pythreejs Documentation

Release 2.4.0

PyThreejs Development Team

Aug 24, 2022

INSTALLATION AND USAGE

Version: 2.4.0

pythreejs is a Jupyter widgets based notebook extension that allows Jupyter to leverage the WebGL capabilities of modern browsers by creating bindings to the javascript library three.js.

By being based on top of the jupyter-widgets infrastructure, it allows for eased integration with other interactive tools for notebooks.

CHAPTER

ONE

QUICKSTART

To get started with pythreejs, install with pip:

```
pip install pythreejs
```

If you are using a notebook version older than 5.3, or if your kernel is in another environment than the notebook server, you will also need to register the front-end extensions.

For the notebook front-end:

```
jupyter nbextension install [--sys-prefix | --user | --system] --py pythreejs
jupyter nbextension enable [--sys-prefix | --user | --system] --py pythreejs
```

For jupyterlab:

```
jupyter labextension install jupyter-threejs
```

Note: If you are installing an older version of pythreejs, you might have to add a version specifier for the labextension to match the Python package, e.g. *jupyter-threejs@1.0.0*.

CHAPTER

TWO

CONTENTS

2.1 Installation

The simplest way to install pythreejs is via pip:

```
pip install pythreejs
```

or via conda:

```
conda install pythreejs
```

With jupyter notebook version >= 5.3, this should also install and enable the relevant front-end extensions. If for some reason this did not happen (e.g. if the notebook server is in a different environment than the kernel), you can install / configure the front-end extensions manually. If you are using classic notebook (as opposed to Jupyterlab), run:

```
jupyter nbextension install [--sys-prefix / --user / --system] --py pythreejs

jupyter nbextension enable [--sys-prefix / --user / --system] --py pythreejs
```

with the appropriate flag. If you are using Jupyterlab, install the extension with:

```
jupyter labextension install jupyter-threejs
```

2.2 Upgrading to 1.x

If you are upgrading to version 1.x from a verion prior to 1.0, there are certain backwards-incompatible changes that you should note:

- `Plain[Buffer]Geometry` was renamed to `[Buffer]Geometry`. This was done in order to be more consistent with the names used in threejs. The base classes for geometry are now called `Base[Buffer]Geometry`. This also avoids the confusion with `Plane[Buffer]Geometry`.
- `LambertMaterial -> MeshLambertMaterial`, and other similar material class renames were done. Again, this was to more closely match the names used in three.js itself.

2.3 Introduction

The pythreejs API attempts to mimic the three.js API as closely as possible, so any resource on its API should also be helpful for understanding pythreejs. See for example the official three.js documentation.

The major difference between the two is the render loop. As we normally do not want to call back to the kernel for every rendered frame, some helper classes have been created to allow for user interaction with the scene with minimal overhead:

2.3.1 Renderer classes

While the `WebGLRenderer` class mimics its three.js counterpart in only rendering frames on demand (one frame per call to its `render()` method), the `Renderer` class sets up an interactive render loop allowing for *Interactive controls* and *Animation views*. Similarly, a `Preview` widget allows for a quick visualization of various threejs objects.

2.3.2 Interactive controls

These are classes for managing user interaction with the WebGL canvas, and translating that into actions. One example is the `OrbitControls` class, which allows the user to control the camera by zooming, panning, and orbital rotation around a target. Another example is the `Picker` widget, which allows for getting the objects and surface coordinates underneath the mouse cursor.

To use controls, pass them to the renderer, e.g.:

```
Renderer(controls=[OrbitControls(...), ...], ...)
```

2.3.3 Animation views

The view widgets for the `AnimationAction` class gives interactive controls to the user for controlling a threejs animation.

Other notable deviations from the threejs API are listed below:

- Buffers are based on numpy arrays, with their inbuilt knowledge of shape and dtype. As such, most threejs APIs that take a buffer are slightly modified (fewer options need to be specified explicitly).
- The generative geometry objects (e.g. `SphereGeometry` and `BoxBufferGeometry`) do not sync their vertices or similar data by default. To gain acess to the generated data, convert them to either the `Geometry` or `BufferGeometry` type with the `from_geometry()` factory method.
- Methods are often not mirrored to the Python side. However, they can be executed with the `exec_three_obj_method()` method. Consider contributing to make methods directly available. Possibly, these can be auto-generated as well.

2.4 Examples

This section contains several examples generated from Jupyter notebooks. The widgets have been embedded into the page.

2.4.1 Geometry types

[1]:
```
from pythreejs import *
from IPython.display import display
from math import pi
```

[2]:
```
# Reduce repo churn for examples with embedded state:
from pythreejs._example_helper import use_example_model_ids
use_example_model_ids()
```

[3]:
```
BoxGeometry(
    width=5,
    height=10,
    depth=15,
    widthSegments=5,
    heightSegments=10,
    depthSegments=15)
```

[3]:
```
BoxGeometry(depth=15.0, depthSegments=15, height=10.0, heightSegments=10, width=5.0,␣
↪widthSegments=5)
```

[4]:
```
BoxBufferGeometry(
    width=5,
    height=10,
    depth=15,
    widthSegments=5,
    heightSegments=10,
    depthSegments=15)
```

[4]:
```
BoxBufferGeometry(depth=15.0, depthSegments=15, height=10.0, heightSegments=10, width=5.
↪0, widthSegments=5)
```

[5]:
```
CircleGeometry(
    radius=10,
    segments=10,
    thetaStart=0.25,
    thetaLength=5.0)
```

[5]:
```
CircleGeometry(radius=10.0, segments=10, thetaLength=5.0, thetaStart=0.25)
```

[6]:
```
CircleBufferGeometry(
    radius=10,
    segments=10,
    thetaStart=0.25,
    thetaLength=5.0)
```

[6]:
```
CircleBufferGeometry(radius=10.0, segments=10, thetaLength=5.0, thetaStart=0.25)
```

[7]:
```
CylinderGeometry(
    radiusTop=5,
    radiusBottom=10,
    height=15,
    radialSegments=6,
    heightSegments=10,
    openEnded=False,
    thetaStart=0,
    thetaLength=2.0*pi)
```

[7]:
```
CylinderGeometry(height=15.0, heightSegments=10, radialSegments=6, radiusBottom=10.0,␣
↪radiusTop=5.0)
```

[8]:
```
CylinderBufferGeometry(
    radiusTop=5,
    radiusBottom=10,
    height=15,
    radialSegments=6,
    heightSegments=10,
    openEnded=False,
    thetaStart=0,
    thetaLength=2.0*pi)
```

[8]:
```
CylinderBufferGeometry(height=15.0, heightSegments=10, radialSegments=6, radiusBottom=10.
↪0, radiusTop=5.0)
```

[9]:
```
dodeca_geometry = DodecahedronGeometry(radius=10, detail=0, _flat=True)
dodeca_geometry
```

[9]:
```
DodecahedronGeometry(radius=10.0)
```

[10]:
```
LineSegments(
    EdgesGeometry(dodeca_geometry),
    LineBasicMaterial(parameters=dict(color='ffffff'))
)
```

[10]:
```
LineSegments(geometry=EdgesGeometry(geometry=DodecahedronGeometry(radius=10.0)),␣
↪material=LineBasicMaterial(),...
```

[ ]:
```
# TODO:
# ExtrudeGeometry(...)
```

[11]:
```
IcosahedronGeometry(radius=10, _flat=True)
```

[11]:
```
IcosahedronGeometry(radius=10.0)
```

[12]:
```
LatheBufferGeometry(
    points=[
        [ 0, -10, 0 ],
        [ 10, -5, 0 ],
        [ 5, 5, 0 ],
```

(continues on next page)

(continued from previous page)

```
        [ 0, 10, 0 ]
    ],
    segments=16,
    phiStart=0.0,
    phiLength=2.0*pi, _flat=True)
```

[12]:
```
LatheBufferGeometry(points=[[0, -10, 0], [10, -5, 0], [5, 5, 0], [0, 10, 0]],␣
↪segments=16)
```

[13]:
```
OctahedronGeometry(radius=10, detail=0, _flat=True)
```

[13]:
```
OctahedronGeometry(radius=10.0)
```

[14]:
```
ParametricGeometry(
    func="""function(u,v,out) {
        var x = 5 * (0.5 - u);
        var y = 5 * (0.5 - v);
        out.set(10 * x, 10 * y, x*x - y*y);
    }""",
    slices=5,
    stacks=10, _flat=True)
```

[14]:
```
ParametricGeometry(func='function(u,v,out) { \n        var x = 5 * (0.5 - u);\n       ␣
↪var y = 5 * (0.5 - v);\...
```

[15]:
```
PlaneGeometry(
    width=10,
    height=15,
    widthSegments=5,
    heightSegments=10)
```

[15]:
```
PlaneGeometry(height=15.0, heightSegments=10, width=10.0, widthSegments=5)
```

[16]:
```
PlaneBufferGeometry(
    width=10,
    height=15,
    widthSegments=5,
    heightSegments=10)
```

[16]:
```
PlaneBufferGeometry(height=15.0, heightSegments=10, width=10.0, widthSegments=5)
```

[ ]:
```
# TODO
# PolyhedronGeometry(...)
```

[17]:
```
# TODO: issues when radius is 0...
RingGeometry(
    innerRadius=10,
    outerRadius=25,
    thetaSegments=8,
    phiSegments=12,
    thetaStart=0,
    thetaLength=6.283185307179586)
```

[17]:
```
RingGeometry(innerRadius=10.0, outerRadius=25.0, phiSegments=12)
```

[18]:
```
# TODO: issues when radius is 0...
RingBufferGeometry(
    innerRadius=10,
    outerRadius=25,
    thetaSegments=8,
    phiSegments=12,
    thetaStart=0,
    thetaLength=6.283185307179586)
```

[18]:
```
RingBufferGeometry(innerRadius=10.0, outerRadius=25.0, phiSegments=12)
```

[ ]:
```
# TODO
# ShapeGeometry(...)
```

[19]:
```
SphereGeometry(
    radius=20,
    widthSegments=8,
    heightSegments=6,
    phiStart=0,
    phiLength=1.5*pi,
    thetaStart=0,
    thetaLength=2.0*pi/3.0)
```

[19]:
```
SphereGeometry(phiLength=4.71238898038469, radius=20.0, thetaLength=2.0943951023931953)
```

[20]:
```
SphereBufferGeometry(
    radius=20,
    widthSegments=8,
    heightSegments=6,
    phiStart=0,
    phiLength=1.5*pi,
    thetaStart=0,
    thetaLength=2.0*pi/3.0)
```

[20]:
```
SphereBufferGeometry(phiLength=4.71238898038469, radius=20.0, thetaLength=2.
↪0943951023931953)
```

[21]:
```
TetrahedronGeometry(radius=10, detail=1, _flat=True)
```

[21]:
```
TetrahedronGeometry(detail=1, radius=10.0)
```

[ ]:
```
# TODO: font loading
# TextGeometry(...)
```

[22]:
```
TorusGeometry(
    radius=20,
    tube=5,
    radialSegments=20,
    tubularSegments=6,
    arc=1.5*pi)
```

[22]:
```
TorusGeometry(arc=4.71238898038469, radialSegments=20, radius=20.0, tube=5.0)
```

[23]:
```
TorusBufferGeometry(radius=100)
```

[23]:
```
TorusBufferGeometry(radius=100.0)
```

[24]:
```
TorusKnotGeometry(
    radius=20,
    tube=5,
    tubularSegments=64,
    radialSegments=8,
    p=2,
    q=3)
```

[24]:
```
TorusKnotGeometry(radius=20.0, tube=5.0)
```

[25]:
```
TorusKnotBufferGeometry(
    radius=20,
    tube=5,
    tubularSegments=64,
    radialSegments=8,
    p=2,
    q=3)
```

[25]:
```
TorusKnotBufferGeometry(radius=20.0, tube=5.0)
```

[ ]:
```
# TODO: handling THREE.Curve
TubeGeometry(
    path=None,
    segments=64,
    radius=1,
    radialSegments=8,
    close=False)
```

[26]:
```
WireframeGeometry(geometry=TorusBufferGeometry(
    radius=20,
    tube=5,
    radialSegments=6,
    tubularSegments=20,
    arc=2.0*pi
))
```

[26]:
```
WireframeGeometry(geometry=TorusBufferGeometry(radialSegments=6, radius=20.0, tube=5.0,␣
↪tubularSegments=20))
```

[ ]:
```

```

2.4.2 Animation

[1]:
```
from pythreejs import *
import ipywidgets
from IPython.display import display
```

[2]:
```
# Reduce repo churn for examples with embedded state:
from pythreejs._example_helper import use_example_model_ids
use_example_model_ids()
```

[3]:
```
view_width = 600
view_height = 400
```

Let's first set up a basic scene with a cube and a sphere,

[4]:
```
sphere = Mesh(
    SphereBufferGeometry(1, 32, 16),
    MeshStandardMaterial(color='red')
)
```

[5]:
```
cube = Mesh(
    BoxBufferGeometry(1, 1, 1),
    MeshPhysicalMaterial(color='green'),
    position=[2, 0, 4]
)
```

as well as lighting and camera:

[6]:
```
camera = PerspectiveCamera( position=[10, 6, 10], aspect=view_width/view_height)
key_light = DirectionalLight(position=[0, 10, 10])
ambient_light = AmbientLight()
```

Keyframe animation

The three.js animation system is built as a keyframe system. We'll demonstrate this by animating the position and rotation of our camera.

First, we set up the keyframes for the position and the rotation separately:

[7]:
```
positon_track = VectorKeyframeTrack(name='.position',
    times=[0, 2, 5],
    values=[10, 6, 10,
            6.3, 3.78, 6.3,
            -2.98, 0.84, 9.2,
           ])
rotation_track = QuaternionKeyframeTrack(name='.quaternion',
    times=[0, 2, 5],
    values=[-0.184, 0.375, 0.0762, 0.905,
            -0.184, 0.375, 0.0762, 0.905,
            -0.0430, -0.156, -0.00681, 0.987,
           ])
```

Next, we create an animation clip combining the two tracks, and finally an animation action to control the animation. See the three.js docs for more details on the different responsibilities of the different classes.

[8]:
```
camera_clip = AnimationClip(tracks=[positon_track, rotation_track])
camera_action = AnimationAction(AnimationMixer(camera), camera_clip, camera)
```

Now, let's see it in action:

[9]:
```
scene = Scene(children=[sphere, cube, camera, key_light, ambient_light])
controller = OrbitControls(controlling=camera)
renderer = Renderer(camera=camera, scene=scene, controls=[controller],
                    width=view_width, height=view_height)
```

[10]:
```
renderer
```

[10]:
```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(10.0, 6.0, 10.0),␣
↪projectionMatrix=(1.4296712803397058...
```

[11]:
```
camera_action
```

[11]:
```
AnimationAction(clip=AnimationClip(duration=5.0, tracks=(VectorKeyframeTrack(name='.
↪position', times=array([0,...
```

Let's add another animation clip, this time animating the color of the sphere's material:

[12]:
```
color_track = ColorKeyframeTrack(name='.material.color',
    times=[0, 1], values=[1, 0, 0,  0, 0, 1])  # red to blue

color_clip = AnimationClip(tracks=[color_track], duration=1.5)
color_action = AnimationAction(AnimationMixer(sphere), color_clip, sphere)
```

[13]:
```
color_action
```

[13]:
```
AnimationAction(clip=AnimationClip(duration=1.5, tracks=(ColorKeyframeTrack(name='.
↪material.color', times=arra...
```

Note how the two animation clips can freely be combined since they affect different properties. It's also worth noting that the color animation can be combined with manual camera control, while the camera animation cannot. When animating the camera, you might want to consider disabling the manual controls.

Animating rotation

When animating the camera rotation above, we used the camera's `quaternion`. This is the most robust method for animating free-form rotations. For example, the animation above was created by first moving the camera manually, and then reading out its `position` and `quaternion` properties at the wanted views. If you want more intuitive axes control, it is possible to animate the `rotation` sub-attributes instead, as shown below.

[14]:
```
f = """
function f(origu, origv, out) {
    // scale u and v to the ranges I want: [0, 2*pi]
    var u = 2*Math.PI*origu;
    var v = 2*Math.PI*origv;

    var x = Math.sin(u);
```

(continues on next page)

(continued from previous page)

```
    var y = Math.cos(v);
    var z = Math.cos(u+v);

    out.set(x,y,z)
}
"""
surf_g = ParametricGeometry(func=f, slices=16, stacks=16);

surf1 = Mesh(geometry=surf_g,
             material=MeshLambertMaterial(color='green', side='FrontSide'))
surf2 = Mesh(geometry=surf_g,
             material=MeshLambertMaterial(color='yellow', side='BackSide'))
surf = Group(children=[surf1, surf2])

camera2 = PerspectiveCamera( position=[10, 6, 10], aspect=view_width/view_height)
scene2 = Scene(children=[surf, camera2,
                         DirectionalLight(position=[3, 5, 1], intensity=0.6),
                         AmbientLight(intensity=0.5)])
renderer2 = Renderer(camera=camera2, scene=scene2,
                     controls=[OrbitControls(controlling=camera2)],
                     width=view_width, height=view_height)
display(renderer2)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(10.0, 6.0, 10.0),␣
↪projectionMatrix=(1.0, 0.0, 0.0, 0.0...
```

[15]:
```
spin_track = NumberKeyframeTrack(name='.rotation[y]', times=[0, 2], values=[0, 6.28])
spin_clip = AnimationClip(tracks=[spin_track])
spin_action = AnimationAction(AnimationMixer(surf), spin_clip, surf)
spin_action
```

[15]:
```
AnimationAction(clip=AnimationClip(tracks=(NumberKeyframeTrack(name='.rotation[y]',␣
↪times=array([0, 2]), value...
```

Note that we are spinning the object itself, and that we are therefore free to manipulate the camera at will.

Morph targets

Set up a simple sphere geometry, and add a morph target that is an oblong pill shape:

[16]:
```
# This lets three.js create the geometry, then syncs back vertex positions etc.
# For this reason, you should allow for the sync to complete before executing
# the next cell.
morph = BufferGeometry.from_geometry(SphereBufferGeometry(1, 32, 16))
```

[17]:
```
import numpy as np

# Set up morph targets:
vertices = np.array(morph.attributes['position'].array)
for i in range(len(vertices)):
    if vertices[i, 0] > 0:
        vertices[i, 0] += 1
```

(continues on next page)

(continued from previous page)

```
morph.morphAttributes = {'position': [
    BufferAttribute(vertices),
]}

morphMesh = Mesh(morph, MeshPhongMaterial(
    color='#ff3333', shininess=150, morphTargets=True))
```

Set up animation for going back and forth between the sphere and pill shape:

[18]:
```
pill_track = NumberKeyframeTrack(
    name='.morphTargetInfluences[0]', times=[0, 1.5, 3], values=[0, 2.5, 0])
pill_clip = AnimationClip(tracks=[pill_track])
pill_action = AnimationAction(AnimationMixer(morphMesh), pill_clip, morphMesh)
```

[19]:
```
camera3 = PerspectiveCamera( position=[5, 3, 5], aspect=view_width/view_height)
scene3 = Scene(children=[morphMesh, camera3,
                         DirectionalLight(position=[3, 5, 1], intensity=0.6),
                         AmbientLight(intensity=0.5)])
renderer3 = Renderer(camera=camera3, scene=scene3,
                     controls=[OrbitControls(controlling=camera3)],
                     width=view_width, height=view_height)
display(renderer3, pill_action)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(5.0, 3.0, 5.0),␣
↪projectionMatrix=(1.0, 0.0, 0.0, 0.0, ...
```

```
AnimationAction(clip=AnimationClip(duration=3.0, tracks=(NumberKeyframeTrack(name='.
↪morphTargetInfluences[0]',...
```

Skeletal animation

First, set up a skinned mesh with some bones:

[20]:
```
import numpy as np

N_BONES = 3

ref_cylinder = CylinderBufferGeometry(5, 5, 50, 5, N_BONES * 5, True)
cylinder = BufferGeometry.from_geometry(ref_cylinder)
```

[21]:
```
skinIndices = []
skinWeights = []
vertices = cylinder.attributes['position'].array
boneHeight = ref_cylinder.height / (N_BONES - 1)
for i in range(vertices.shape[0]):

    y = vertices[i, 1] + 0.5 * ref_cylinder.height

    skinIndex = y // boneHeight
    skinWeight = ( y % boneHeight ) / boneHeight

    # Ease between each bone
```

(continues on next page)

(continued from previous page)

```
    skinIndices.append([skinIndex, skinIndex + 1, 0, 0 ])
    skinWeights.append([1 - skinWeight, skinWeight, 0, 0 ])

cylinder.attributes = dict(
    cylinder.attributes,
    skinIndex=BufferAttribute(skinIndices),
    skinWeight=BufferAttribute(skinWeights),
)

shoulder = Bone(position=(0, -25, 0))
elbow = Bone(position=(0, 25, 0))
hand = Bone(position=(0, 25, 0))

shoulder.add(elbow)
elbow.add(hand)
bones = [shoulder, elbow, hand]
skeleton = Skeleton(bones)

mesh = SkinnedMesh(cylinder, MeshPhongMaterial(side='DoubleSide', skinning=True))
mesh.add(bones[0])
mesh.skeleton = skeleton
```

[22]:
```
helper = SkeletonHelper(mesh)
```

Next, set up some simple rotation animations for the bones:

[23]:
```
# Rotate on x and z axes:
bend_tracks = [
    NumberKeyframeTrack(
        name='.bones[1].rotation[x]',
        times=[0, 0.5, 1.5, 2],
        values=[0, 0.3, -0.3, 0]),
    NumberKeyframeTrack(
        name='.bones[1].rotation[z]',
        times=[0, 0.5, 1.5, 2],
        values=[0, 0.3, -0.3, 0]),
    NumberKeyframeTrack(
        name='.bones[2].rotation[x]',
        times=[0, 0.5, 1.5, 2],
        values=[0, -0.3, 0.3, 0]),
    NumberKeyframeTrack(
        name='.bones[2].rotation[z]',
        times=[0, 0.5, 1.5, 2],
        values=[0, -0.3, 0.3, 0]),
]
bend_clip = AnimationClip(tracks=bend_tracks)
bend_action = AnimationAction(AnimationMixer(mesh), bend_clip, mesh)

# Rotate on y axis:
wring_tracks = [
    NumberKeyframeTrack(name='.bones[1].rotation[y]', times=[0, 0.5, 1.5, 2], values=[0,␣
↪0.7, -0.7, 0]),
```

(continues on next page)

(continued from previous page)

```
    NumberKeyframeTrack(name='.bones[2].rotation[y]', times=[0, 0.5, 1.5, 2], values=[0,␣
↪0.7, -0.7, 0]),
]

wring_clip = AnimationClip(tracks=wring_tracks)
wring_action = AnimationAction(AnimationMixer(mesh), wring_clip, mesh)
```

[24]:
```
camera4 = PerspectiveCamera( position=[40, 24, 40], aspect=view_width/view_height)
scene4 = Scene(children=[mesh, helper, camera4,
                         DirectionalLight(position=[3, 5, 1], intensity=0.6),
                         AmbientLight(intensity=0.5)])
renderer4 = Renderer(camera=camera4, scene=scene4,
                     controls=[OrbitControls(controlling=camera4)],
                     width=view_width, height=view_height)
display(renderer4)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(40.0, 24.0, 40.0),␣
↪projectionMatrix=(1.0, 0.0, 0.0, 0....
```

[25]:
```
bend_action
```

[25]:
```
AnimationAction(clip=AnimationClip(duration=2.0, tracks=(NumberKeyframeTrack(name='.
↪bones[1].rotation[x]', tim...
```

[26]:
```
wring_action
```

[26]:
```
AnimationAction(clip=AnimationClip(duration=2.0, tracks=(NumberKeyframeTrack(name='.
↪bones[1].rotation[y]', tim...
```

[ ]:
```

```

2.4.3 Textures

[1]:
```
from pythreejs import *
from IPython.display import display
from math import pi
```

[2]:
```
# Reduce repo churn for examples with embedded state:
from pythreejs._example_helper import use_example_model_ids
use_example_model_ids()
```

[3]:
```
checker_tex = ImageTexture(imageUri='img/checkerboard.png')
earth_tex = ImageTexture(imageUri='img/earth.jpg')
```

[4]:
```
checker_tex
```

[4]:
```
ImageTexture(imageUri='img/checkerboard.png', offset=(0.0, 0.0), repeat=(1.0, 1.0),␣
↪version=1)
```

[5]:
```
earth_tex
```

[5]:
```
ImageTexture(imageUri='img/earth.jpg', offset=(0.0, 0.0), repeat=(1.0, 1.0), version=1)
```

[6]:
```
#
# Create checkerboard pattern
#

# tex dims need to be power of two.
arr_w = 16
arr_h = 8

import numpy as np

def gen_checkers(width, height, n_checkers_x, n_checkers_y):
    array = np.ones((width, height, 3), dtype='float32')

    # width in texels of each checker
    checker_w = width / n_checkers_x
    checker_h = height / n_checkers_y


    for y in range(height):
        for x in range(width):
            color_key = int(x / checker_w) + int(y / checker_h)
            if color_key % 2 == 0:
                array[x, y, :] = [ 0, 0, 0 ]
            else:
                array[x, y, :] = [ 1, 1, 1 ]
    # We need to flip x/y since threejs/webgl insists on column-major data for␣
↪DataTexture
    return np.swapaxes(array, 0, 1)


data_tex = DataTexture(
    data=gen_checkers(arr_w, arr_h, 4, 2),
    format="RGBFormat",
    type="FloatType",
)
```

[7]:
```
data_tex
```

[7]:
```
DataTexture(data=array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
     ...
```

[8]:
```
data_tex.data = gen_checkers(arr_w, arr_h, 8, 2)
```

[ ]:
```

```

2.4.4 Renderer properties

[1]:
```
from pythreejs import *
from IPython.display import display
import ipywidgets
```

[2]:
```
# Reduce repo churn for examples with embedded state:
from pythreejs._example_helper import use_example_model_ids
use_example_model_ids()
```

Transparent background

To have the render view use a transparent background, there are three steps you need to do: 1. Ensure that the `background` property of the `Scene` object is set to `None`. 2. Ensure that `alpha=True` is passed to the constructor of the `Renderer` object. This ensures that an alpha channel is used by the renderer. 3. Ensure that the `clearOpacity` property of the `Renderer` object is set to 0. For more details about this, see below.

[3]:
```
ball = Mesh(geometry=SphereGeometry(),
            material=MeshLambertMaterial(color='red'))
key_light = DirectionalLight(color='white', position=[3, 5, 1], intensity=0.5)

c = PerspectiveCamera(position=[0, 5, 5], up=[0, 1, 0], children=[key_light])

scene = Scene(children=[ball, c, AmbientLight(color='#777777')], background=None)

renderer = Renderer(camera=c,
                    scene=scene,
                    alpha=True,
                    clearOpacity=0,
                    controls=[OrbitControls(controlling=c)])
display(renderer)
```

```
Renderer(camera=PerspectiveCamera(children=(DirectionalLight(color='white', intensity=0.
↪5, position=(3.0, 5.0,...
```

The use of clear color/opacity is explained in more detailed in the docs of three.js, but in short: - If `autoClear` is true the renderer output is cleared on each rendered frame. - If `autoClearColor` is true the background color is cleared on each frame. - When the background color is cleared, it is reset to `Renderer.clearColor`, with an opacity of `Renderer.clearOpacity`.

[4]:
```
# Let's set up some controls for the clear color/opacity:

opacity = ipywidgets.FloatSlider(min=0., max=1.)
ipywidgets.jslink((opacity, 'value'), (renderer, 'clearOpacity'))

color = ipywidgets.ColorPicker()
ipywidgets.jslink((color, 'value'), (renderer, 'clearColor'))

display(ipywidgets.HBox(children=[
    ipywidgets.Label('Clear color:'), color, ipywidgets.Label('Clear opactiy:'),␣
↪opacity]))
```

```
HBox(children=(Label(value='Clear color:'), ColorPicker(value='black'), Label(value=
↪'Clear opactiy:'), FloatSl...
```

Scene background

If we set the background property of the scene, it will be filled in on top of whatever clear color is there, basically making the clear color ineffective.

[5]:
```
scene_background = ipywidgets.ColorPicker()
_background_link = None

def toggle_scene_background(change):
    global _background_link
    if change['new']:
        _background_link = ipywidgets.jslink((scene_background, 'value'), (scene,
↪'background'))
    else:
        _background_link.close()
        _background_link = None
        scene.background = None

scene_background_toggle = ipywidgets.ToggleButton(False, description='Scene Color')
scene_background_toggle.observe(toggle_scene_background, 'value')

display(ipywidgets.HBox(children=[
    ipywidgets.Label('Scene background color:'), scene_background, scene_background_
↪toggle]))
```

```
HBox(children=(Label(value='Scene background color:'), ColorPicker(value='black'),␣
↪ToggleButton(value=False, d...
```

[ ]:
```

```

2.4.5 Thick line geometry

Three.js has some example code for thick lines via an instance-based geometry. Since WebGL does not guarantee support for line thickness greater than 1 for GL lines, pytheejs includes these objects by default.

[1]:
```
from pythreejs import *
from IPython.display import display
from ipywidgets import VBox, HBox, Checkbox, jslink
import numpy as np
```

[2]:
```
# Reduce repo churn for examples with embedded state:
from pythreejs._example_helper import use_example_model_ids
use_example_model_ids()
```

First, let's set up a normal GL line for comparison. Depending on your OS/browser combination, this might not respect the `linewidth` argument. E.g. most browsers on Windows does not support linewidth greater than 1, due to lack of support in the ANGLE library that most browsers rely on.

[3]:
```
g1 = BufferGeometry(
    attributes={
        'position': BufferAttribute(np.array([
            [0, 0, 0], [1, 1, 1],
            [2, 2, 2], [4, 4, 4]
        ], dtype=np.float32), normalized=False),
        'color': BufferAttribute(np.array([
            [1, 0, 0], [1, 0, 0],
            [0, 1, 0], [0, 0, 1]
        ], dtype=np.float32), normalized=False),
    },
)
m1 = LineBasicMaterial(vertexColors='VertexColors', linewidth=10)
line1 = LineSegments(g1, m1)
line1
```

[3]:
```
LineSegments(geometry=BufferGeometry(attributes={'position':␣
↪BufferAttribute(array=array([[0., 0., 0.],
      ...
```

Next, we'll set up two variants of the instance geometry based lines. One with a single color, and one with vertex colors.

[4]:
```
g2 = LineSegmentsGeometry(
    positions=[
        [[0, 0, 0], [1, 1, 1]],
        [[2, 2, 2], [4, 4, 4]]
    ],
)
m2 = LineMaterial(linewidth=10, color='cyan')
line2 = LineSegments2(g2, m2)
line2
```

[4]:
```
LineSegments2(geometry=LineSegmentsGeometry(positions=array([[[0., 0., 0.],
        [1., 1., 1.]],

       [[2...
```

[5]:
```
g3 = LineSegmentsGeometry(
    positions=[
        [[0, 0, 0], [1, 1, 1]],
        [[2, 2, 2], [4, 4, 4]]
    ],
    colors=[
        [[1, 0, 0], [1, 0, 0]],
        [[0, 1, 0], [0, 0, 1]]
    ],
)
m3 = LineMaterial(linewidth=10, vertexColors='VertexColors')
line3 = LineSegments2(g3, m3)
line3
```

[5]:
```
LineSegments2(geometry=LineSegmentsGeometry(colors=array([[[1., 0., 0.],
        [1., 0., 0.]],
```

(continues on next page)

(continued from previous page)

```
       [[0., ...
```

Finally, let's set up a simple scene and renderer, and add some checkboxes so we can toggle the visibility of the different lines.

[6]:
```
view_width = 600
view_height = 400
camera = PerspectiveCamera(position=[10, 0, 0], aspect=view_width/view_height)
key_light = DirectionalLight(position=[0, 10, 10])
ambient_light = AmbientLight()
```

[7]:
```
scene = Scene(children=[line1, line2, line3, camera, key_light, ambient_light])
controller = OrbitControls(controlling=camera, screenSpacePanning=False)
renderer = Renderer(camera=camera, scene=scene, controls=[controller],
                    width=view_width, height=view_height)
```

[8]:
```
chks = [
    Checkbox(True, description='GL line'),
    Checkbox(True, description='Fat line (single color)'),
    Checkbox(True, description='Fat line (vertex colors)'),
]
jslink((chks[0], 'value'), (line1, 'visible'))
jslink((chks[1], 'value'), (line2, 'visible'))
jslink((chks[2], 'value'), (line3, 'visible'))
VBox([renderer, HBox(chks)])
```

[8]:
```
VBox(children=(Renderer(camera=PerspectiveCamera(aspect=1.5, position=(10.0, 0.0, 0.0),␣
↪projectionMatrix=(1.42...
```

For reference, the code below shows how you would recreate the line geometry and material from the kernel. The only significant difference is that you need to declare the render view resolution on material creation, while the included `LineMaterial` automatically sets this.

[9]:
```
# The line segment points and colors.
# Each array of six is one instance/segment [x1, y1, z1, x2, y2, z2]
posInstBuffer = InstancedInterleavedBuffer( np.array([
    [0, 0, 0, 1, 1, 1],
    [2, 2, 2, 4, 4, 4]
], dtype=np.float32))
colInstBuffer = InstancedInterleavedBuffer( np.array([
    [1, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 1]
], dtype=np.float32))

# This uses InstancedBufferGeometry, so that the geometry is reused for each line segment
lineGeo = InstancedBufferGeometry(attributes={
    # Helper line geometry (2x4 grid), that is instanced
    'position': BufferAttribute(np.array([
        [ 1,  2, 0], [1,  2, 0],
        [-1,  1, 0], [1,  1, 0],
        [-1,  0, 0], [1,  0, 0],
        [-1, -1, 0], [1, -1, 0]
```

(continues on next page)

(continued from previous page)

```
    ], dtype=np.float32)),
    'uv': BufferAttribute(np.array([
        [-1,  2], [1,  2],
        [-1,  1], [1,  1],
        [-1, -1], [1, -1],
        [-1, -2], [1, -2]
    ], dtype=np.float32)),
    'index': BufferAttribute(np.array([
        0, 2, 1,
        2, 3, 1,
        2, 4, 3,
        4, 5, 3,
        4, 6, 5,
        6, 7, 5
    ], dtype=np.uint8)),
    # The line segments are split into start/end for each instance:
    'instanceStart': InterleavedBufferAttribute(posInstBuffer, 3, 0),
    'instanceEnd': InterleavedBufferAttribute(posInstBuffer, 3, 3),
    'instanceColorStart': InterleavedBufferAttribute(colInstBuffer, 3, 0),
    'instanceColorEnd': InterleavedBufferAttribute(colInstBuffer, 3, 3),
})
```

[10]:
```
# The line material shader:
lineMat = ShaderMaterial(
    vertexShader='''
#include <common>
#include <color_pars_vertex>
#include <fog_pars_vertex>
#include <logdepthbuf_pars_vertex>
#include <clipping_planes_pars_vertex>

uniform float linewidth;
uniform vec2 resolution;

attribute vec3 instanceStart;
attribute vec3 instanceEnd;

attribute vec3 instanceColorStart;
attribute vec3 instanceColorEnd;

varying vec2 vUv;

void trimSegment( const in vec4 start, inout vec4 end ) {

    // trim end segment so it terminates between the camera plane and the near plane

    // conservative estimate of the near plane
    float a = projectionMatrix[ 2 ][ 2 ]; // 3nd entry in 3th column
    float b = projectionMatrix[ 3 ][ 2 ]; // 3nd entry in 4th column
    float nearEstimate = - 0.5 * b / a;

    float alpha = ( nearEstimate - start.z ) / ( end.z - start.z );
```

(continues on next page)

(continued from previous page)

```
    end.xyz = mix( start.xyz, end.xyz, alpha );

}

void main() {

    #ifdef USE_COLOR

        vColor.xyz = ( position.y < 0.5 ) ? instanceColorStart : instanceColorEnd;

    #endif

    float aspect = resolution.x / resolution.y;

    vUv = uv;

    // camera space
    vec4 start = modelViewMatrix * vec4( instanceStart, 1.0 );
    vec4 end = modelViewMatrix * vec4( instanceEnd, 1.0 );

    // special case for perspective projection, and segments that terminate either in,␣
↪or behind, the camera plane
    // clearly the gpu firmware has a way of addressing this issue when projecting into␣
↪ndc space
    // but we need to perform ndc-space calculations in the shader, so we must address␣
↪this issue directly
    // perhaps there is a more elegant solution -- WestLangley

    bool perspective = ( projectionMatrix[ 2 ][ 3 ] == - 1.0 ); // 4th entry in the 3rd␣
↪column

    if ( perspective ) {

        if ( start.z < 0.0 && end.z >= 0.0 ) {

            trimSegment( start, end );

        } else if ( end.z < 0.0 && start.z >= 0.0 ) {

            trimSegment( end, start );

        }

    }

    // clip space
    vec4 clipStart = projectionMatrix * start;
    vec4 clipEnd = projectionMatrix * end;

    // ndc space
    vec2 ndcStart = clipStart.xy / clipStart.w;
```

(continues on next page)

(continued from previous page)

```
    vec2 ndcEnd = clipEnd.xy / clipEnd.w;

    // direction
    vec2 dir = ndcEnd - ndcStart;

    // account for clip-space aspect ratio
    dir.x *= aspect;
    dir = normalize( dir );

    // perpendicular to dir
    vec2 offset = vec2( dir.y, - dir.x );

    // undo aspect ratio adjustment
    dir.x /= aspect;
    offset.x /= aspect;

    // sign flip
    if ( position.x < 0.0 ) offset *= - 1.0;

    // endcaps
    if ( position.y < 0.0 ) {

        offset += - dir;

    } else if ( position.y > 1.0 ) {

        offset += dir;

    }

    // adjust for linewidth
    offset *= linewidth;

    // adjust for clip-space to screen-space conversion // maybe resolution should be␣
↪based on viewport ...
    offset /= resolution.y;

    // select end
    vec4 clip = ( position.y < 0.5 ) ? clipStart : clipEnd;

    // back to clip space
    offset *= clip.w;

    clip.xy += offset;

    gl_Position = clip;

    vec4 mvPosition = ( position.y < 0.5 ) ? start : end; // this is an approximation

    #include <logdepthbuf_vertex>
    #include <clipping_planes_vertex>
    #include <fog_vertex>
```

(continues on next page)

(continued from previous page)

```
}
''',
    fragmentShader='''
uniform vec3 diffuse;
uniform float opacity;

varying float vLineDistance;

#include <common>
#include <color_pars_fragment>
#include <fog_pars_fragment>
#include <logdepthbuf_pars_fragment>
#include <clipping_planes_pars_fragment>

varying vec2 vUv;

void main() {

    #include <clipping_planes_fragment>


    if ( abs( vUv.y ) > 1.0 ) {

        float a = vUv.x;
        float b = ( vUv.y > 0.0 ) ? vUv.y - 1.0 : vUv.y + 1.0;
        float len2 = a * a + b * b;

        if ( len2 > 1.0 ) discard;

    }

    vec4 diffuseColor = vec4( diffuse, opacity );

    #include <logdepthbuf_fragment>
    #include <color_fragment>

    gl_FragColor = vec4( diffuseColor.rgb, diffuseColor.a );

    #include <premultiplied_alpha_fragment>
    #include <tonemapping_fragment>
    #include <encodings_fragment>
    #include <fog_fragment>

}
''',
    vertexColors='VertexColors',
    uniforms=dict(
        linewidth={'value': 10.0},
        resolution={'value': (100., 100.)},
        **UniformsLib['common']
    )
)
```

```
[11]: Mesh(lineGeo, lineMat)
```

```
[11]: Mesh(geometry=InstancedBufferGeometry(attributes={'position':␣
      ↪BufferAttribute(array=array([[ 1.,  2.,  0.],
        ...
```

```
[ ]:
```

2.5 API Reference

The pythreejs API attempts to mimic the three.js API as closely as possible. This API reference therefore does not attempt to explain the purpose of any forwarded objects or attributes, but can still be useful for:

- The trait signatures of various properties.
- Classes, properties and methods custom to pythreejs.
- Variations from the three.js API, e.g. for `BufferAttribute`.

2.5.1 _base

Preview

RenderableWidget

ThreeWidget

2.5.2 animation

tracks

BooleanKeyframeTrack

ColorKeyframeTrack

NumberKeyframeTrack

QuaternionKeyframeTrack

StringKeyframeTrack

VectorKeyframeTrack

AnimationAction

AnimationClip

AnimationMixer

AnimationObjectGroup

AnimationUtils

KeyframeTrack

PropertyBinding

PropertyMixer

2.5.3 audio

AudioAnalyser

AudioListener

Audio

PositionalAudio

2.5.4 cameras

ArrayCamera

Camera

CombinedCamera

CubeCamera

OrthographicCamera

PerspectiveCamera

StereoCamera

2.5.5 controls

Controls

FlyControls

OrbitControls

Picker

TrackballControls

2.5.6 core

BaseBufferGeometry

BaseGeometry

BufferAttribute

BufferGeometry

Clock

DirectGeometry

EventDispatcher

Geometry

InstancedBufferAttribute

InstancedBufferGeometry

InstancedInterleavedBuffer

InterleavedBufferAttribute

InterleavedBuffer

Layers

Object3D

Raycaster

2.5.7 extras

core

CurvePath

Curve

Font

Path

ShapePath

Shape

curves

ArcCurve

CatmullRomCurve3

CubicBezierCurve3

CubicBezierCurve

EllipseCurve

LineCurve3

LineCurve

QuadraticBezierCurve3

QuadraticBezierCurve

SplineCurve

objects

ImmediateRenderObject

2.5.8 geometries

BoxBufferGeometry

BoxGeometry

BoxLineGeometry

CircleBufferGeometry

CircleGeometry

ConeGeometry

CylinderBufferGeometry

CylinderGeometry

DodecahedronGeometry

EdgesGeometry

ExtrudeGeometry

IcosahedronGeometry

LatheBufferGeometry

LatheGeometry

LineGeometry

LineSegmentsGeometry

OctahedronGeometry

ParametricGeometry

PlaneBufferGeometry

PlaneGeometry

PolyhedronGeometry

RingBufferGeometry

RingGeometry

ShapeGeometry

SphereBufferGeometry

SphereGeometry

TetrahedronGeometry

TextGeometry

TorusBufferGeometry

TorusGeometry

TorusKnotBufferGeometry

TorusKnotGeometry

TubeGeometry

WireframeGeometry

2.5.9 helpers

ArrowHelper

AxesHelper

Box3Helper

BoxHelper

CameraHelper

DirectionalLightHelper

FaceNormalsHelper

GridHelper

HemisphereLightHelper

PlaneHelper

PointLightHelper

PolarGridHelper

RectAreaLightHelper

SkeletonHelper

SpotLightHelper

VertexNormalsHelper

2.5.10 lights

AmbientLight

DirectionalLightShadow

DirectionalLight

HemisphereLight

LightShadow

Light

PointLight

RectAreaLight

SpotLightShadow

SpotLight

2.5.11 loaders

AnimationLoader

AudioLoader

BufferGeometryLoader

Cache

CompressedTextureLoader

CubeTextureLoader

DataTextureLoader

FileLoader

FontLoader

ImageBitmapLoader

ImageLoader

JSONLoader

Loader

LoadingManager

MaterialLoader

ObjectLoader

TextureLoader

2.5.12 materials

LineBasicMaterial

LineDashedMaterial

LineMaterial

Material

MeshBasicMaterial

MeshDepthMaterial

MeshLambertMaterial

MeshMatcapMaterial

MeshNormalMaterial

MeshPhongMaterial

MeshPhysicalMaterial

MeshStandardMaterial

MeshToonMaterial

PointsMaterial

RawShaderMaterial

ShaderMaterial

ShadowMaterial

SpriteMaterial

2.5.13 math

interpolants

CubicInterpolant

DiscreteInterpolant

LinearInterpolant

QuaternionLinearInterpolant

Box2

Box3

Cylindrical

Frustum

Interpolant

Line3

Math

Plane

Quaternion

Ray

Sphere

Spherical

Triangle

2.5.14 objects

Blackbox

Bone

CloneArray

Group

LOD

Line2

LineLoop

LineSegments2

LineSegments

Line

Mesh

Points

Skeleton

SkinnedMesh

Sprite

2.5.15 renderers

webgl

WebGLBufferRenderer

WebGLCapabilities

WebGLExtensions

WebGLGeometries

WebGLIndexedBufferRenderer

WebGLLights

WebGLObjects

WebGLProgram

WebGLPrograms

WebGLProperties

WebGLShader

WebGLShadowMap

WebGLState

WebGLRenderTargetCube

WebGLRenderTarget

2.5.16 scenes

FogExp2

Fog

Scene

2.5.17 textures

CompressedTexture

CubeTexture

DataTexture3D

DataTexture

DepthTexture

ImageTexture

TextTexture

Texture

VideoTexture

2.5.18 traits

2.6 Extending pythreejs

While you can do a lot with pythreejs out of the box, you might have some custom rendering you want to do, that would be more efficient to configure as a separate widget. To be able to integrate such objects with pythreejs, the following extension guide can be helpful.

2.6.1 Blackbox object

Pythreejs exports a `Blackbox` Widget, which inherits `Object3D`. The intention is for third-party widget libraries to inherit from it on both the Python and JS side. You would add the traits needed to set up your object, and have the JS side set up the corresponding three.js object. The three.js object itself would not be synced across the wire, which is why it is called a blackbox, but you can still manipulate it in a scene (transforming, putting it as a child, etc.). This can be very efficient e.g. for complex, generated objects, where the final three.js data would be prohibitively expensive to synchronize.

Example implementation

Below is an example implementation for rendering a crystal lattice. It takes a basis structure, and then tiles copies of this basis in x/y/z, potentially generating thousands of spheres.

Note: This example is not a good/optimized crystal structure viewer. It is merely used to convey the concept of a widget with a few parameters translating to something with potentially hugh amounts of data/objects.

Python:

```
import traitlets
import pythreejs

class CubicLattice(pythreejs.Blackbox):
    _model_name: traitlets.Unicode('CubicLatticeModel').tag(sync=True)
    _model_module = traitlets.Unicode('my_module_name').tag(sync=True)

    basis = traitlets.List(
        trait=pythreejs.Vector3(),
        default_value=[[0, 0, 0]],
        max_length=5
    ).tag(sync=True)

    repetitions = traitlets.List(
        trait=traitlets.Int(),
        default_value=[5, 5, 5],
        min_length=3,
        max_length=3
    ).tag(sync=True)
```

JavaScript:

```
import * as THREE from "three";

import {
    BlackboxModel
} from 'jupyter-threejs';


const atomGeometry = new THREE.SphereBufferGeometry(0.2, 16, 8);
const atomMaterials = [
    new THREE.MeshLambertMaterial({color: 'red'}),
    new THREE.MeshLambertMaterial({color: 'green'}),
    new THREE.MeshLambertMaterial({color: 'yellow'}),
    new THREE.MeshLambertMaterial({color: 'blue'}),
    new THREE.MeshLambertMaterial({color: 'cyan'}),
];

export class CubicLatticeModel extends BlackboxModel {
    defaults() {
        return {...super.defaults(), ...{
            _model_name: 'CubicLatticeModel',
            _model_module: 'my_module_name',
            basis: [[0, 0, 0]],
            repetitions: [5, 5, 5],
        }};
    }

    // This method is called to create the three.js object of the model:
    constructThreeObject() {
        const root = new THREE.Group();
        // Create the children of this group:
        // This is the part that is specific to this example
        this.createLattice(root);
        return root;
    }

    // This method is called whenever the model changes:
    onChange(model, options) {
        super.onChange(model, options);
        // If any of the parameters change, simply rebuild children:
        this.createLattice();
    }

    // Our custom method to build the lattice:
    createLattice(obj) {
        obj = obj || this.obj;

        // Set up the basis to tile:
        const basisInput = this.get('basis');
        const basis = new THREE.Group();
        for (let i=0; i < basisInput.length; ++i) {
            let mesh = new THREE.Mesh(atomGeometry, atomMaterials[i]);
            mesh.position.fromArray(basisInput[i]);
            basis.add(mesh);
```

(continues on next page)

(continued from previous page)

```
        }

        // Tile in x, y, z:
        const [nx, ny, nz] = this.get('repetitions');
        const children = [];
        for (let x = 0; x < nx; ++x) {
            for (let y = 0; y < ny; ++y) {
                for (let z = 0; z < nz; ++z) {
                    let copy = basis.clone();
                    copy.position.set(x, y, z);
                    children.push(copy);
                }
            }
        }

        obj.remove(...obj.children);
        obj.add(...children);
    }
}
```

This code should then be wrapped up in a widget extension (see documentation from ipywidgets on how to do this).

Usage:

```
import pythreejs
from IPython.display import display
from my_module import CubicLattice

lattice = CubicLattice(basis=[[0,0,0], [0.5, 0.5, 0.5]])

# Preview the lattice directly:
display(lattice)

# Or put it in a scene:
width=600
height=400
key_light = pythreejs.DirectionalLight(position=[-5, 5, 3], intensity=0.7)
ambient_light = pythreejs.AmbientLight(color='#777777')

camera = pythreejs.PerspectiveCamera(
    position=[-5, 0, -5],
    children=[
        # Have the key light follow the camera:
        key_light
    ],
    aspect=width/height,
)
control = pythreejs.OrbitControls(controlling=camera)

scene = pythreejs.Scene(children=[lattice, camera, ambient_light])

renderer = pythreejs.Renderer(camera=camera,
                              scene=scene,
```

(continues on next page)

(continued from previous page)

```
                                controls=[control],
                                width=width, height=height)

display(renderer)
```

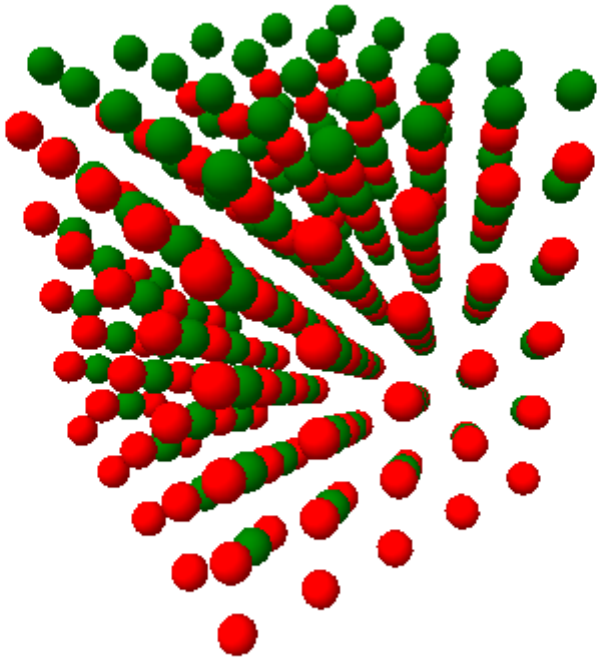

Fig. 1: Figure: Example view of the rendered lattice object.

2.7 Developer install

To install a developer version of pythreejs, you will first need to clone the repository:

```
git clone https://github.com/jupyter-widgets/pythreejs.git
cd pythreejs
```

Next, install it with a develop install using pip:

```
pip install -e .
```

If you are not planning on working on the JS/frontend code, you can simply install the extensions as you would for a *normal install*. For a JS develop install, you should link your extensions:

```
jupyter nbextension install [--sys-prefix / --user / --system] --symlink --py pythreejs

jupyter nbextension enable [--sys-prefix / --user / --system] --py pythreejs
```

with the appropriate flag. Or, if you are using Jupyterlab:

```
jupyter labextension link ./js
```