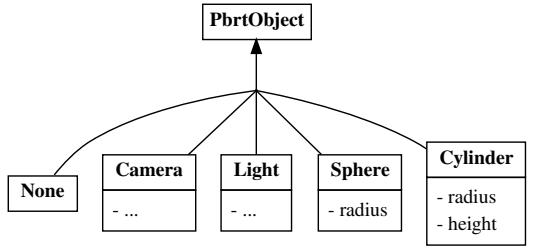



And for a PBRT sphere, we can thus extract the parsed parameters and create a normal Blender object from this, such as a uv sphere in this case.

However, the astute among you probably recognizes though that this doesn't really import a sphere as much as it converts it to a mesh object. So that is what I will address next...

Sum Types and Object Orientation

```
enum PbrtObject
{
    None,
    Camera(...),
    Light(...),
    Sphere{radius: f32},
    Cylinder{radius: f32,
             height: f32},
}
```



Sum Types and Object Orientation

Larger rendering framework, particularly educational ones often arrange supported types in hierarchies, typically using object oriented programming.

And we kind of want to do something similar: We want to tell Blender that this specific object is actually a sphere, with this and that property, whereas that other one is a cylinder with *these* other properties, etc.

In some programming languages such as Rust, this is effectively captured using something called sum-types, or more simply `enums`. And we kind of would like something similar for properties in Blender.

Faking it in Blender – 1

```python
class PbrtSphere(bpy.types.PropertyGroup):
    """PBRT Sphere properties."""

    radius: bpy.props.FloatProperty(
        name="radius",
        description="The sphere's radius.",
        default=1.0,
    )
```


Faking it in Blender – 1

And as far as I know, there is no direct support for something like that, but it is relatively easy to fake it with a bit of Python meta-programming.

First, we would start with creating a property class, such as this one for a sphere.

Faking it in Blender – 2

```
class PbrtShape(PropertyGroup):
    type: EnumProperty(
        name="type",
        items=[("none", "None", ""),
               ("camera", "Camera", ""),
               ("light", "Light", ""),
               ("sphere", "Sphere", ""),
               ("cylinder", "Cylinder", ""), ...],
        default="none")
    none: BoolProperty(name="none", default=False)
    camera: PointerProperty(type=PbrtCamera)
    light: PointerProperty(type=PbrtLight)
    sphere: PointerProperty(type=PbrtSphere)
    cylinder: PointerProperty(type=PbrtCylinder)
```


Faking it in Blender – 2

Then we create another class to represent the entire hierarchy. Which in this case would include cameras, lights and other objects, but most crucially, we have an enumeration of these different options, allowing us to differentiate between them and store various properties in each sub-type.

And this is where we get back to our imported sphere: By using some *other* kind of object as a proxy, we can see and manipulate it instead, letting it contain things such as transforms, but also keeping all unique object properties around as well.

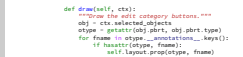
Thus, when we import the sphere, we still create a `uv` sphere, but we *also* mark it is a sphere, and add the unique sphere properties to it: That is, the radius in this case.

(Probably not ideal from a storage perspective, as we would have to store everything for all objects, but seems to work okay in my prototype at least!)

Selecting the Derived Property

```python
def draw(self, ctx):
    """Draw the edit category buttons."""
    obj = ctx.selected_objects
    otype = getattr(obj.pbrt, obj.pbrt.type)
    for fname in otype.__annotations__.keys():
        if hasattr(otype, fname):
            self.layout.prop(otype, fname)
```

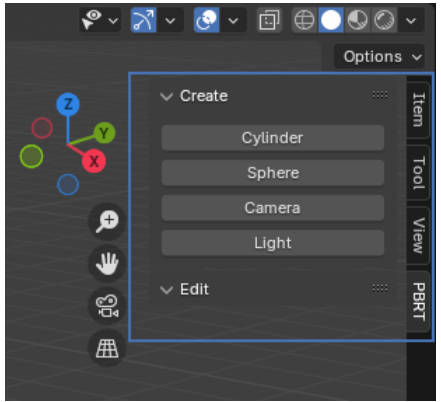

Selecting the Derived Property



Furthermore, we can use a bit of Python meta-programming to do other tasks with this approach, such as drawing different panel layouts for different types of objects.

We simply grab the `type` property first, then trawl over the properties for that sub-type, which themselves can be found in this slightly weird (`__annotations__`) double-underscored (dunder) attribute.

Proxy Object Panel



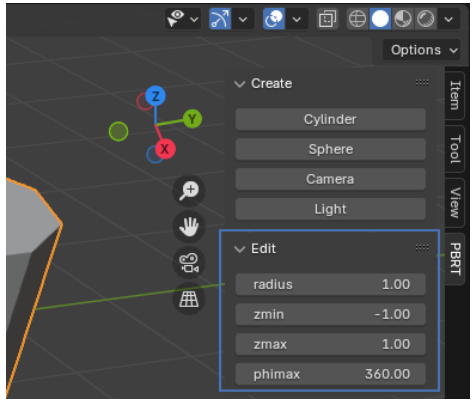
Proxy Object Panel

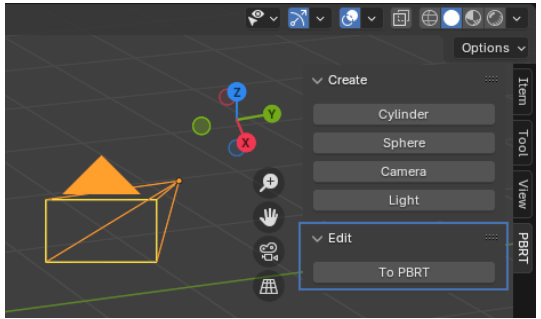
So in my prototype, I use this technique to add a side-bar for adding and manipulating PBRT properties:

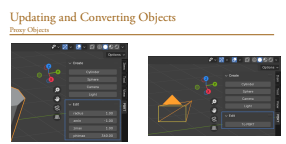
In it, I can create new objects with PBRT attributes already set…

Updating and Converting Objects

Proxy Objects







…And also detect if the selected object is an active PBRT proxy, and then display or update those properties in this interface. Further, for regular Blender object, there is the option to *convert* them!

(Unfortunately, I haven't come up with a good mechanism for *updating* how the object looks in the UI just yet, e.g., if you change the sphre radius, but that may be possible with some kind of listener object, but that is outside the scope of this talk.)

Render Properties

```
class RenderPBRT(bpy.types.RenderEngine):
    bl_idname = ""
    bl_label = ""

    # ...
```

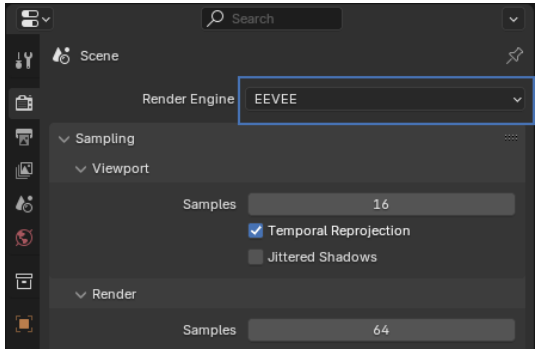
```
class RenderPBRT4(RenderPBRT):
    bl_idname = "PBRT4"
    bl_label = "PBRT4"

class RenderPBRT3(RenderPBRT):
    bl_idname = "PBRT3"
    bl_label = "PBRT3"

class RenderPBRT2(RenderPBRT):
    bl_idname = "PBRT2"
    bl_label = "PBRT2"
```

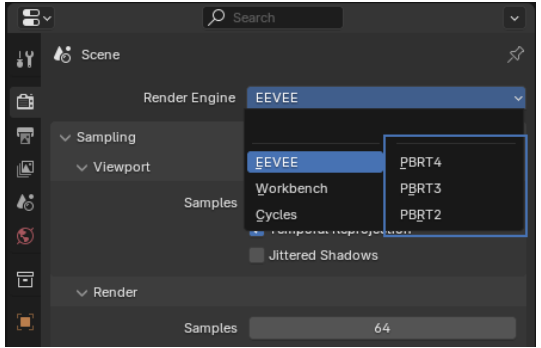

The same mechanism is also used to add some support for rendering properties.

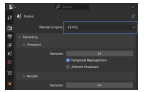
This talk is mostly about importers and exporters, but as I mentioned before, the PBRT format is a combined scene and rendering format, so we really need some notion of rendering properties anyway.

Thankfully, it is actually really straightforward to add new renderers to contain such properties: We only need a class that derives from the Blender `RenderEngine` type, although in my case I wanted three different ones: One for each version of PBRT I intend to support.

Render Properties







Thus, when those are registered, we get three new drop-down entries in our Rendering menu: One for each version.

Render Properties

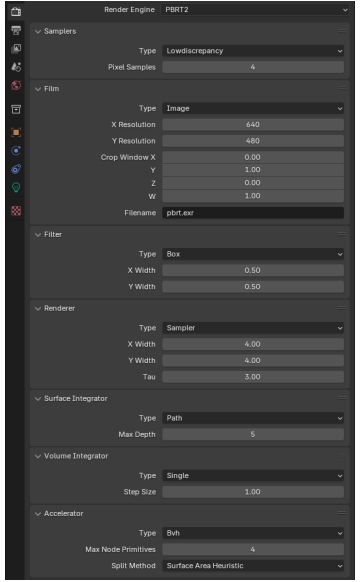
Panel Example

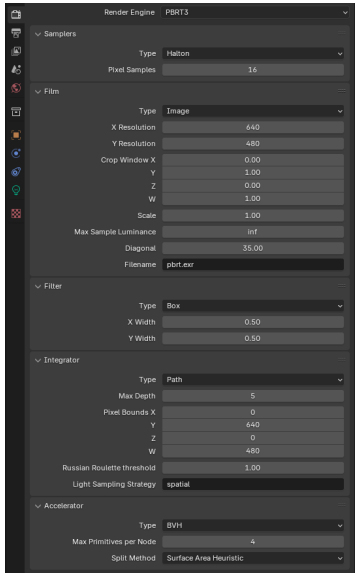
```python
class PbrtV4Accelerator(PbrtProperty):
    """PBRT Accelerator properties."""

    type: bpy.props.EnumProperty(
        name="Accelerator Type",
        description="Accelerator Type",
        items=[("bvh", "BVH", ""),
               ("kdtree", "Kd-Tree", "")],
        default="bvh",
    )

    bvh: bpy.props.PointerProperty(type=PbrtV4BvhAccelerator)
    kdtree: bpy.props.PointerProperty(type=PbrtV4KdTreeAccelerator)
```

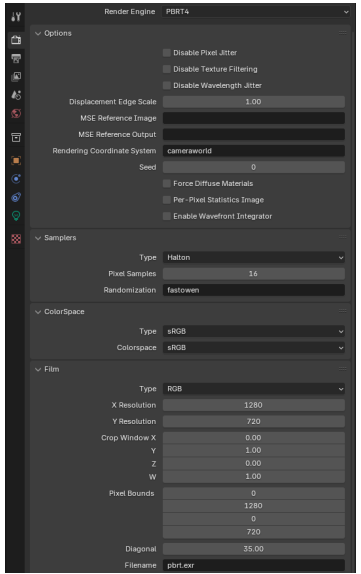

Render Properties
Panel Example
class PbrtV4Accelerator(PbrtProperty):
"""PBRT Accelerator properties."""
type: bpy.props.EnumProperty(
name="Accelerator Type",
description="Accelerator Type",
items=[("bvh", "BVH", ""),
("kdtree", "Kd-Tree", "")],
default="bvh",
)
bvh: bpy.props.PointerProperty(type=PbrtV4BvhAccelerator)
kdtree: bpy.props.PointerProperty(type=PbrtV4KdTreeAccelerator)

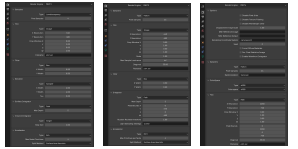
That was the easy part though. After that, you unfortunately need to create separate panels and property classes for each property you want to control.

Here's an example of this for the `Accelerator` property in PBRT version 4 and an example panel that adapts to the different property types, just as for the proxy objects.









Unfortunately for me though, there are still loads of these properties and panels, which was quite tedious to create, but should *hopefully* be done now, barring bugs of course.

The Exporter

The Exporter
Exporter
.pbrt

So, that was all about the proxies and render properties, which leaves us with the exporter, which arguably is one of the easier parts.

Earlier Work

The Exporter

- https://github.com/giuliojiang/pbrt-v3-blender-exporter (2.79)
- https://github.com/stig-atle/io_scene_pbrt (2.8x)
- https://github.com/NicNel/bpbrt4 (2.9, Windows)

And in fact, there already exists quite a few PBRT exporters out there, although, they are all probably due for an update, judging from their `README`s and currently advertised Blender version support.

But, that's no reason to harp on them. Just that they exist is a great boon, and makes for a good starting point when continuing the work, even if I am starting from scratch.

Back to the Exporter

What do we need?

- An Export *Operator*[1]
- An Export Panel[2] (optional)
- An Export Menu Entry[3] (optional)

[1] https://docs.blender.org/manual/en/latest/interface/operators.html
[2] https://docs.blender.org/api/current/bpy.types.Panel.html
[3] https://docs.blender.org/api/current/bpy.types.Menu.html

Back to the Exporter
What do we need?
Exporter
.pbrt
- An Export *Operator*[1]
- An Export Panel[2] (optional)
- An Export Menu Entry[3] (optional)
[1] https://docs.blender.org/manual/en/latest/interface/operators.html
[2] https://docs.blender.org/api/current/bpy.types.Panel.html
[3] https://docs.blender.org/api/current/bpy.types.Menu.html

So, what do we *actually* need to create an exporter?

Basically, we need exactly the same things as we do for the importer:

- An operator to do actual export, and then, optionally,
  - a panel and a menu entry.

The Operator

The Exporter


```python
class ExportPBRT(bpy.types.Operator, bpy_extras.io_utils.ExportHelper):
    """Blender Exporter class for PBRT scenes."""
    # ... Config variables...

    def draw(self, context):
        """Specify how to draw the Blender panel UI."""

    def execute(self, context):
        """Execute the PBRT export process."""
```


And the operator is very similar to our importer, the main difference is that instead of deriving from the `ImportHelper`, we derive from the `ExportHelper`.

After that, we still need a set of configuration variables, a `draw` method if we want a panel, and an `execute` method to actually drive the operator.

The Operator

The Exporter – Configuration Variables


```python
class ExportPBRT(bpy.types.Operator, bpy_extras.io_utils.ExportHelper):
    """Blender Exporter class for PBRT scenes."""
    bl_idname = "export_scene.pbrt"
    bl_label = "Export PBRT"
    bl_options = {'PRESET'}
    filename_ext = ".pbrt"
    filter_glob: StringProperty(default="*.pbrt;*.pbrt.gz")
    mode: EnumProperty()
    compress: BoolProperty()
```


ECEN24

LUND UNIVERSITY

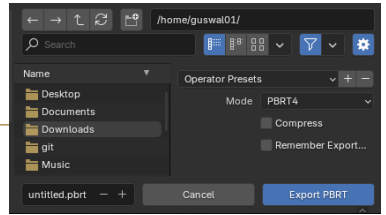
For the configuration variables, it is very similar: We need an identifier to call the operator using the `bl_idname` variable.

The main difference is that we don't have to worry about multiple files this time: Once started, the intended export location can be found in the `filepath` property, which we don't even have to create explicitly thanks to the inheritance hierarchy here.

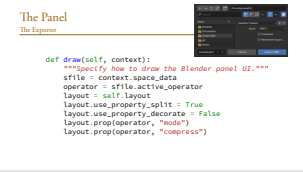
(We can of course still export to multiple files, but that is probably better handled manually instead.)

The Panel

The Exporter




```python
def draw(self, context):
    """Specify how to draw the Blender panel UI."""
    sfile = context.space_data
    operator = sfile.active_operator
    layout = self.layout
    layout.use_property_split = True
    layout.use_property_decorate = False
    layout.prop(operator, "mode")
    layout.prop(operator, "compress")
```

The draw panel, again, works the same, you either add it directly as a draw method in the operator class, or create a separate one for it. Giving us a panel that looks something like what you can see up here.

The Operator

The Exporter – `execute()`


```python
def execute(self, context):
    """Execute the PBRT export process."""
    with open(self.filepath, "w", encoding="utf-8") as f:
        with contextlib.redirect_stdout():
            export_camera(self.mode, context)
            export_renderer(self.mode, context)
            export_world(self.mode, context)
    return {"FINISHED"}
```


And of course, we need to actually do something when we export, which we handle by implementing the `execute` function.

And here, you are simply given a file-path, which you should use to populate with your scene and render data. How you do it however, is entirely up to you. I happen to like to use regular print statements by temporarily redirecting `stdout`, but to each their own.

The Operator

The Exporter – `export_renderer()`


```python
def export_renderer(mode, ctx):
    """Export PBRT renderer settings."""
    prop = getattr(ctx.scene.PbrtRenderProperties, mode)
    prop.to_pbrt()
```


As for *actually* printing the various settings: Here's a short example of how I do it:

1. First, I simply extract the rendering properties, which of course, may differ somewhat depending on which version of PBRT is actually being used.
2. Then, I call this `to_pbrt` convenience method.

The Operator

The Exporter – to_pbrt()


```
def to_pbrt(self):
    """Print a PBRT representation of this object."""
    ptype = getattr(self, self.type)
    print(f"{self.NAME} \"{self.type}\"")
    for name in ptype.__annotations__.keys():
        prop = ptype.rna_type.properties[name]
        val = getattr(ptype, name)
        ntype = pbrt_type(val, prop)
        val = pbrt_value(val, prop)
        print(f"\"{ntype} {name}\" [{val}]")
```


And that method, is *this*, which I suspect is a bit hard to read here, but in short it uses the same basic method that I used for the proxy and render objects, but this time to print out the properties in the correct PBRT format.

This of course means that we need to perform a bit of type and value translations to match what PBRT expects, but nothing truly major needs to be done here. And, the main reason this is a method is to allow inheritance to specialize the few corner cases that uses a slightly different format.

The Operator

The Exporter – `export_world()`


```
def export_world(engine, ctx):
    """Export the PBRT world objects."""
    print("WorldBegin")
    for obj in bpy.data.objects:
        if obj.type == "LIGHT":
            export_light(obj, engine, ctx)
        else:
            export_shape(obj, engine, ctx)
    if engine in ("PBRT2", "PBRT3"):
        print("WorldEnd")
```


And for the world objects themselves, there is, I hope, nothing really unexpected happening here: We simply traverse the list of scene objects and print an appropriate representation for each of them.

The Operator

The Exporter – `export_shape()`


```
def export_shape(obj, mode, ctx):
    """Export a PBRT shape."""
    print("AttributeBegin")
    print("    Transform %s" % (get_transform(obj)))
    print("    Material ..." % (get_material(obj)))
    if obj.pbrt.type != "none":
        export_pbrt_object(obj, mode, ctx)
    elif obj.type == "MESH":
        # ...
    print("AttributeEnd")
```


The Operator
The Exporter – export_shape()
Exporter
.pbrt
def export_shape(obj, mode, ctx):
"""Export a PBRT shape."""
print("AttributeBegin")
print(" Transform %s" % (get_transform(obj)))
print(" Material ..." % (get_material(obj)))
if obj.pbrt.type != "none":
export_pbrt_object(obj, mode, ctx)
elif obj.type == "MESH":
# ...
print("AttributeEnd")

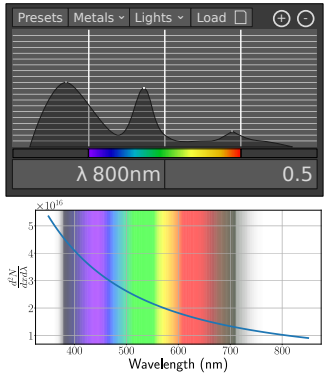
Which, for a shape looks something like *this*. Which, really is just:

1. Print an `AttributeBegin` and `End` wrapper to avoid material and transforms from affecting other objects, then
2. Print out the world transform and material for the object, and,
3. Check if the object has been marked as a proxy, and if so, handle it specially and print that object the same as for the rendering property,
4. Otherwise, convert the regular Blender objects to something PBRT can recognize, such as meshes, etc.

And that is pretty much everything I have to say about exporters.

Missing Feature(s)

- Spectral Properties



Missing Feature(s)
• Spectral Properties

The last thing I wanted to bring up is something I haven't yet found a good way of solving: Namely spectral properties.

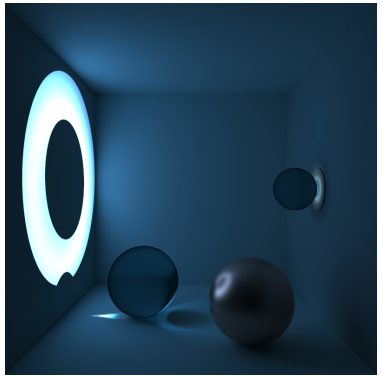
In short, the RGB colors we use are really just a very efficient hack to represent material properties, but they have some real limitations when it comes to representing things in a physically correct way: In reality, how light interacts with a material can differ a lot depending on what part of the spectrum the light actually contain.

Right now, I am simply doing a lossy conversion to RGB for everything, since I couldn't find a practical way of working with spectral data in Blender. Obviously, I could simply keep the original spectra somewhere for PBRT objects, but that makes them impractical to modify.

I figured it would be nice to have some kind of UI widget for spectra like this, similar to curves: Allowing us pick some presets from metals or lights, and then allow us to modify the responses accordingly. But that is just me musing on what would be nice for working with spectral properties in Blender, and not an actual suggestion.

Conclusion

1. PBRT
2. Importers / Exporters
3. Render Properties and Proxies



Conclusion
1. PBRT
2. Importers / Exporters
3. Render Properties and Proxies

Which brings me to the end of this talk, and just to quickly summarize things: During this presentation I've talked about…

- PBRT; what it is, and what kind of renderings it can do, along with its file-format, and,
- Briefly shown how to create custom importers and exporters for Blender, using PBRT as an example, as well as,
- How to add rendering properties to influence these, and,
- How to use proxy objects in Blender to represent PBRT objects.

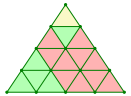
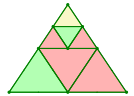
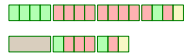
Thanks for Listening!

Questions and Answers

- Thanks for listening!
- Questions and Answers

Thanks for Listening!
Questions and Answers
• Thanks for listening!
• Questions and Answers

And with all that, I think it's about time to conclude the presentation, and I guess open up for any other kind of questions and, I hope, some answers!

Making Blender ♡ PBRT
Create your own Importers and Exporters

- https://gustafwaldemarson.com/
  - https://gustafwaldemarson.com/pages/publications/
- gustaf.waldemarson@cs.lth.se
- gustaf.waldemarson@arm.com

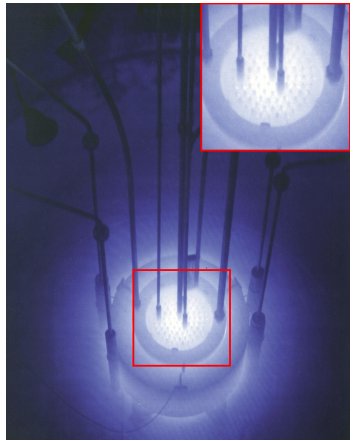
LUND UNIVERSITY

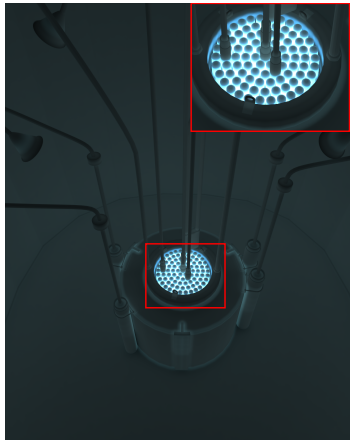
Links / Contact
Making Blender ♡ PBRT
Create your own Importers and Exporters
• https://gustafwaldemarson.com/
– https://gustafwaldemarson.com/pages/publications/
• gustaf.waldemarson@cs.lth.se
• gustaf.waldemarson@arm.com

Before I end: You can (eventually) find links to most of the material related to this work on my homepage, and of course, if anyone has more questions about PBRT, or even glTF, Vulkan ray-tracing or micromaps that I talked about last year, feel free to contact me on any of these emails, or just chat me up at some point during the conference!

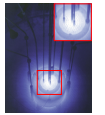
The End

Superluminal Reactor Rendering





Superluminal Reactor Rendering

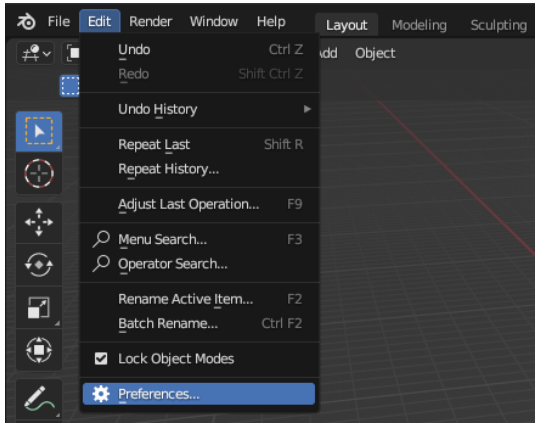


Testing Pipeline

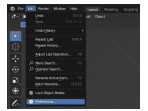
Testing Pipeline

Intermission: Installing and Enabling Addons

Blender <= 4.1



Intermission: Installing and Enabling Addons
Blender <= 4.1

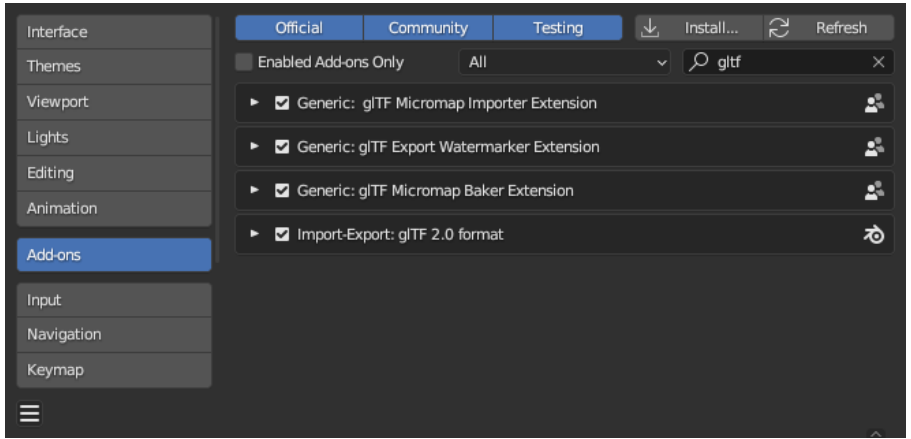


And now is a good point for this little reminder for completeness's sake: I typically install a new by dropping the file into the addon directory, but forget to enable it, and then sit around scratching my head for a bit wondering where all my stuff has gone.

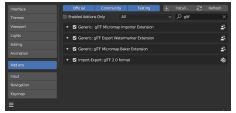
So this is easily fixed done by opening the *Edit*-menu, and going to *Preferences* dialogue.

Intermission: Installing and Enabling Addons

Blender <= 4.1



Intermission: Installing and Enabling Addons
Blender <= 4.1

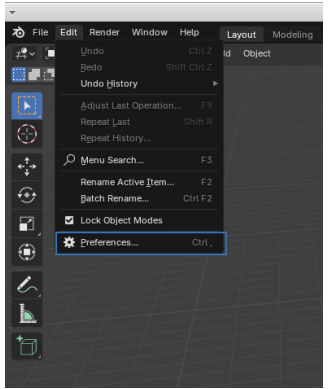


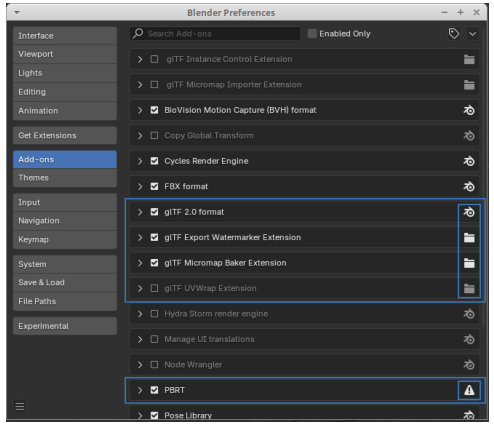
Here, we open the *Add-ons* section. Then I recommend enabling all sections and searching for "gltf". Then simply click *enable* to make the add-on usable.

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.

Intermission: Installing and Enabling Addons

Blender 4.2+ (Legacy Addons)





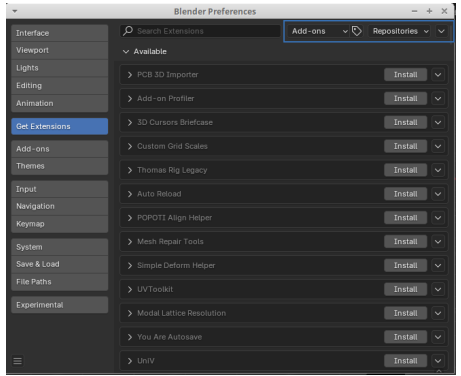
Intermission: Installing and Enabling Addons
Blender 4.2+ (Legacy Addons)

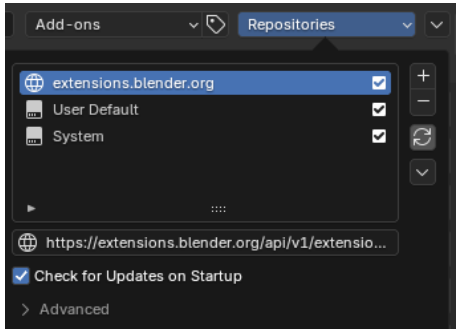
However, in Blender 4.2, the addons have changed somewhat: If I understand correctly, they have been split up into 'addons' for old-school ones, and extensions for new properly updated ones.

Legacy addons are installed just as before: Simply add the files in the appropriate directory and enable it among the preferences.

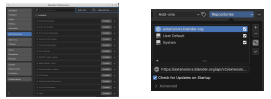
Intermission: Installing and Enabling Addons

Blender 4.2+ (Extensions)





Intermission: Installing and Enabling Addons
Blender 4.2+ (Extensions)



New extensions however are found under a new tab, appropriately called *extensions*. Here, addons can be downloaded and enabled directly over the internet, at least when Blender has been given sufficient permissions to do so.

(It is also possible to add more repositories from here.)

It is my understanding that this will be the way addons should be developed going forward, as it makes a lot of things easier under the hood, but I suggest you watch the talk about extensions by Nika Kutsniashvili to learn more about this.

References I

[1] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 4th. MIT Press, 2023.

[2] Gustaf Waldemarson. *Handling Custom Data in glTF Files with Exporter/Importer Plugins*. The Blender Foundation, Youtube. 2023. URL: https://youtu.be/4fBGM8qc21M?t=1783.

[3] Gustaf Waldemarson and Michael Doggett. "Photon Mapping Superluminal Particles". In: *Eurographics 2020 - Short Papers*. Ed. by Alexander Wilkie and Francesco Banterle. The Eurographics Association, 2020. ISBN: 978-3-03868-101-4. DOI: 10.2312/egs.20201004.

[4] Gustaf Waldemarson and Michael Doggett. "Succinct Opacity Micromaps". In: *Proc. ACM Comput. Graph. Interact. Tech.* 7.3 (Aug. 2024). DOI: 10.1145/3675385. URL: https://doi.org/10.1145/3675385.