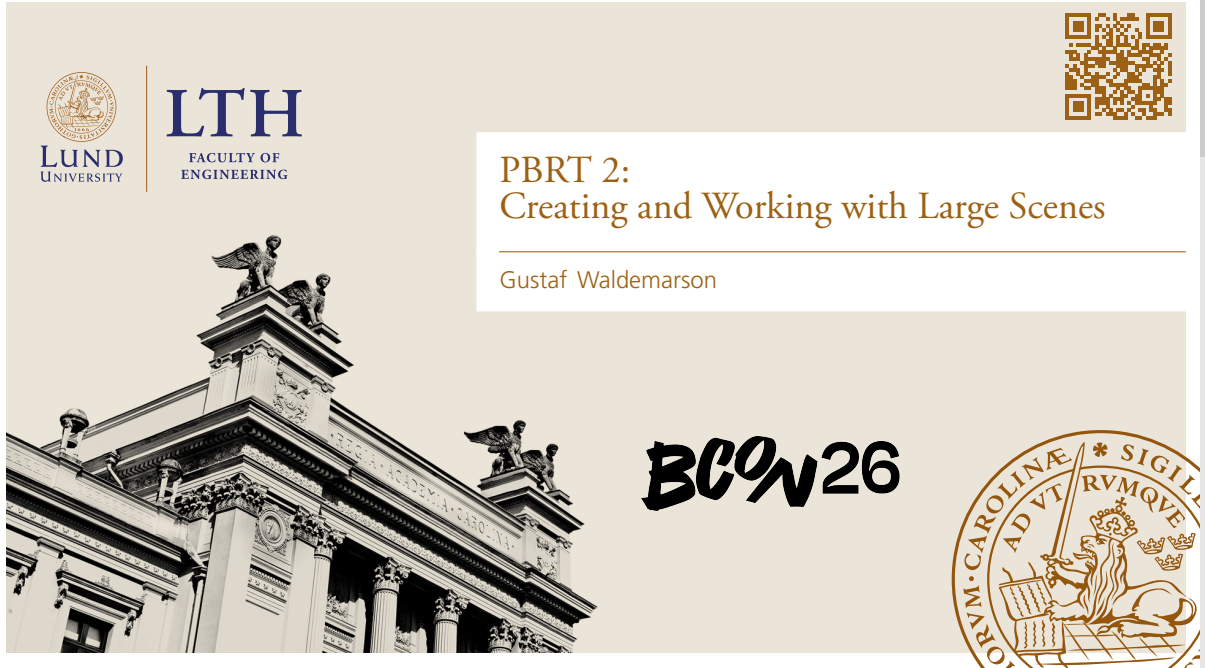


2026-10-22



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

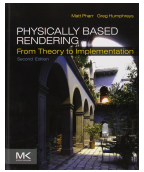
Thanks everyone for coming! Nice to see so many stick around for some of the later technical talks!

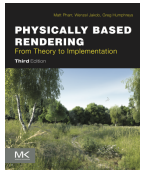
Who am I?

- (Former) Industrial PhD Student at the Lund University Graphics Group
- Compiler Performance Engineer at Arm

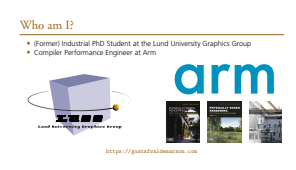












But first, I think some kind of introduction is in order: My name is Gustaf Waldemarson, a now *former* industrial PhD student from the Lund University Graphics Group, as I finally graduated this summer, having worked with various ray-tracing related topics, and particularly those related to something known as *micromaps*, but that is a topic for another time!

Now though, I am currently working as a GPU Compiler Performance Engineer at Arm, basically trying to make shaders run faster.

But with that, I am obliged to say that I do not represent Arm during this event, nor is the content sponsored by them. As such, any views or opinions herein are entirely my own.

Similarly, I am not affiliated with the people behind `PBRT` either. It just so happens that it is something that I've used quite a bit during my PhD, and even a bit before it.

Agenda

1. What is PBRT?
   - BCON24
   - Renders: PBRT vs Cycles
2. Creating Scenes
3. Large Scenes and Interactivity
4. Curves, Materials, and Volumes
5. Limitations



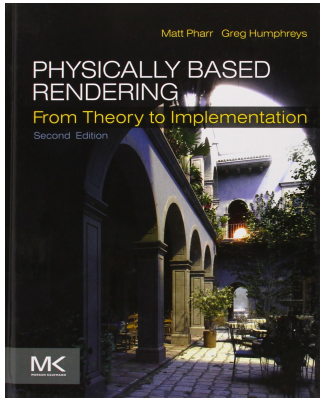
Agenda
1. What is PBRT?
   – BCON24
   – Renders: PBRT vs Cycles
2. Creating Scenes
3. Large Scenes and Interactivity
4. Curves, Materials, and Volumes
5. Limitations

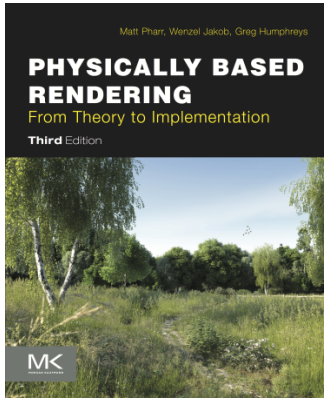


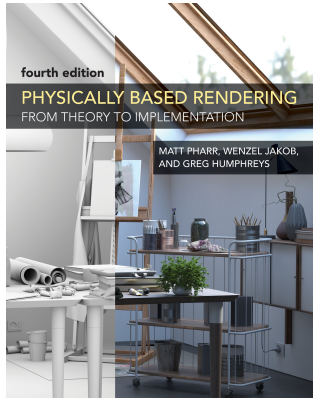
But with that, let's start the talk proper shall we? In this talk I will discuss a number of things:

1. First of all, I will *briefly* present `PBRT`. The rendering framework and identically named scene file format.
   - And the keen-eyed among you might have noticed the "2" in the title: This talk *lightly* builds upon the work I presented at BlenderCon 2024, even if the topics in this talk are mostly tangential to that work. Still, I will include a short recap of its content just in case!
2. From there, I will move into the *new* content: Namely how to efficiently create Blender scenes and objects with the Python API, particularly for scenes with thousands of objects and instances.
3. And, as some of you may know; throwing that many objects at the Blender viewport is a sure-fire way to lock up the interface and make it very hard to work with the scene in general. Thus, I include some tips for alleviating this problem.
4. After that, I will move on to discuss some slightly more esoteric things, such as the challenges to reconstruct curve objects, volumes and even a little about how material shader graphs are constructed with the Python API.
5. Finally, I will mention some of the limitations with this tooling.

What is PBRT?







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

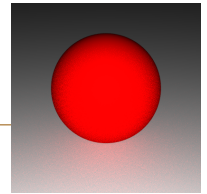


So let us begin with that aforementioned recap; starting with *PBRT* itself!

In full, it is an acronym for *Physically Based Ray Tracer*, but the acronym is often used to refer to both the award winning text-book, and the rendering framework they describe, both of which are still used in education and research for developing new, or improving existing graphics algorithms.

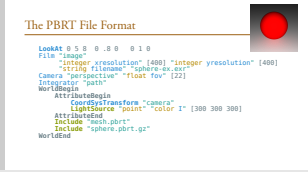
The framework is also open-source with a very liberal license, and has consequently been quite influential over the last few decades. And the last two editions have even been made freely available online, making them an excellent place for learning about all things ray-tracing.

The PBRT File Format



```
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
```


The PBRT File Format

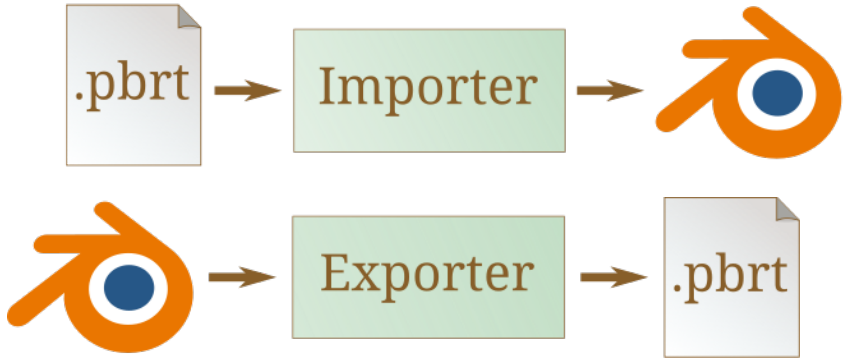


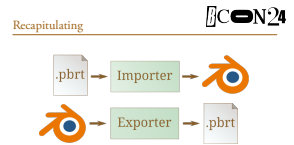
The acronym is not *limited* to the book and framework however; it also refers to a text-based file format for scenes that can be rendered with these frameworks.

In short, this file describes both *how* the scene should be rendered, and *what* the scene, or its *world* contains.

As an example, the simple scene file shown on this slide creates a sphere with a base plane, and renders it with a path tracer, creating the image shown up here to the right.

Recapitulating

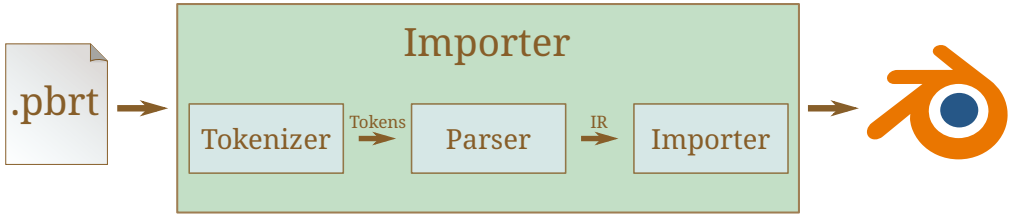


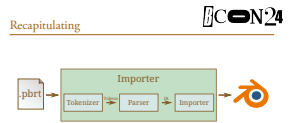


So, back in 2024, I presented how to create tools to read any kind of user-defined file formats and use that to create a Blender scene, as well as doing the opposite: Taking an existing Blender scene and exporting *that* to another format.

Naturally, I used the `PBRT` format as an example to showcase this.

Recapitulating

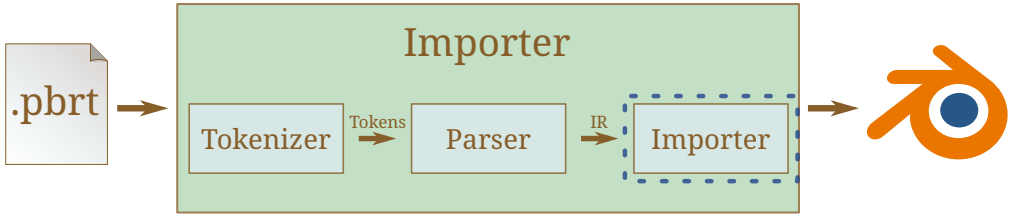


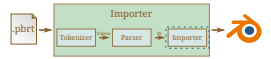


Admittedly though, back in 2024 this framework was just barely functional: The parser and tokenizers were fully implemented and working, but the importer, the part that creates the *actual* geometry that Blender will be working with was *very* barebone. Only a handful of object types were implemented, and there were quite a few bugs to be found as well, particularly related to the camera orientation.

Technically that was fine back then, as the focus was on how to create the Blender infrastructure around importers and exporters, not necessarily to showcase all the fancy scenes that `PBRT` is typically known for.

What's New?





That said, it was *my* goal to eventually use *those* scenes in various other projects. So while the tool this not complete by any stretch of the imagination, it is now considerably more fleshed out.

Consequently, *this* time around, the intention is to dive a bit deeper into the importer and show how various objects are created inside Blender with the Python API and how this is done efficiently when the scene grows larger.

Some Results — I



"Landscape" 358 Meshes, 407 824 Instances, 26 050 093 Triangles

Some Results — I



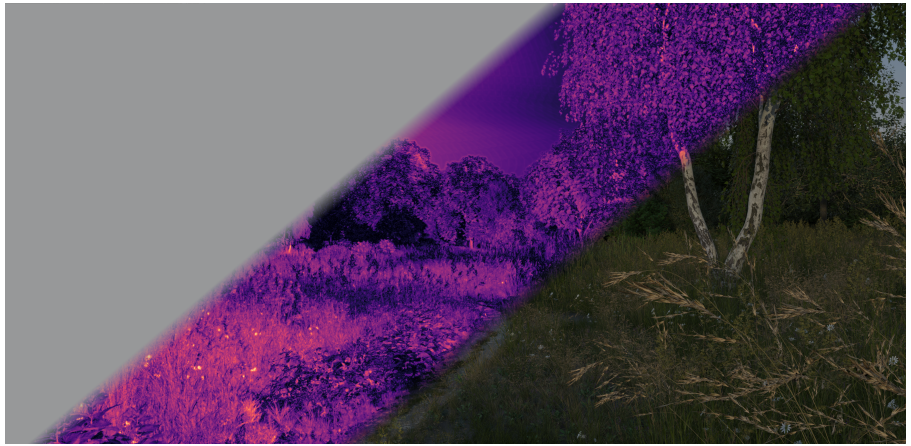
"Landscape" 358 Meshes, 407 824 Instances, 26 050 093 Triangles

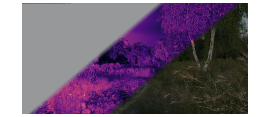
And just to give you a small preview of the scenes that are now supported by this tool: Here's the so-called "Landscape" scene that was prominently featured for the third edition of PBRT, but of course, now renderered with *Cycles*. All in all, it is a very large outdoor scene, that only really uses around 300 unique meshes, but with a *very* large number of instances, giving us a un-instanced tally of around 26 million triangles.

Today though, this is probably not *that* big for an off-line renderer, but it *is* already too large for Blender to work with *interactively*: My machines all drop down below 1 fps in the viewport if I open this scenes as-is, making any subsequent `GUI` operations very hard to perform.

Consequently, I will present a technique I use to ensure that I can still work with these scenes.

Some Results — II





Additionally, the scenes that are imported with this tool are close enough to the original `PBRT` renders that it is actually possible to compare them:

Here, the left-most slice is the original `PBRT` version, and the right-most from `Cycles`, with the so-called ꟻLIP metric error map in the middle; where generally speaking, the brighter it is, the more "incorrect" it is.

Typically, the errors shown in this map come from various issues with my tooling; places where object parameters are unimplemented or simply incorrect, making it an excellent tool for finding bugs.

It is never going to be perfect though: Some rendering differences simply depend on quirks inside `Cycles` and `PBRT`, so a perfect correspondence is unlikely, but generally, the darker the error map is, the better it is.

Starting Off — The Intermediate Representation

```
world = {
    "lights": [...],
    "shapes": [...],
    "textures": [...],
    "materials": [...],
    "named_materials": {...},
    "named_objects": {...},
    "instances": [...],
}
```


Starting Off — The Intermediate Representation
world = {
"lights": [...],
"shapes": [...],
"textures": [...],
"materials": [...],
"named_materials": {...},
"named_objects": {...},
"instances": [...],
}

Moving on to the new content then! How do we actually start creating these scenes with the Python API?

First, let us assume that we are already given some representation of the scene or "world" in the form of a nested Python dictionary. Our job is to convert *that* to objects that Blender understands and can work with.

Creating the Scene — Overview

```python
material_map = {}
for index, material in enumerate(world["materials"]):
    material_map[index] = material.factory(material)
for s in world["shapes"]:
    shape.factory(s, material_map)
for al in world["area_lights"]:
    area_lights.factory(al)
for ls in world["lights"]:
    lights.factory(ls)

set_cycles_render_settings(pbrt_settings, world)
```


Creating the Scene — Overview

```python
material_map = {}
for index, material in enumerate(world["materials"]):
    material_map[index] = material.factory(material)
for s in world["shapes"]:
    shape.factory(s, material_map)
for al in world["area_lights"]:
    area_lights.factory(al)
for ls in world["lights"]:
    lights.factory(ls)

set_cycles_render_settings(pbrt_settings, world)
```

At the highest level, we “simply” want to go over all of these objects, and convert them to their closest Blender equivalent, which, obviously, is easier said than done.

But the first thing I want to highlight is the order of operations: Each shape always has a material, both on the `PBRT` and Blender side, so we really want to create those first so that each new Blender object can have its corresponding Blender material set directly.

Although, for this presentation, I’ll wait a little with the materials as those are somewhat more complicated to create.

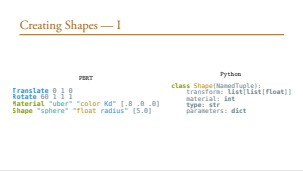
Creating Shapes — I

PBRT

```
Translate 0 1 0
Rotate 60 1 1 1
Material "uber" "color Kd" [.8 .0 .0]
Shape "sphere" "float radius" [5.0]
```

Python

```
class Shape(NamedTuple):
    transform: list[list[float]]
    material: int
    type: str
    parameters: dict
```

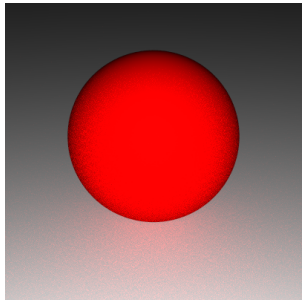



Starting out with the shapes: On the **PBRT** side, we are given a textual representation like this that describes a transform, material, and what type of the shape it is, along with an arbitrary amount of shape-specific parameters.

This I translate into a Python class with a transform matrix, an index to a material object, a string for the type, and a dictionary for all other parameters.

Creating Shapes — II

```python
def shape(s, import_settings):
    SHAPE_BUILDER = {
        "sphere": sphere,
        "trianglemesh": trianglemesh,
        "plymesh": plymesh,
        # ...
    }
    build = SHAPE_BUILDER[s.type]
    return build(s, import_settings)
```



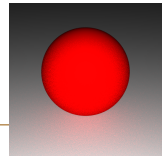
Creating Shapes — II

def shape(s, import_settings):
    SHAPE_BUILDER = {
        "sphere": sphere,
        "trianglemesh": trianglemesh,
        "plymesh": plymesh,
        # ...
    }
    build = SHAPE_BUILDER[s.type]
    return build(s, import_settings)



Then, I simply map that shape type to a dedicated import function that does all the heavy lifting for creating the *geometry* of the Blender object.

Creating Shapes — III — Sphere



```python
def sphere(s, import_settings):
    """Convert a PBRT sphere to a Blender shape."""
    radius = s.parameters.get("radius", [1.0])[0]
    kwargs = dict(radius=radius)
    bl_op = bpy.ops.mesh.primitive_uv_sphere_add(**args)
    return obj
```


Creating Shapes — III — Sphere

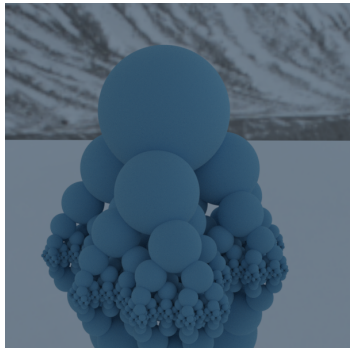


def sphere(s, import_settings):
    """Convert a PBRT sphere to a Blender shape."""
    radius = s.parameters.get("radius", [1.0])[0]
    kwargs = dict(radius=radius)
    bl_op = bpy.ops.mesh.primitive_uv_sphere_add(**args)
    return obj

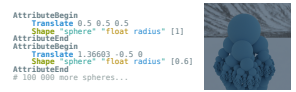
As an example, the sphere simply extracts the relevant parameters, such as the radius, with an appropriate default, and constructs it somehow, such as with primitive operators that already exist in Blender.

Blender Operators Can Be Slow!

```
AttributeBegin
    Translate 0.5 0.5 0.5
    Shape "sphere" "float radius" [1]
AttributeEnd
AttributeBegin
    Translate 1.36603 -0.5 0
    Shape "sphere" "float radius" [0.6]
AttributeEnd
# 100 000 more spheres...
```

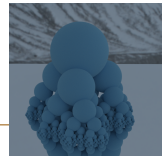


Blender Operators Can Be Slow!



However, calling operators in this fashion can be quite slow, since they are often plumbed into other things, such as the undo/redo system, so they are *usually* best used sparingly for long running operations, such as those that constructs full scenes.

Caching — I



```python
def cached_creation(name, operation, arguments, cache):
    """Create a new object or copy from the cache."""
    key = tuple(arguments.items())
    if key in cache:
        me_copy = cache[key].data.copy()
        return bpy.data.objects.new(name, me_copy)
    operation(**arguments)
    obj = bpy.context.active_object
    obj.name = name
    cache[key] = obj
    return obj
```


Caching — I



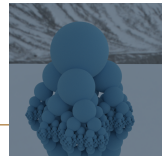
def cached_creation(name, operation, arguments, cache):
    """Create a new object or copy from the cache."""
    key = tuple(arguments.items())
    if key in cache:
        me_copy = cache[key].data.copy()
        return bpy.data.objects.new(name, me_copy)
    operation(**arguments)
    obj = bpy.context.active_object
    obj.name = name
    cache[key] = obj
    return obj

And, there is a relatively simple way to manage that, using caching: Simply put, each time we want to create an object, we first check if we have already created it before with exactly the same parameters…

If that is the case, we then make a copy of that prior object, renaming it as necessary.

Otherwise, if it is the first time we are creating it, we call the operators and then store it in the cache, preparing it for the next time.

Caching — II



```python
def sphere(s, cache, import_settings):
    """Convert a PBRT sphere to a Blender shape."""
    radius = s.parameters.get("radius", [1.0])[0]
    args = dict(radius=radius)
    bl_op = bpy.ops.mesh.primitive_uv_sphere_add
    obj = cached_creation("Sphere", bl_op, args, cache)
    return obj
```

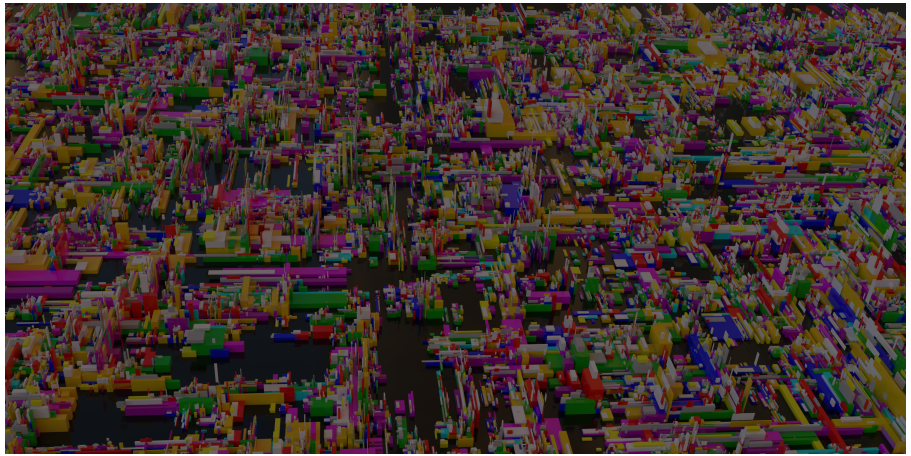

Caching — II



def sphere(s, cache, import_settings):
    """Convert a PBRT sphere to a Blender shape."""
    radius = s.parameters.get("radius", [1.0])[0]
    args = dict(radius=radius)
    bl_op = bpy.ops.mesh.primitive_uv_sphere_add
    obj = cached_creation("Sphere", bl_op, args, cache)
    return obj

And just to show how this caching mechanism is used: We simply replace the last two lines with the Blender operator we should call, and pass it along to the caching function with an appropriate name for our new object.

Instances — I



"City Env" — PBRTv2 — 641 754 Meshes

Instances — I



"City Env" — PBRTv2 — 641 754 Meshes

However, reducing the operator calls is really just a band aid if your scene literally contains hundreds of thousands of meshes, such as in this fake city environment scene where almost all meshes are duplicated many times over. To handle that, we need to turn our attention to *instances*.

Instances — II



PBRT

```
ObjectBegin "Mesh.000"
    Shape "sphere"
ObjectEnd

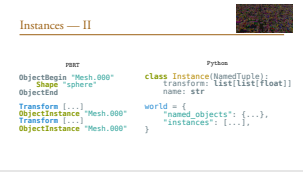
Transform [...]
ObjectInstance "Mesh.000"
Transform [...]
ObjectInstance "Mesh.000"
```

Python

```
class Instance(NamedTuple):
    transform: list[list[float]]
    name: str

world = {
    "named_objects": {...},
    "instances": [...],
}
```


Instances — II



In PBRT, an *instance* is a mapping of a string name, to a group of shapes that can later be *instantiated* with different transforms, which can be used cheaply replicate a single object multiple times in the scene.

Similar to the Shape, this is very straight-forward to represent in Python:

- The groups are stored in a separate entry in the world, and
- The instances simply use a transform and string to refer to these groups.

Instances — III



```python
collection_map = {}
for group_name, shapes in world["named_objects"].items():
    group = bpy.data.collections.new(group_name)
    collection_map[group_name] = group
    for s in shapes:
        group.objects.link(shape(s, material_map))

for trf, group_name in world["instances"]:
    instance = bpy.data.objects.new(group_name, None)
    instance.instance_type = "COLLECTION"
    instance.instance_collection = collection_map[group_name]
    instance.matrix_world = mathutils.Matrix(trf)
```


Instances — III



collection_map = {}
for group_name, shapes in world["named_objects"].items():
group = bpy.data.collections.new(group_name)
collection_map[group_name] = group
for s in shapes:
group.objects.link(shape(s, material_map))

for trf, group_name in world["instances"]:
instance = bpy.data.objects.new(group_name, None)
instance.instance_type = "COLLECTION"
instance.instance_collection = collection_map[group_name]
instance.matrix_world = mathutils.Matrix(trf)

And interestingly enough, this grouping maps very nicely to how instances are created in Blender:

In short, we need to create a new "collection" for each of the instance-groups, add all the associated shapes to that collection, and finally create the Blender instance objects as you can see at bottom: Where the `instance_type`, `COLLECTION`, and transform are set.

Instance Conversion



```
Transform [ 20 0 0 0 0 20 0 0 0 0 .01 0 0 0 -1 1 ]
Shape "trianglemesh" "integer indices" [ 0 1 2 2 3 0 ]
Shape "trianglemesh" "integer indices" [ 0 1 2 2 3 0 ]
Shape "trianglemesh" "integer indices" [ 0 1 2 2 3 0 ]
```



```
ObjectBegin "Mesh.001"
  Shape "trianglemesh" "integer indices" [ 0 1 2 2 3 0 ]
ObjectEnd
Transform [ 20 0 0 0 0 20 0 0 0 0 .01 0 0 0 -1 1 ]
ObjectInstance "Mesh.001"
ObjectInstance "Mesh.001"
ObjectInstance "Mesh.001"
```

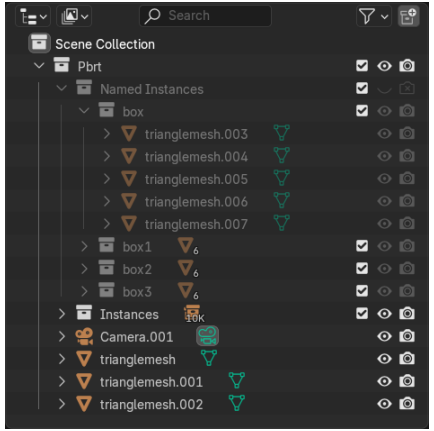

Instance Conversion
Transform [20 0 0 0 0 20 0 0 0 0 .01 0 0 0 -1 1]
Shape "trianglemesh" "integer indices" [0 1 2 2 3 0]
Shape "trianglemesh" "integer indices" [0 1 2 2 3 0]
Shape "trianglemesh" "integer indices" [0 1 2 2 3 0]
ObjectBegin "Mesh.001"
Shape "trianglemesh" "integer indices" [0 1 2 2 3 0]
ObjectEnd
Transform [20 0 0 0 0 20 0 0 0 0 .01 0 0 0 -1 1]
ObjectInstance "Mesh.001"
ObjectInstance "Mesh.001"
ObjectInstance "Mesh.001"
"City Env" (PBRTv2) 66 Meshes, 641 814 Instances (Originally 641 754 Meshes)

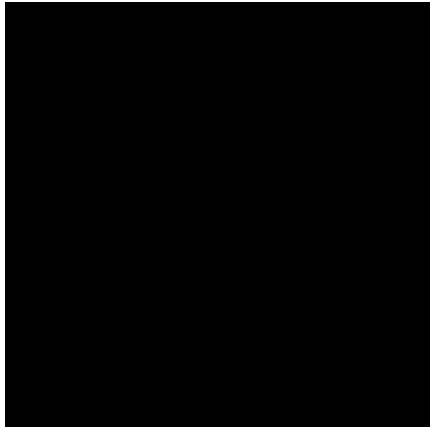
Another thing one can do is combine caching with instances: Whenever the same shape is created multiple times but with different transforms, convert them to instances with the corresponding transform.

Here, I illustrate that concept using two *PBRT* code snippets. Of course, this is actually done on the intermediate Python representation instead, but the effect is ultimately the same.

As an example, doing this on our *City Environment* scene from before drops it down to just 66 meshes, but with 641 814 instances, making it much more manageable.

Arranging the Scene in Collections — I





Arranging the Scene in Collections — I

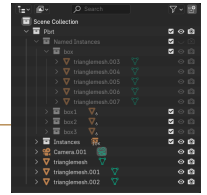




"ArcSphere" (PBRTv3) 0 Meshes, 0 Instances.

And while we are on the topic of collections; as most people know, they are useful not only used for instances, but are also useful for keeping the scene itself tidy.

Arranging the Scene in Collections — II



```
pbrt_col = bpy.data.collections.new("Pbrt")
bpy.context.scene.collection.children.link(pbrt_col)
instances_col = None
named_col = None
if world["named_objects"]:
    named_col = bpy.data.collections.new("Named Instances")
    pbrt_col.children.link(instance_col)
if world["instances"]:
    instance_col = bpy.data.collections.new("Instances")
    pbrt_col.children.link(named_instance_col)
```


Arranging the Scene in Collections — II

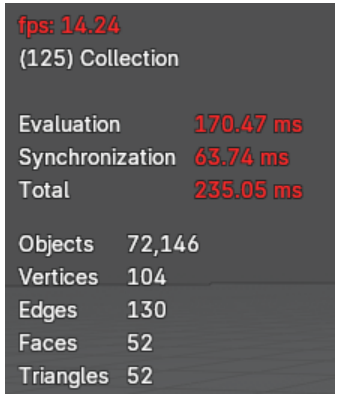


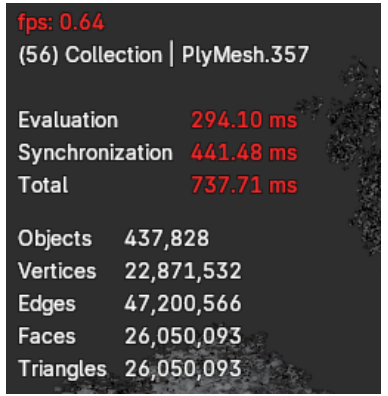
pbrt_col = bpy.data.collections.new("Pbrt")
bpy.context.scene.collection.children.link(pbrt_col)
instances_col = None
named_col = None
if world["named_objects"]:
named_col = bpy.data.collections.new("Named Instances")
pbrt_col.children.link(instance_col)
if world["instances"]:
instance_col = bpy.data.collections.new("Instances")
pbrt_col.children.link(named_instance_col)

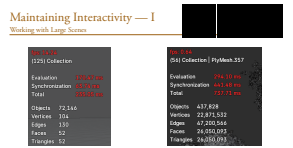
And this works almost exactly like for instances: We first create a new top-level collection as before called "Pbrt", but now also link it to the current scene context. Then any subsequent collection that we add is linked into that one, giving us this arrangement.

Maintaining Interactivity — I

Working with Large Scenes







Now though, we are running into the problem I mentioned in the beginning though: Once there's enough objects visible in the viewport, be they meshes or instances, our interactive performance is going to suffer.

The only way I know to work around this is to start hiding objects, and for here I mainly choose to hide the instances, as they tend to be much more numerous.

But when you do this, you typically also want to hide them in some *uniform* way. E.g., if you want to hide 50 % of all instances, you typically don't want to hide all objects in one isolated corner of the scene; you want them spread out over more or less the entire scene.

Maintaining Interactivity — II

Working with Large Scenes

```
ins_ratio = settings.get("viewport_hiding", 0.0)
ins_counts = defaultdict(float)
ins_totals = defaultdict(int)
for trf, group_name in world["instances"]:
    ins_counts[group_name] += ins_ratio
    ins_totals[group_name] += 1
    cnt = ins_counts[group_name]
    tot = ins_totals[group_name]
    if tot > 1 and cnt >= 1.0:
        for vl in bpy.context.scene.view_layers:
            instance.hide_set(True, view_layer=vl)
        ins_counts[group_name] -= 1.0
```


Maintaining Interactivity — II

Working with Large Scenes

```
ins_ratio = settings.get("viewport_hiding", 0.0)
ins_counts = defaultdict(float)
ins_totals = defaultdict(int)
for trf, group_name in world["instances"]:
    ins_counts[group_name] += ins_ratio
    ins_totals[group_name] += 1
    cnt = ins_counts[group_name]
    tot = ins_totals[group_name]
    if tot > 1 and cnt >= 1.0:
        for vl in bpy.context.scene.view_layers:
            instance.hide_set(True, view_layer=vl)
        ins_counts[group_name] -= 1.0
```

There a number of ways to accomplish this; the simplest would be to use random numbers: If a sample between 0 and 1 is below your target ratio, hide it.

I do not like non-determinism however, so I used a slightly more complicated scheme is outlined on this slide:

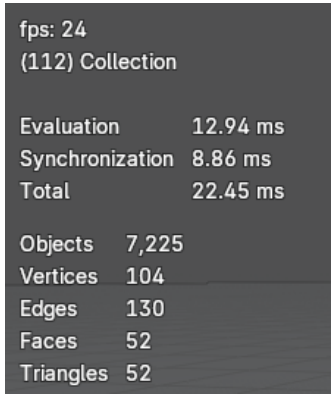
Basically, each group tracks a ratio of the currently hidden instances: Once that ratio goes above one, the current instance is hidden, and the ratio is reset to 0.

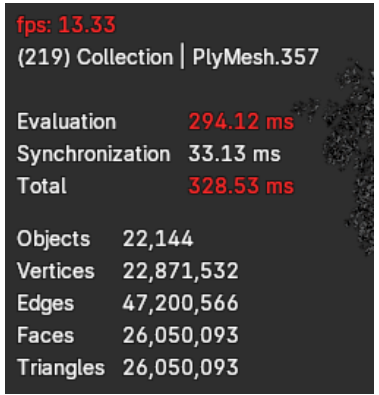
(The only caveat being: Always show single instances.)

Maintaining Interactivity — III

Working with Large Scenes

[redacted]





Hiding 90 % and 95 % of all instances respectively.

BCON26

LUND UNIVERSITY

Maintaining Interactivity — III
Working with Large Scenes



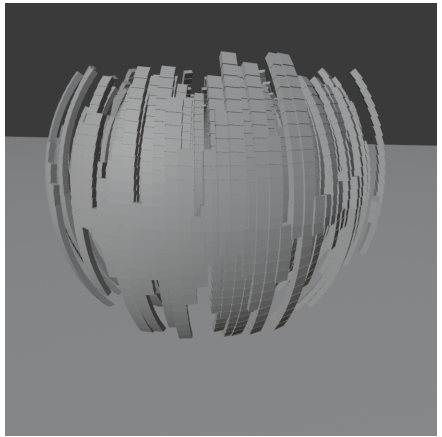


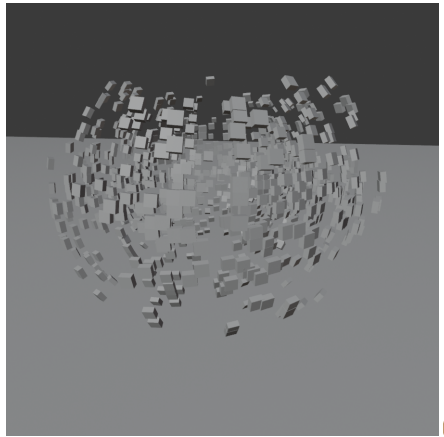
Hiding 90 % and 95 % of all instances respectively.

As you can see here: When I hide between 90–95 % of all instances in these scenes the viewport performance doubles for the *ArcSphere* and goes all the way up to around 13 fps for *Landscape*; making it much more usable from the sub 1 fps from before.

Maintaining Interactivity — IV

Working with Large Scenes



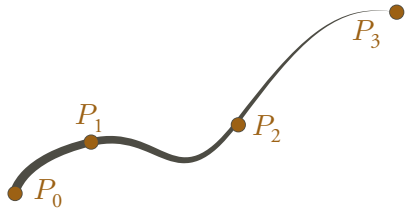


Maintaining Interactivity — IV
Working with Large Scenes
Hiding 90 %

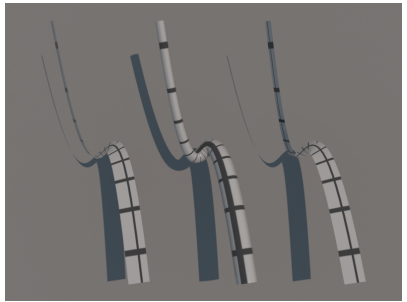
And just to illustrate what doing this kind of culling looks like: To the left is the original scene, and on the right, 90 % of all instances are hidden from the viewport in a uniform way.

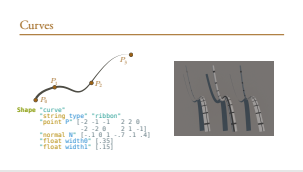
(I should note that in the past I have seen some strange effects from doing this, e.g., exporting the scene to `glTF` and then re-importing that file due to some `depsgraph` quirks, but I was not able to reproduce that anymore, so might not be an issue any longer, or maybe hiding via viewport layers work differently.)

Curves




```
Shape "curve"
      "string type" "ribbon"
      "point P" [-2 -1 -1   2 2 0
                 -2 -2 0    2 1 -1]
      "normal N" [-.1 0 1 -.7 .1 .4]
      "float width0" [.35]
      "float width1" [.15]
```

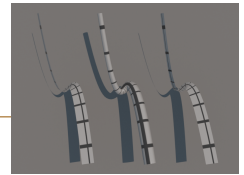


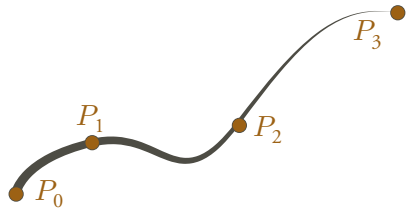


Moving on to some slightly more esoteric objects:

Curves are collective term for many types of objects, but generally speaking is some kind of line traced between a set of points, possibly with varying thicknesses or other parameters, which also makes it suitable for modelling an interesting variety of objects, such as grass, fur, or strands of hair.

Curves

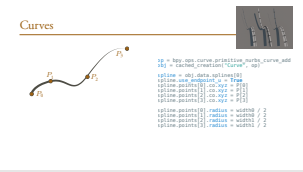





```
op = bpy.ops.curve.primitive_nurbs_curve_add
obj = cached_creation("Curve", op)

spline = obj.data.splines[0]
spline.use_endpoint_u = True
spline.points[0].co.xyz = P[0]
spline.points[1].co.xyz = P[1]
spline.points[2].co.xyz = P[2]
spline.points[3].co.xyz = P[3]

spline.points[0].radius = width0 / 2
spline.points[1].radius = width0 / 2
spline.points[2].radius = width1 / 2
spline.points[3].radius = width1 / 2
```

Curve objects such as these are straight-forward to add in Blender, and belong to the `NURB` family of objects: We simply extract the individual points and widths, and set any other curve specific parameters, which I have omitted here to make things at least somewhat readable.

Curves — Hair

```
Shape "curve"
      "string type" ["cylinder"]
      "point P" [ -0.4 15.3 -1.0
                  -0.4 15.4 -1.0
                  -0.4 15.5 -1.0
                  -0.4 15.5 -1.0 ]
Shape "curve"
      "string type" ["cylinder"]
      "point P" [ -0.4 15.4 -1.0
                  -0.4 15.5 -1.0
                  -0.4 15.5 -1.0
                  -0.4 15.6 -1.1 ]
# ~3_000_000 more curves...
```



Curves — Hair

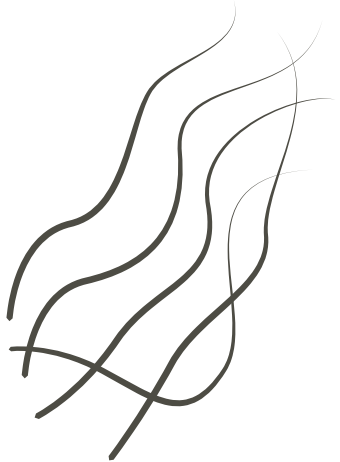


This is all fine, but problem starts to crop when we scale this up to *real* models. As an example, this curvy-hair model contains over 3 million curve objects.

For a renderer, 3 million curves don't really constitute *that* much data, the problem is that Blender wants to be able to model these objects, so it needs way more memory to do so. And we can't de-duplicate them or turn them into instances either: Each curve have unique positions after all.

Curves — Bundling





```
for c in curves:
    P = c.get("P", [])
    width0 = c.get("width0", [width])[0]
    width1 = c.get("width1", [width])[0]

    spline = obj.data.splines.new("NURBS")
    spline.points.add(3)

    spline.points[0].co.xyzw = (*P[0], 1.0)
    spline.points[1].co.xyzw = (*P[1], 1.0)
    spline.points[2].co.xyzw = (*P[2], 1.0)
    spline.points[3].co.xyzw = (*P[3], 1.0)

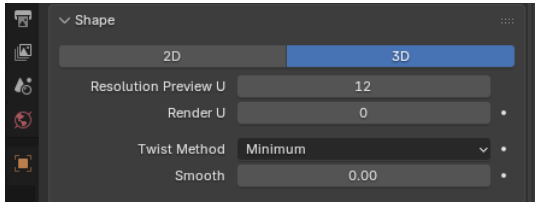
    spline.points[0].radius = width0 / 2
    spline.points[1].radius = width0 / 2
    spline.points[2].radius = width1 / 2
    spline.points[3].radius = width1 / 2
```


Curves — Bundling

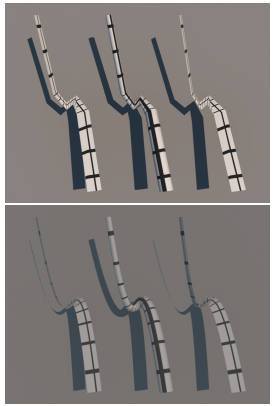
The solution is to do something similar: Each curve object can actually store *many* strands of hair: They don't have to be connected, they only have to be of the same "type".

Thus, in a pre-processing step I go over all curves and group them based on their type, creating what I call "curve bundles".

Curves — Resolution




```
obj.data.resolution_u = opt.get("curve_resolution_u", 1)
obj.data.resolution_v = opt.get("curve_resolution_v", 1)
```



Curves — Resolution



However, it is important to know that when rendering curves they are currently converted to something different that `Cycles` can *actually* render, which at least currently uses quite a lot of memory by default.

We can control this however by changing the `U/V` resolutions for the curve, making them quite a bit cheaper to use memory-wise, which we probably like right now in the middle of the RAM-crisis.

Materials and Textures — I



"Landscape" 258 Materials and 247 Textures

Materials and Textures — I



"Landscape" 258 Materials and 247 Textures

That was shapes and instances, but obviously, we all know that the surface materials of our objects contribute massively to the appearance of our renders.

Unfortunately, it is also what in my opinion, is one of the more difficult aspects of this work. Not because it is a lot of code, but rather because you need to manage multiple things at once.

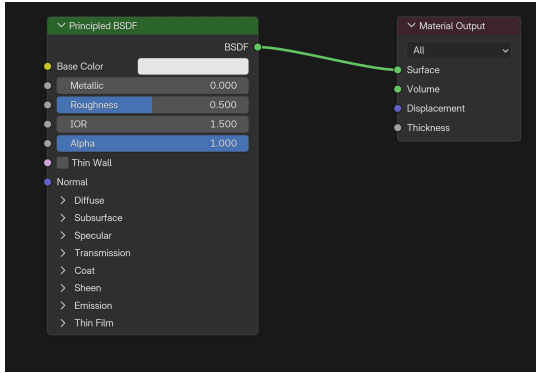
Materials and Textures — Building Graphs — II

```python
bl_mat = bpy.data.materials.new(name="Uber")
bl_mat.use_nodes = True
bl_mat.node_tree.nodes.clear()

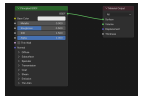
nodes = bl_mat.node_tree.nodes
links = bl_mat.node_tree.links

output = nodes.new("ShaderNodeOutputMaterial")
output.location = (400.0, 0.0)
bsdf = nodes.new("ShaderNodeBsdfPrincipled")
bsdf.location = (0.0, 0.0)

links.new(bsdf.outputs["BSDF"],
          output.inputs["Surface"])
```



Materials and Textures — Building Graphs — II



Actually constructing the material is not *that* hard, but it *is* very verbose. In general, we first create our material, then ask blender to add new nodes to our `node_tree`, then, whenever we want to connect two nodes together, we instead ask it to add a *link* with the name of the two sockets that should be connected.

All in all, the code to the left yields the node-graph to the right.

Materials and Textures — III

PBRT

```
Texture "basecolor"
  "spectrum" "imagemap"
  "string filename"
  ["Abies_concolor_tr_01_co.png"]
Texture "bumpmap"
  "float" "imagemap"
  "string filename"
  ["Abies_concolor_tr_01_co.png"]
Material "uber"
  "float roughness" [ 0.50 ]
  "texture Kd" [ "basecolor" ]
  "texture bumpmap" [ "bumpmap" ]
  "rgb Kr" [ 0.033 0.033 0.033 ]
```

Python

```
class Texture(typing.NamedTuple):
    value: str
    type: str
    parameters: dict

class Material(NamedTuple):
    type: str
    parameters: dict

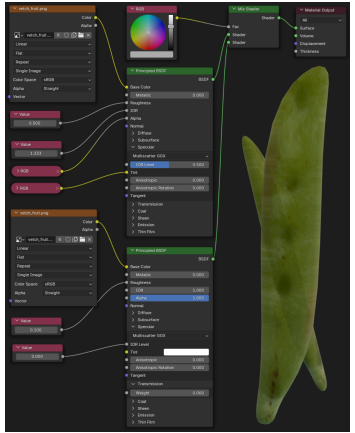
world = {
    "materials": [...],
    "textures": [...],
}
```


Materials and Textures — III

Thankfully, the material systems of **PBRT** and **Blender** are very similar to each other: Each can have multiple different types of materials, and both of them can have a variety of different parameters set, such as scalars, vectors, colors or even textures.

Materials and Textures — Socket Inputs — IV

- Scalars
- Colors
- Textures



Materials and Textures — Socket Inputs — IV

- Scalars
- Colors
- Textures



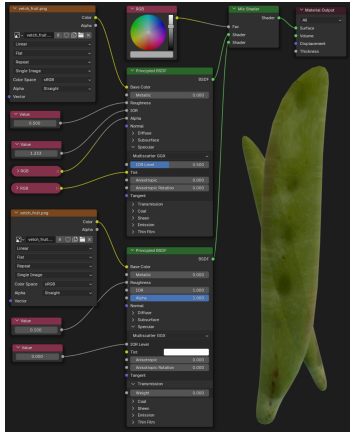
And these parameters map nicely to the Blender: They each map to various “primitive” node types.

(Next slide)

`PBRT` though, similar to `Blender` allows us to *mix* things up, lettings us combine multiple textures, or even multiple materials. But of course, there is already support for node types to do that so that is mostly just a coding issue of adding the appropriate nodes.

Materials and Textures — Socket Inputs — IV

- Scalars
- Colors
- Textures
+ Mixing Textures
+ Mixing Materials



Materials and Textures — Socket Inputs — IV

- Scalars
- Colors
- Textures
+ Mixing Textures
+ Mixing Materials

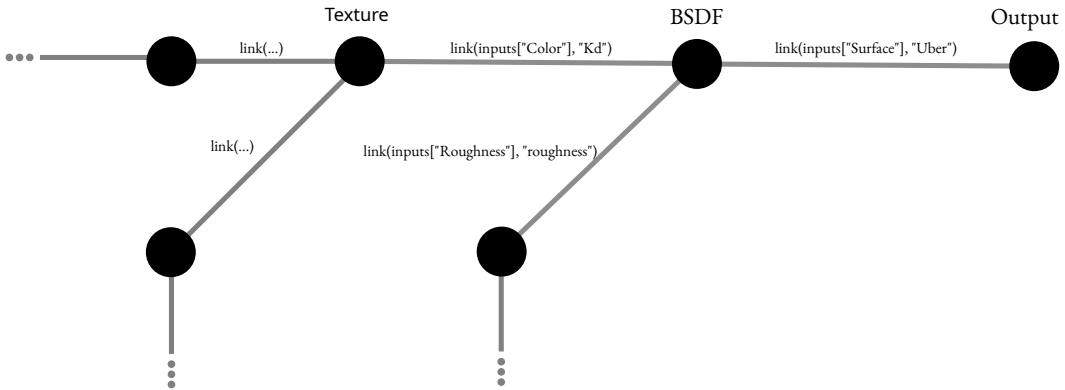


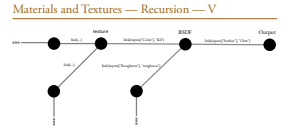
And these parameters map nicely to the Blender: They each map to various “primitive” node types.

(Next slide)

`PBRT` though, similar to `Blender` allows us to *mix* things up, lettings us combine multiple textures, or even multiple materials. But of course, there is already support for node types to do that so that is mostly just a coding issue of adding the appropriate nodes.

Materials and Textures — Recursion — V





This does add a bit of wrinkle though: While we are creating a material; we may *also* want to create *another* material. Which, of course, is best solved with recursion.

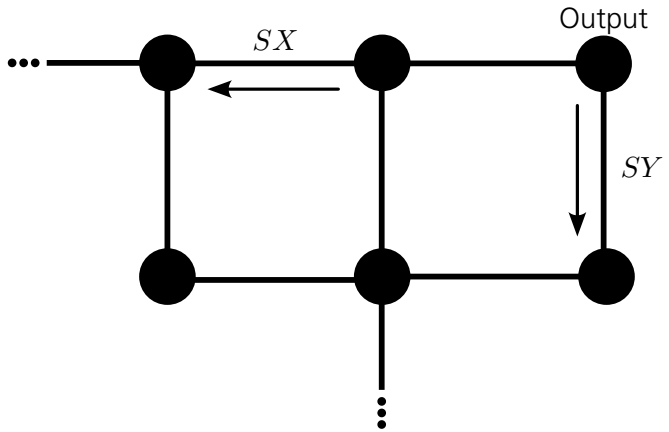
That said, this part was messy and annoying to do for all materials, so I added a very stupid material/node abstraction to handle it, but as I believe some who are here already know, this is probably best converted to the `nodebpy` extension/module that Brady presented yesterday.

Still, the process would be roughly the same, which I sketched out roughly here:

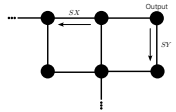
1. First, I create my shader output node, then ask my customized linking function to create a new material, and link that to the "Surface" input socket.
2. Then, when the material is created, for each parameter, it calls that link function, but with the name for the different parameter that it should extract from the `PBRT` side, and which socket it should be connected to on the `Blender` side.

   And of course, when one of these new parameters are detected to be a texture, or a material mixer, this process is repeated recursively.

Materials and Textures — Graphs Layout — VI



Materials and Textures — Graphs Layout — VI



Another complication with the material graph is the *layout*. That is, what does the graph look like when you open it? Basically, the author is allowed to place his or her nodes almost arbitrarily on this 2D plane based on personal taste.

It is actually possible to ignore this aspect entirely, basically letting all nodes end up on-top of each others, or possibly use some kind of auto-layout addon such as `NodeArrange`, but I still created a relatively simple grid system for arranging them basically pushing nodes back backwards or forwards depending on the situation, but again, it seems like `nodebpy` is a much better option for this in the future.

Materials and Textures — Assignment — VII

```
mat = material_map.get(s.material, material_map.get(0))

obj.data.materials.append(mat)
obj.active_material_index = len(obj.data.materials) - 1
obj.active_material = mat
```


Materials and Textures — Assignment — VII

```
mat = material_map.get(s.material, material_map.get(0))

obj.data.materials.append(mat)
obj.active_material_index = len(obj.data.materials) - 1
obj.active_material = mat
```

The last, but oh so important part we need to do, is of course assign the material onto our shapes. And this is by far the easiest part: Since each `PBRT` object can only have one material, we do not have to care about things such as material slots: We simply add our material to the first slot and mark it as the active material.

Volumes



Homogeneous



Heterogeneous

Volumes

Homogeneous

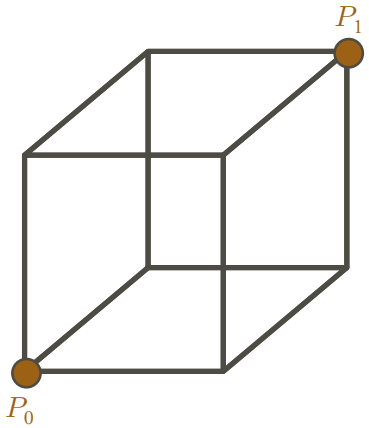
Heterogeneous

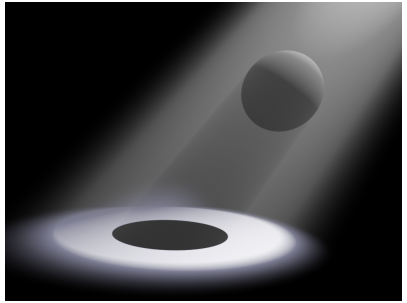
The last Blender object that I want to talk about are *volumes* and *volumetrics*, and from what I've often heard here, often a thorn in the side of many artists, or at least, the engine that has to render them!

In the graphics community we usually call these *participating media*, and they typically comes in one of two different forms:

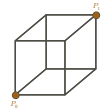
- *Homogenous* for media that should behave the same everywhere within its bounds, and,
- *Hetereogeneous* media, where that behavior is allowed to be variable.

Basic Homogeneous Volumes





Basic Homogeneous Volumes

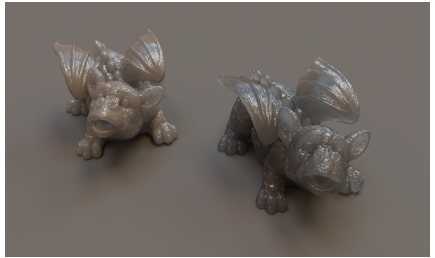




Starting with the homogeneous: The typical `PBRT` volume of this type is defined using two points of an Axis Aligned Bounding Box, with traditional volume properties, such as scattering, absorption and emission. These even map directly to the Blender *Principled Volume* or *Volume Coefficients* shaders, making them very straight-forward represent with a simple cube.

Homogeneous Medium Interfaces

```
MakeNamedMedium "back"
                "string type" "homogeneous"
AttributeBegin
    MediumInterface "back" ""
    Material "glass"
    Shape "plymesh"
          "string filename" "dragon.ply"
AttributeEnd
```



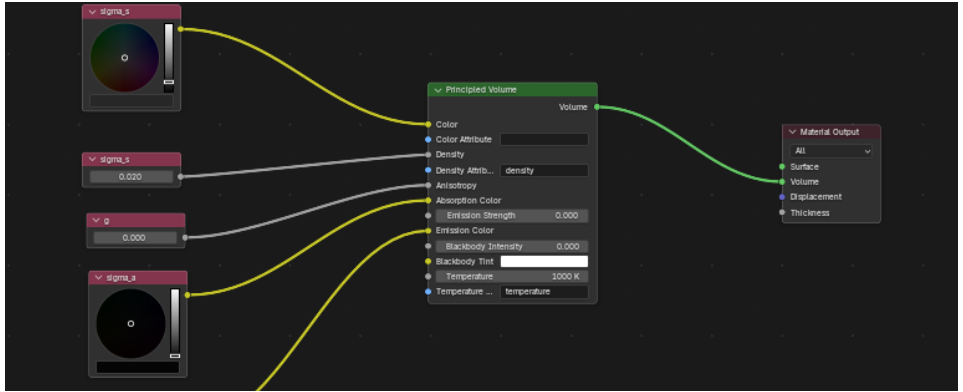
Homogeneous Medium Interfaces

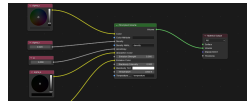


At the slightly more complex end, are *Medium Interfaces*; basically, in `PBRT` all shapes can tell the system what it considers to be the interior and exterior medium.

As far as I know though, it is not possible to set an “exterior” volume on a per-shape basis, so this is something I am currently ignoring, but all objects *can* add volumetric shading to their interior as a part of their material graph output…**(next slide)**

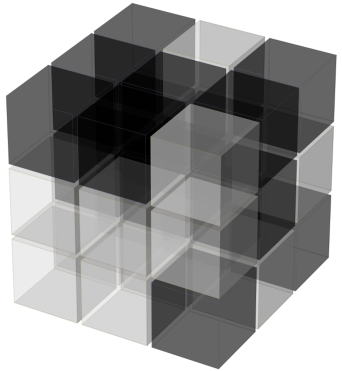
Volume Material

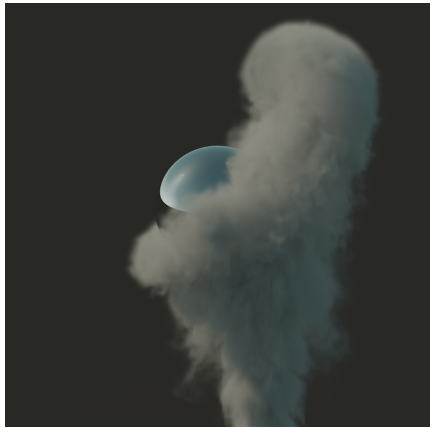




Both the simple cube type and medium inerface are straight-forward to implement in Blender: As you can see, all we have to do is attach volume shading to the shader output in the material graph.

Hetereogeneous Media





Hetereogeneous Media





Obviously, the opposite to homogeneou media, is the hetereogeneous; i.e., media where the "thickness" of the media is not the same everywhere.

Abstractly, it can be thought of as a 3D grid of cells, or *voxels* that contain a value that each represents some particular quantity; typically the "density" of the medium in question. And with enough of these voxels, we can represent complex shapes, such as the cloud here.

In Blender, these objects are almost always represented by a so-called *VDB* object, which can *almost* be thought of as this voxel-grid, but there are a couple of gotchas one should be aware of…**(next-slide)**

VDB Gotchas

Volumes — Voxel Grids

```python
import openvdb
name = "density"
nx = ny = nz = 5
density = np.ones(shape=(nx, ny, nz))
path = bpy.path.abspath(f"//{name}.vdb")
grid = openvdb.FloatGrid()
grid.copyFromArray(density)
grid.name = name
openvdb.write(path, grids=[grid])
bpy.ops.object.volume_import(filepath=path)
```


VDB Gotchas

Volumes — Voxel Grids

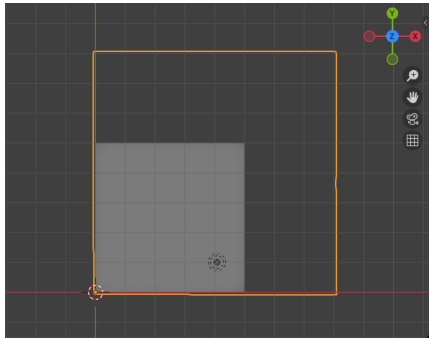
```python
import openvdb
name = "density"
nx = ny = nz = 5
density = np.ones(shape=(nx, ny, nz))
path = bpy.path.abspath(f"//{name}.vdb")
grid = openvdb.FloatGrid()
grid.copyFromArray(density)
grid.name = name
openvdb.write(path, grids=[grid])
bpy.ops.object.volume_import(filepath=path)
```

The built-in Python module `openvdb` allows us to create and manipulate `VDB` volumes, which in the simplest case, as illustrated here, is simply copied from a 3D `numpy` array.

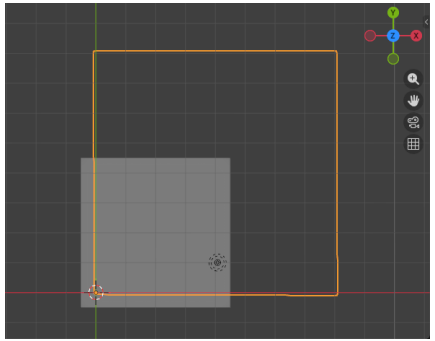
Unfortunately though, we cannot import a `VDB` object directly from memory, so we need to do a bit of file object rigmarole to make Blender aware of it.

VDB Gotchas

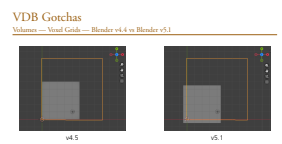
Volumes — Voxel Grids — Blender v4.4 vs Blender v5.1



v4.5



v5.1



Running this script shows us some of the things we need to watch out for:

1. First, the grid is not a $5 \times 5 \times 5$ grid, but rather a $8 \times 8 \times 8$ grid; which is due to the way `VDBs` represent volumes internally.
2. Second, the *placement* of the `VDBs` object have changed a little between various Blender releases: Back in 4.5, the volume started in the corner as one might expect, but nowadays, in both 5.1 and 5.2 it starts middle of the first voxel.

   Frankly, I am not sure if that is a bug or not, but I would argue that having the volume continue outside the bounds that Blender highlights is quite counter-intuitive.

   (It may also be useful to allow the user to set the origin to the center of the volume to make it easier to transform.)

VDB Gotchas

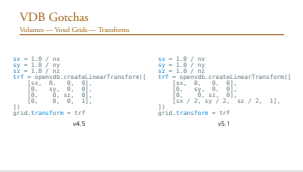
Volumes — Voxel Grids — Transforms

```
sx = 1.0 / nx
sy = 1.0 / ny
sz = 1.0 / nz
trf = openvdb.createLinearTransform([
    [sx,  0,   0,  0],
    [0,   sy,  0,  0],
    [0,    0, sz,  0],
    [0,    0,  0,  1],
])
grid.transform = trf
```

v4.5

```
sx = 1.0 / nx
sy = 1.0 / ny
sz = 1.0 / nz
trf = openvdb.createLinearTransform([
    [sx,  0,   0,  0],
    [0,   sy,  0,  0],
    [0,    0, sz,  0],
    [sx / 2, sy / 2,  sz / 2,  1],
])
grid.transform = trf
```

v5.1

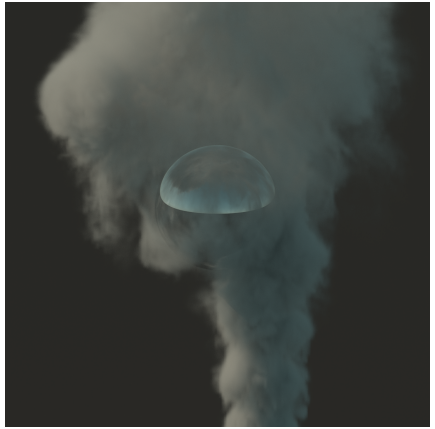


When you know about it though, it is relatively easy to correct: For 4.5, you don't really need to do anything, but I find it helpful to scale the volume down to the unit cube to make it easier to position it later, and for 5.1, you include a half voxel translation on all axes (last row).

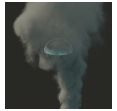
Volumes — Blender Labs

A Wish List?

- In Memory Volumes
- Packing VDBs
- NanoVDB / NeuralVDB



Volumes — Blender Labs
A Wish List?
- In Memory Volumes
- Packing VDBs
- NanoVDB / NeuralVDB

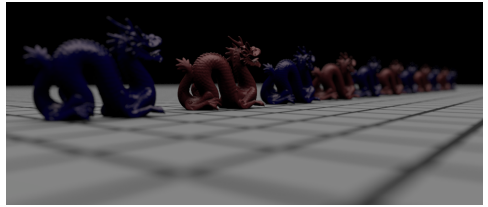


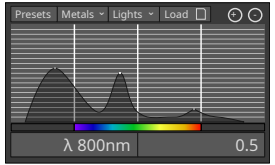
I do believe volumes are under active work in Blender labs however, so I guess things might change again here in the future, but while I am on that topic, there are a few other things that would be nice to have, such as:

- A bit better support for smaller volumes, as it feels a bit silly to create an external file for that.
- Or perhaps a way to pack the VDBs into the .blend file, or maybe even arbitrary files for that matter?
- And maybe extend the VDB support to other formats, such as `NanoVDB`, or even the more recent `NeuralVDB` format.

Limitations

- Many materials/objects left to implement!
- Large scene support
- "Real-world properties"
  - lights and materials
  - Spectral values





Limitations
• Many materials/objects left to implement!
• Large scene support
• "Real-world properties"
– lights and materials
– Spectral values

As we know though, most tools come with some kind of trade-offs, and this import tool is no exception: The most obvious limitation is that there are still quite a few materials and objects left to implement, and many of them need various parameters tweaks, such as the surface roughness. Beyond that…**(next slide)**

Large Scene Support

- Finding specific hidden objects?
- `material_utils` can help!



Large Scene Support
- Finding specific hidden objects?
- material_utils can help!



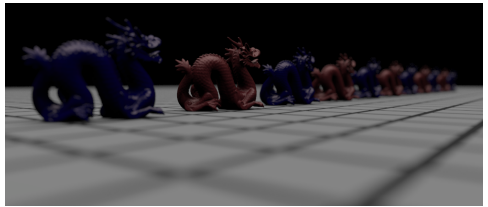
As I mentioned earlier, hiding objects or even generating them at render time with geometry nodes, is currently the best way to maintain interactivity in large scenes. It works, but is cumbersome, and may be annoying if you need to find a specific object among the many thousands of them, and to make matters worse, when the scene has this many objects, the search itself can get bogged down.

If you need to find the objects attached to specific materials, addons such as `material_utils` can alleviate it somewhat, but it runs into similar problems when the number of materials increases.

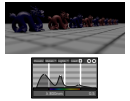
Truth be told, these scenes were intended for rendering and not to be extended with modelling tools, so it is not entirely unexpected to find performance issues. Still, it would be nice if it was faster, or even if the 3D viewport was *simply* decoupled from the from the rest of the interface to keep it responsive.

Real World Materials

- `Measured`
- `Fourier`
- Spectral Values



Real World Materials
• Measured
• Fourier
• Spectral Values



`PBRT` supports a few different materials that as far as I know have nothing that is even remotely similar in Blender: So called "measured" materials; basically, some real-world material have been meticulously measured in a lab from many angles and stored for later use.

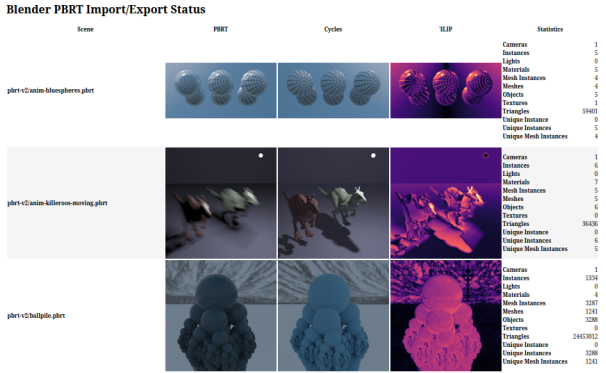
The "fourier" material is similar, but the data is effectively compressed.

While the uses for these are probably quite esoteric still, it may be of interest to make it easier to import materials such as these in the future.

Distribution

- Codeberg





https://gustafwaldemarson.com/publications/blender-pbrt-import/
https://codeberg.org/Xaldew/blender-pbrt-io.git



I was initially hopeful of being able to distribute this work in time for the conference, but due to the final bits of administration to finalize my doctoral research and some personal issues I've had to address this summer, it's been a bit delayed, but I hope to make it available on Codeberg at this location down here within a month or so.

In the mean-time, I have made a selection of renderings from `PBRT` and `Cycles` available on my personal home page, along with their corresponding ∃LIP error-map.

Conclusions

1. PBRT and Cycles
2. How to create scenes
   - Curves
   - Materials
   - Volumes
3. Scaling it to Large Scenes with Interactivity
4. Limitations



Conclusions
1. PBRT and Cycles
2. How to create scenes
   – Curves
   – Materials
   – Volumes
3. Scaling it to Large Scenes with Interactivity
4. Limitations

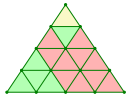
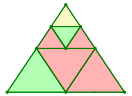
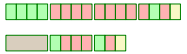


So what are the main takeaways from this presentation?

1. (Here, I've *briefly* presented `PBRT` the scene format and rendering framework with very different priorities when compared to `Cycles`.), and,
2. Shown a bit more of how it can be used to create scenes full of objects, with instances, *curves*, materials, and even *volumes*.
3. And, I also showed one technique for allowing us to work with these scenes when they start to reach the limits of what Blender lets us work with interactively.
4. And finally, I mentioned some of the limitations I found that I cannot currently work around, that I suppose we can consider as a wish-list for the future, although I will admit that it is a bit fat-fetched at times!

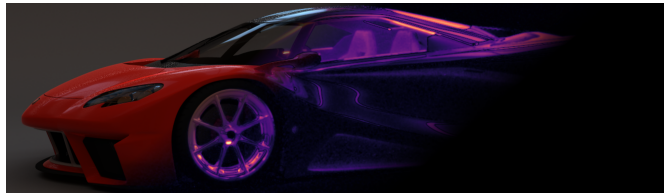
Thanks for Listening

Links / Contact

PBRT 2:
Creating and Working with Large Scenes

- https://gustafwaldemarson.com/pages/publications/
- gustaf.waldemarson@arm.com
- gustaf.waldemarson@gmail.com



BCON26

LUND UNIVERSITY

Thanks for Listening
Links / Contact
PBRT 2:
Creating and Working with Large Scenes
• https://gustafwaldemarson.com/pages/publications/
• gustaf.waldemarson@arm.com
• gustaf.waldemarson@gmail.com



And with that, I want to think you all for listening, and I suppose, open up for questions.

If you wish, you may also contact me through any of these emails. Thank you.

The End

2026-10-22

This page is intentionally left blank.

Extra Material

Extra Material

Triangle Meshes

PLY Problems

```
#!/bin/sh
find "${1-.}" -type f -name "*.ply" \
     -exec sed -i 's/^property float u$/property float s/g' {} \; \
     -exec sed -i 's/^property float v$/property float t/g' {} \;
```


Triangle Meshes
PLY Problems
#!/bin/sh
find "${1-.}" -type f -name "*.ply" \
-exec sed -i 's/^property float u$/property float s/g' {} \; \
-exec sed -i 's/^property float v$/property float t/g' {} \;

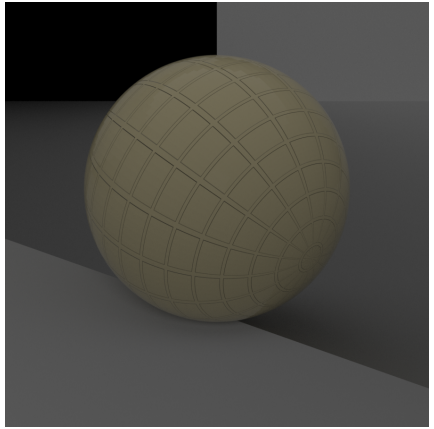
Of course, the most common object we care about is of course the triangle mesh. In PBRT, these are specified as either a list of indices, vertices, normals, etc. directly in the text file, or as a much more compact PLY file, which of course, we can use Blender internal routines to read for us.

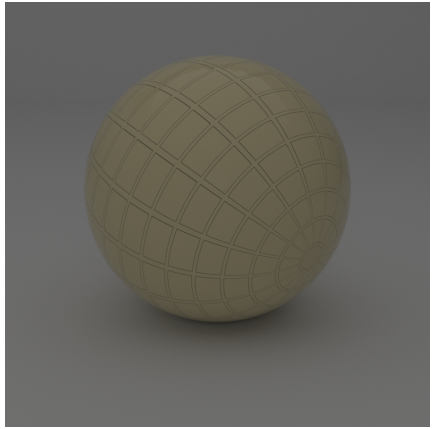
However, here I ran into a mildly annoying issue: Blender currently only recognize some of the common property keywords that are in common use. And unfortunately, PBRT is using a different one for the texture coordinates, namely “u, v” instead of “s, t”.

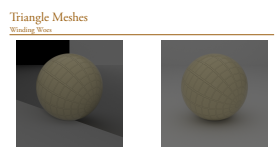
I have added a note to myself try to fix this in Blender at some point, but that probably wouldn’t propagate back to some older releases. Instead, I created a small script to modify PLY files to change the keyword to one that Blender recognizes.

Triangle Meshes

Winding Woes







A related problem is that some PBRT typically treat all surfaces as two-sided, whereas Blender usually does not. This occasionally leads to some trouble if the triangle winding is inconsistent, leading to flipped normals and other similar issues, as you can see in these images.

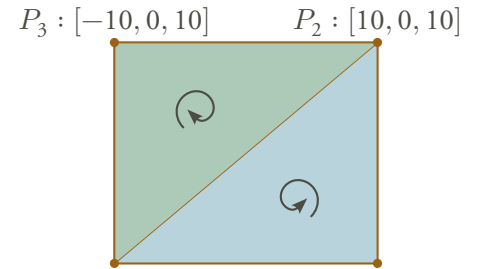
Triangle Meshes

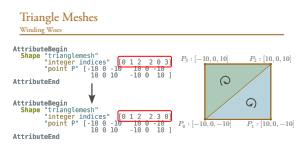
Winding Woes

```
AttributeBegin
  Shape "trianglemesh"
        "integer indices"  [0 1 2  2 0 3]
        "point P" [-10 0 -10   10 0 -10
                   10 0 10    -10 0  10 ]
AttributeEnd
```



```
AttributeBegin
  Shape "trianglemesh"
        "integer indices"  [0 1 2  2 3 0]
        "point P" [-10 0 -10   10 0 -10
                   10 0 10    -10 0  10 ]
AttributeEnd
```





Thankfully, this is relatively rare and mostly only show up in hand-made scenes such as the previous example, so it is relatively easy to fix it manually, since it is not easily fixed automatically.

References I

[1] Pontus Andersson, Jim Nilsson, and Tomas Akenine-Möller. "Visualizing and Communicating Errors in Rendered Images". In: *Ray Tracing Gems II*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. 2021. Chap. 19, pp. 301–320.

[2] Matt Pharr, Greg Humphreys, and Wenzel Jakob. *Physically Based Rendering: From Theory to Implementation*. 1st ed. 2004; 2nd ed. 2010; 3rd ed. 2016; 4th ed. 2023. Online 3rd and 4th ed. at `https://pbr-book.org/3ed-2018` and `https://pbr-book.org/4ed`. Morgan Kaufmann and MIT Press, 2023. URL: `https://pbr-book.org`.

[3] Gustaf Waldemarson. *PBRT 2: Creating and Working with Large Scenes*. The Blender Foundation, Youtube. 2026. URL: `https://youtu.be/iao8H8jPrqA`.

[4] Gustaf Waldemarson. *PBRT: Create your own Importers and Exporters*. The Blender Foundation, Youtube. 2024. URL: `https://youtu.be/BEbscsBRIx0`.

References I

[1] Pontus Andersson, Jim Nilsson, and Tomas Akenine-Möller. "Visualizing and Communicating Errors in Rendered Images". In: Ray Tracing Gems II. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. 2021. Chap. 19, pp. 301–320.

[2] Matt Pharr, Greg Humphreys, and Wenzel Jakob. Physically Based Rendering: From Theory to Implementation. 1st ed. 2004; 2nd ed. 2010; 3rd ed. 2016; 4th ed. 2023. Online 3rd and 4th ed. at https://pbr-book.org/3ed-2018 and https://pbr-book.org/4ed. Morgan Kaufmann and MIT Press, 2023. URL: https://pbr-book.org.

[3] Gustaf Waldemarson. PBRT 2: Creating and Working with Large Scenes. The Blender Foundation, Youtube. 2026. URL: https://youtu.be/iao8H8jPrqA.

[4] Gustaf Waldemarson. PBRT: Create your own Importers and Exporters. The Blender Foundation, Youtube. 2024. URL: https://youtu.be/BEbscsBRIx0.