Chapter 5: Geometries and Coordinate Systems

K. Jünemann
Department Informations- und Elektrotechnik
HAW Hamburg

Content

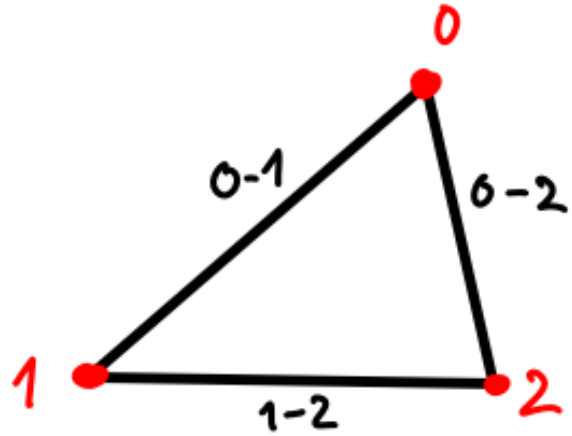
Chapter 5: Geometries and Coordinate Systems

- Defining geometries with vertices and faces
- Defining geometries in `three.js`
- Predefined geometries and loading of geometries
- Coordinate systems

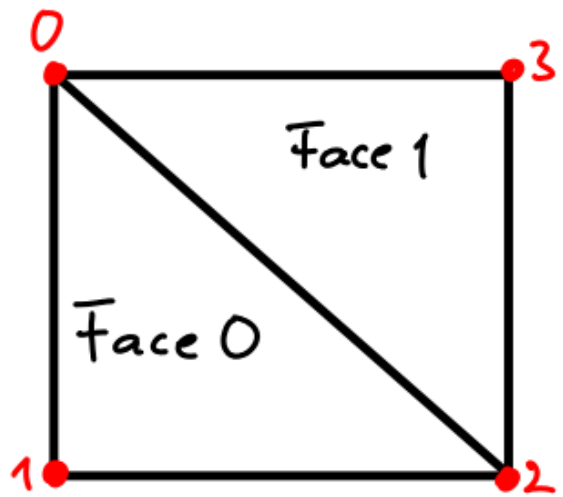
Defining geometries with vertices and faces

Basic shape of an object defined as a *mesh* (wireframe):

- *Mesh* : collection of faces.
- *Face* : triangle consisting of three connected vertices.

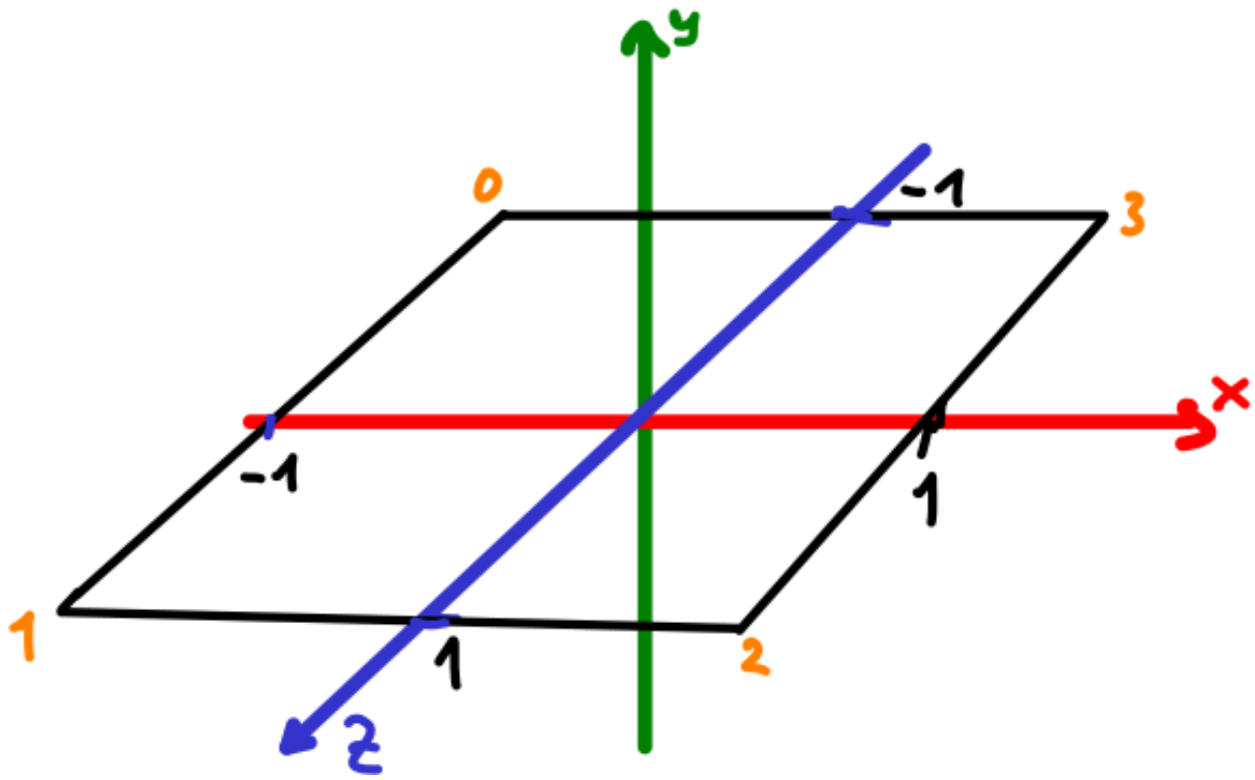


- Example 1: a square
  - Face 0:
    - Vertices 0,1,2
    - Edges: 0-1, 0-2, 1-2
  - Face 1:
    - Vertices 0,2,3
    - Edges: 0-2, 2-3, 3-0



Defining geometries with vertices and faces

Location of square in coordinate system



Defining geometries in `three.js`

- Geometries are represented by `THREE.BufferGeometry` class
  - constructed from set of vertices
  - can store further vertex attributes
  - class design determined by requirement that data are efficiently passed to GPU

- Side remark: Until early 2021 there was a `THREE.Geometry` class
  - more convenient but slower data storage
  - still there but deprecated
  - Lot's of code using it still out there

Defining geometries in `three.js`

Create square from example 1 with `three.js`:

```
// create empty wireframe:
const geo = new THREE.BufferGeometry();
// define 6 vertices: 3 for each face
const vertices = new Array(6);
vertices[0] = new THREE.Vector3(...);
...
vertices[5] = new THREE.Vector3(...);
geo.setFromPoints(vertices);

// use geometry to create proper object:
const obj = new THREE.Mesh(geo, material);
scene.add(obj);
```

- `THREE.Mesh` needs geometry *and* material (see below).
- `THREE.Mesh` is derived from `THREE.Object3D`.

Defining geometries in `three.js`

Number of vertices in geometry object: 3× number of faces!

- Vertices duplicated for each face
- memory wasted
- most flexible approach (see chapter 12)

Slightly alternative approach: *indexed geometry*

- stores each vertex only once
- defines faces in terms of vertex indices

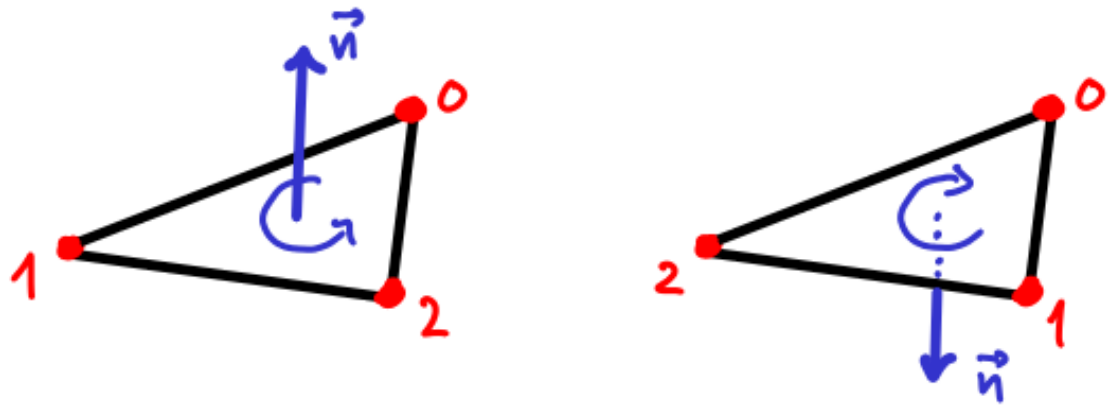
Defining geometries in `three.js`

Create square with indexed geometry:

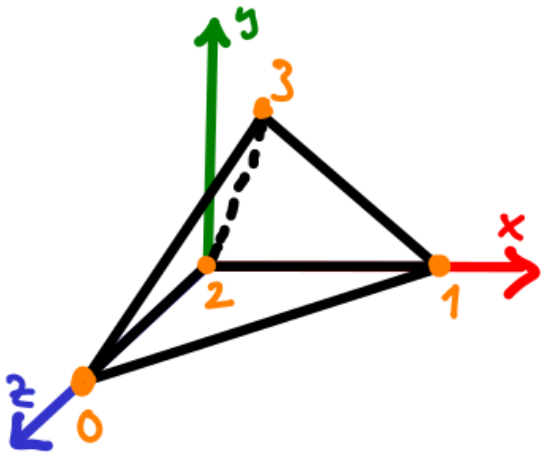
```
// create empty wireframe:
const geo = new THREE.BufferGeometry();
// define 4 vertices
const vertices = new Array(4);
vertices[0] = new THREE.Vector3(...);
...
vertices[1] = new THREE.Vector3(...);
geo.setFromPoints(vertices);    // as before
// define faces
const faceIndices = [0, 1, 2,
                     0, 2, 3];
geo.setIndex(faceIndices);
```

Defining geometries with vertices and faces

- Note: faces have an orientation (direction of normal vector).



- Example 2: a tetrahedron (pyramid with triangular base)

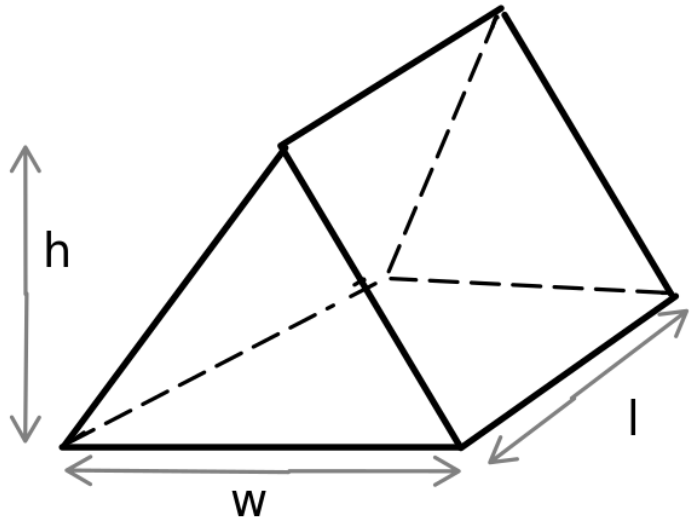


- Face 0: Vertices 0,2,1
- Face 1: Vertices 0,3,2
- Face 2: Vertices 1,2,3
- Face 3: Vertices 0,1,3

Exercise 1

Write a function `createRoof(l,w,h)` that returns a geometry object for a roof, given the following parameters:

- length `l` of roof
- width `w` of roof
- height `h` of roof



- Make sure all faces are correctly oriented.
- Use this function to create a red roof with `l=2`, `w=1` and `h=0.8`.

Predefined geometries

`three.js` provides a large number of predefined geometries:

- `BoxGeometry`
- `PlaneGeometry`
- `SphereGeometry`
- etc.

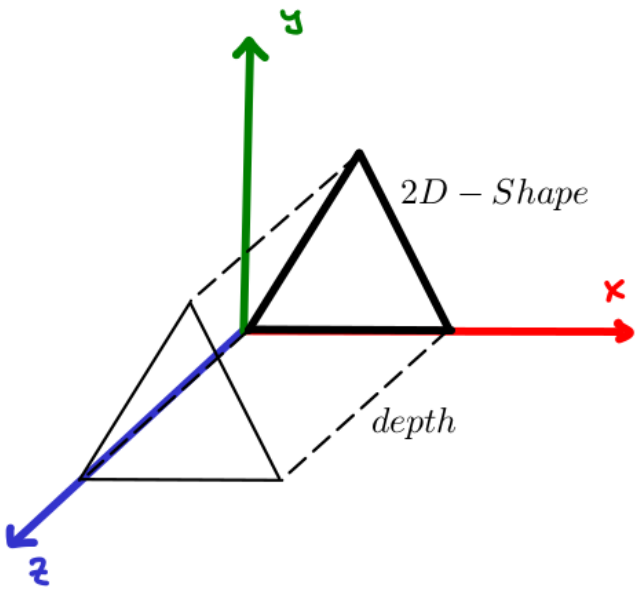
- Structure of `BufferGeometry` objects determined by GPU data requirements
  - data are passed to GPU with shader programs
- Before version 125 there was a `Geometry` object storing data in a more intuitive but slower way.

Predefined geometries

Useful tools to build your own geometry:

- `LatheGeometry`
  - creates volume of revolution around y axis
- `ExtrudeGeometry`
  - extrudes a 2D-shape in x-y-plane into z-direction.

Example: a roof with `ExtrudeGeometry`

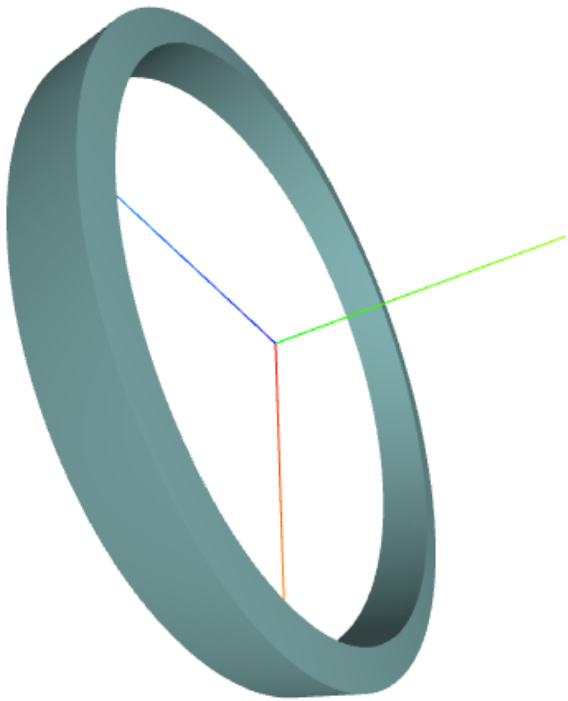


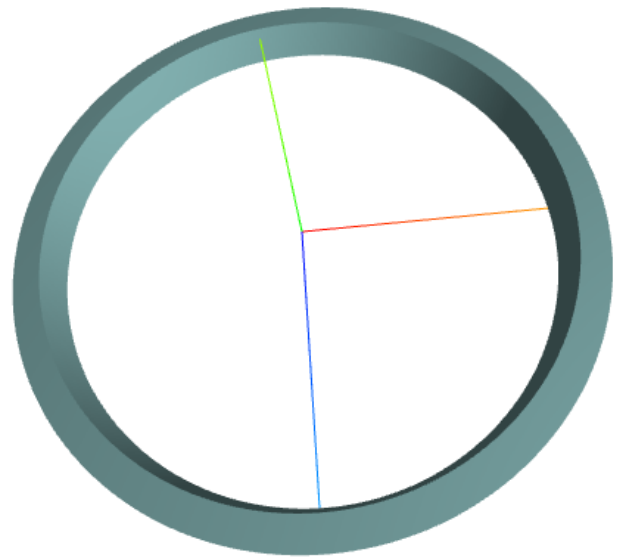

```
const base = new THREE.Shape();
base.moveTo(0,0);
base.lineTo(1,0);
base.lineTo(1/2,1);
base.lineTo( 0, 0 );

const extrudeOpts = {
  depth: 2,
  bevelEnabled: false,
};
const geo = new THREE.ExtrudeGeometry( base,
      extrudeOpts );
```

Exercise 2

1. Read the documentation of `LatheGeometry` and build a ring using a `LatheGeometry`.
2. Build another ring using `ExtrudeGeometry`.

   Hint: Use the `holes` property of the `Shape` class.





Predefined geometries

The famous Utah teapot is included in three.js library:

```
const teapotGeo =
new THREE.TeapotGeometry(0.5, 10);
const teapot = new THREE.Mesh(teapotGeo, mat);
scene.add(teapot);
```

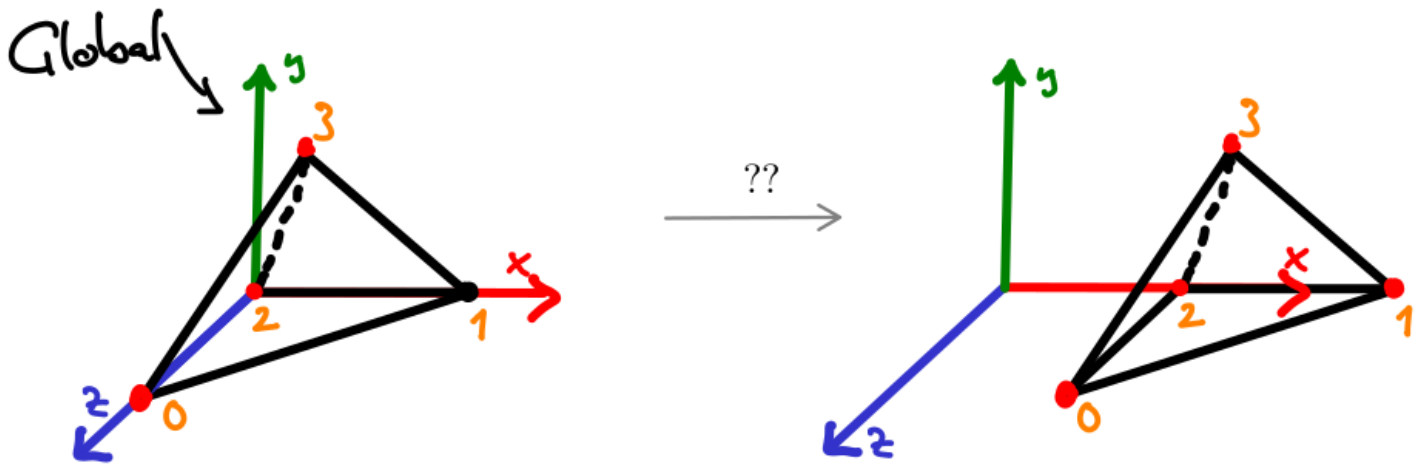
Loading of geometries

- Geometries can be loaded from external files.
- There exist many file formats, e.g. the Wavefront OBJ format (file extension `.obj`)
- `three.js` contains loaders for the most common file formats.
- Due to the *cross origin security policy* loading of external geometries require the application to run with a web server.

Coordinate systems

The *global* coordinate system:

- Provides a global frame of reference.
- Connected to `THREE.scene`.
- There's just *one* global coordinate system.
- Also called *world* coordinate system or *world space*.

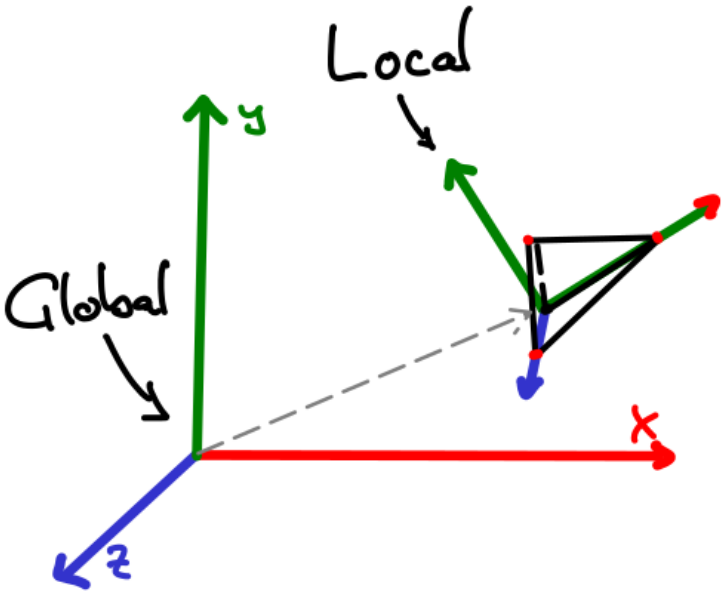


How can we change the position of an object in world space?

Coordinate systems

Each object defines its own *local* coordinate system:

- also called *object space*,
- moves around with object.



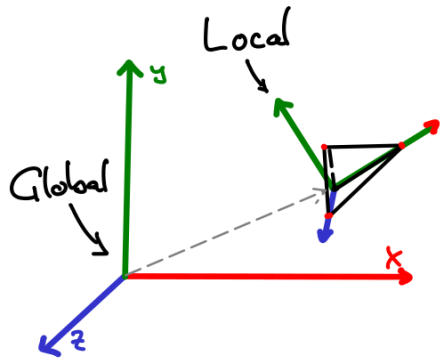
Location of local coordinate system *L* in global coordinate system *G* is specified by 6 parameters:

- *translation* of origin of *L* in *G* (3 parameters)
- *rotation* of *L* with respect to *G* (3 parameters)

Coordinate systems

Specification of local coordinate system of an object `obj` in `three.js`:

- `obj.position`: translation of origin, type: `Vector3`
- `obj.rotation`: rotation w.r.t. global coordinate system, type `Euler` (similar to `Vector3`)



Example: Position of some object

```
const obj = new THREE.Mesh(tetGeo, mat);
// move obj to (1,0,0):
obj.position.x = 1;
// rotate around local z axis:
obj.rotation.z = Math.PI/4;
```

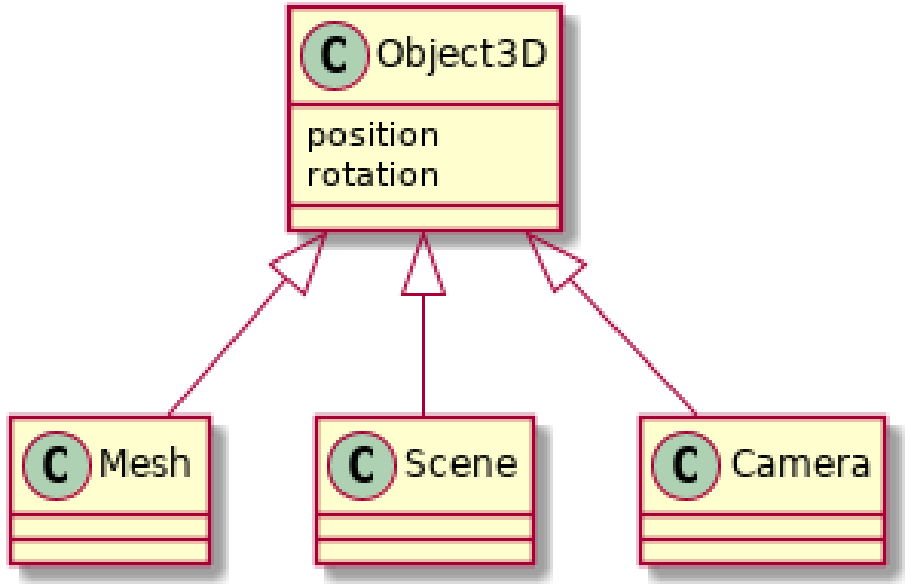
Coordinate systems

- Vertices of a `Mesh` object are defined w.r.t *local* coordinate system.
  - vertex coordinates don't change as object moves in global coordinate system.
- The coordinate system of an object `obj` can be visualized with

```
// len: length of axes
obj.add(new THREE.AxesHelper(len));
```

Coordinate systems

`three.js` class hierarchy: each object of type `Object3D` defines a coordinate system.



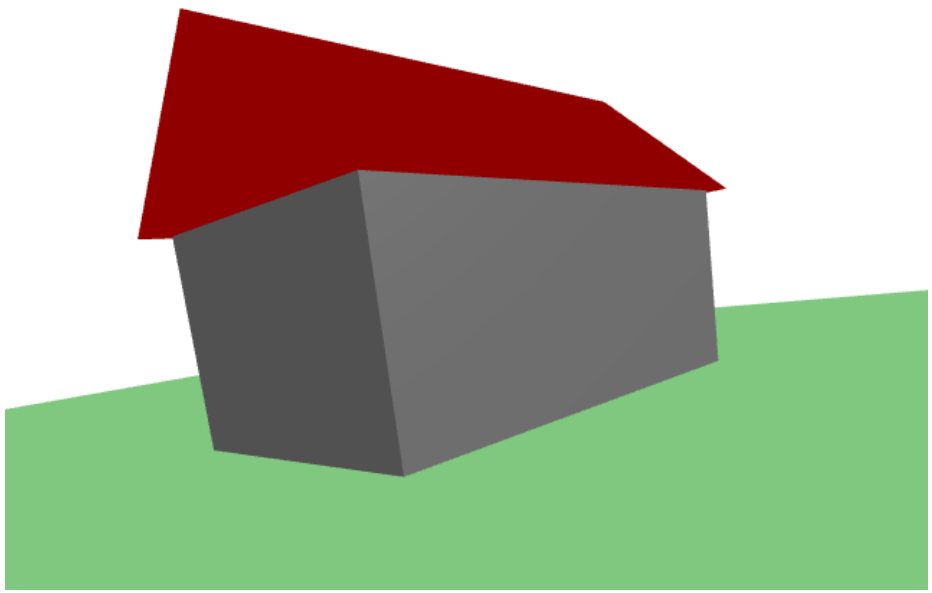
- *World space*: coordinate system of `scene` object
- *Object space*: coordinate system of any `Mesh` object
- A lot of other types derive from `Object3D`

Exercise 3: build a house

Use code template in `house` directory:

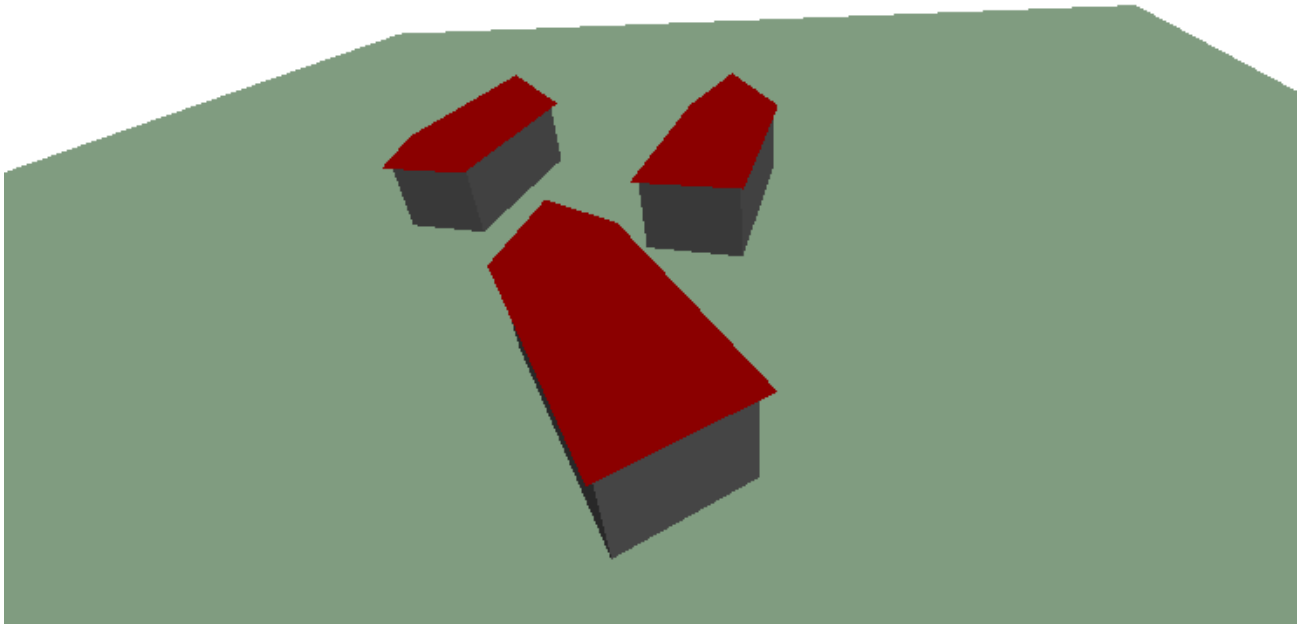
- Add a green plane representing the ground
  (use `THREE.PlaneGeometry`)
- Add a grey box representing the main body of the house.
  (use `THREE.BoxGeometry`)
- Add a red roof on top of the main body.

Use any material type you like



Coordinate systems

Example: A house consists of roof, body, etc.
How can the entire house be positioned?

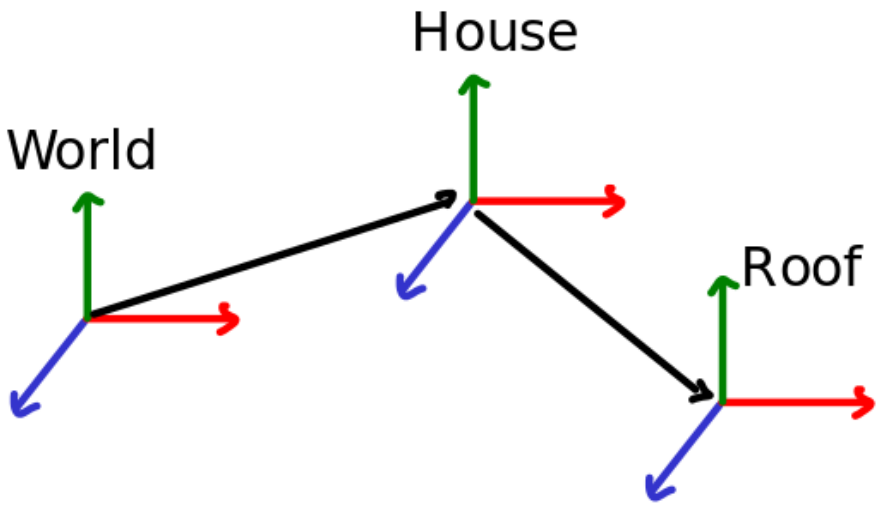


With a *hierarchy* of coordinate systems!

Coordinate systems

Hierarchy of coordinate systems for the house:

- an *abstract* house object defining a coordinate system for the entire house:
  - the house object is positioned in world space.
  - it is parent to all other parts of the house.
- the parts of the house are added to the abstract house object:
  - parts are positioned in house coordinate systems.
  - they are children of the house object.



Coordinate systems

The coordinate system hierarchy in `three.js`:

1. Abstract house of type `Object3D` (or `Group`):

```
const house = new THREE.Object3D();
scene.add(house); // scene: parent of house
house.position.x = ...    // w.r.t. scene
```

   `house.position` and `house.rotation` locate the house in world space.

2. The various parts are added to the house:

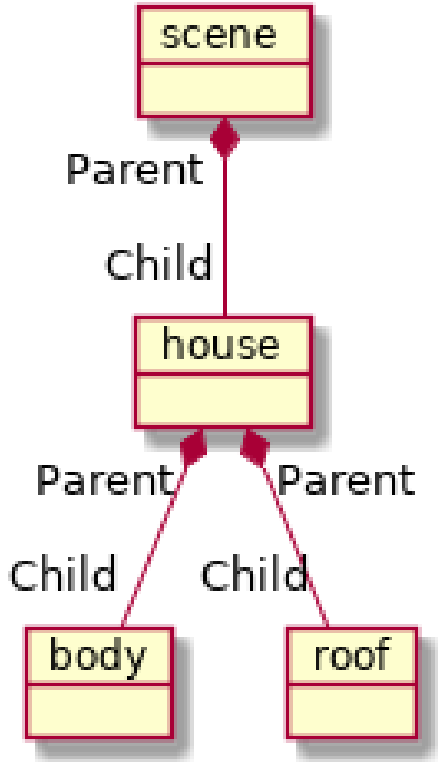
```
const roof = new THREE.Mesh(...);
house.add(roof);
roof.position.x = ...  // w.r.t to house
```

   `roof.position` and `roof.rotation` locate the roof in the house coordinate system.

Coordinate systems

Coordinate system hierarchy is parent - child relation:

- `scene` is parent of `house`, `house` is child of `scene`.
- `house` is parent of `roof`, `roof` is child of `house`.
- Same for `body`.




```
house.parent  // == scene
house.children  // array
```

Summary:

- `Object3D.add` establishes parent-child relation
- `Object3D.position` and `.rotation` relate to the parent coordinate system.
- You may use `Group` instead of `Object3D`.
  `Group` objects are return by geometry loaders

Exercise 4: A small village

- Use `Object3D.clone()` to create two more houses
- Position them as shown here: