Practical 3 - Hierarchical modeling

Objectives if this practical:

- improve our understanding of OpenGL coordinate frames
- deal with hierarchical scenes, such as an articulated kinematic chain

Pre-requisites:

- it is best but not mandatory to have completed Practical 2 - Meshes and modeling (practical2.html)

We provide a new **`viewer`** (_downloads/29de1d1695f00037f082b6f49fa09e5e/viewer2.py), with an implementation of the VertexArray and ColorMesh classes, along with an object loader and trackball implementation. You can check the correctness of your previous exercises against this implementation, and fill in this viewer for the exercises below. We provide a **`cylinder.obj`** (_downloads/a6a27cf5f07a525fbc97b79f569b670b/cylinder.obj) to build on in this practical. You will need the **`transform.py`** (_downloads/8779a270ec6c50ad0ccff91e33cd0cf0/transform.py) module provided in practical 1.

Graphics transforms

Until now we have dealt with independent static sets of objects and have largely reasoned in clipping coordinate space or with minor variations thereof. As seen in the main course, more transforms are routinely used to structure the scene around a world coordinate system, where scene objects, lights and camera viewpoints can all be expressed in a common coordinate frame. Coordinate frames are structured as follows:

(_images/vertex_shader_spaces.png)

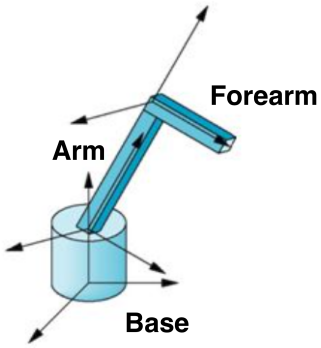
- Object coordinate frame is a self-centered coordinate system where the geometry of objects is defined. The center of gravity of the object, or its central point of contact to the ground is typically set to be the origin.

- The model transform is used to transform object coordinates to world coordinates, possibly reajusting its size. That transform is thus usually rigid (translation + rotation), possibly with a scale adjustment.
- The view matrix is the transform from world space point to camera (or eye) coordinates. It is a rigid transform which essentially expresses where the camera is placed in the scene.
- The projection matrix then transforms coordinates from eye coordinates to clip coordinates in $[0,1]^3$, where the geometry can be processed for depth testing and clipping against the view frustum.

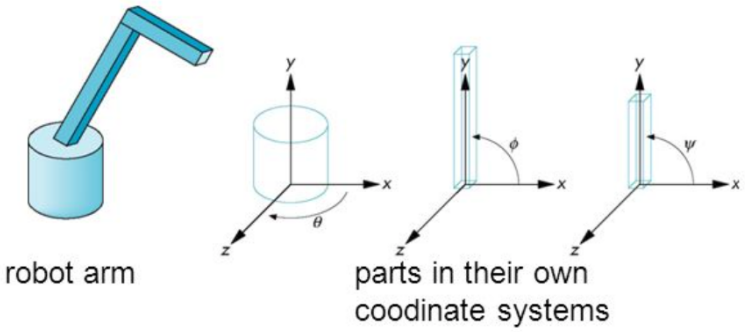
All these transforms are expressed as 4×4 matrices, which are simply to be composed (multiplied) and applied to object coordinates in the vertex shader to directly get their clip coordinates, as explained in Practical 1: 6. Projection Transform (practical1.html#projection).

Hierarchical transforms

The transform chain above can be further refined, by noting that many objects are defined in hierarchical fashion, where sub-objects are naturally expressed in the local coordinate frame of a parent object. This is the case for articulated objects with kinematic chains, such as this robotic arm, with a rotating base, an arm and a forearm:



(_images/robot_arm2.png)
Each object in the hierarchy has its own coordinate frame, expressed relative to its parent. In the case of the robotic arm, each transform relative to the parent node is parameterized by a single rotation:



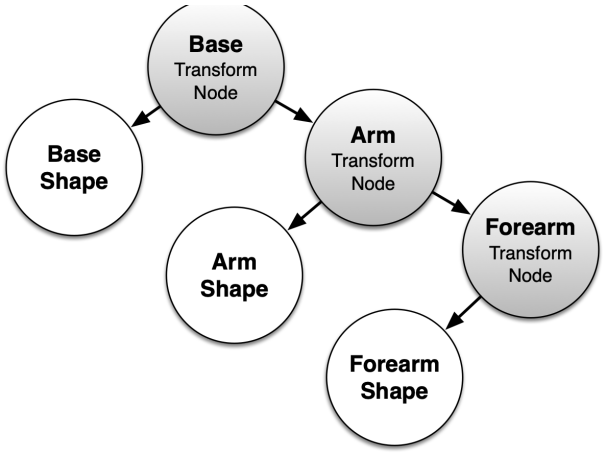
(_images/robot_arm.png)

Transform nodes

A practical way of representing hierarchies is to add a new type of ‘transform node’ object in our application that:

- is drawable (has a `draw()` method)
- may have children: it contains a list of drawable objects
- defines a new coordinate system relative to its parent, which affects all its children and descendants
- may be assigned some arbitrary parameters or properties useful for drawing all its children and descendant nodes

Using this new drawable type, we can then represent our robot arm with some cylinders and adequately placed transform nodes. In computer graphics this is called a *scene graph*:



(_images/robot_scene_graph.png)
We can write a Python **Node** class implementing this definition:

```
class Node:
    """ Scene graph transform and parameter broadcast node """
    def __init__(self, name='', children=(), transform=identity(), **param):
        self.transform, self.param, self.name = transform, param, name
        self.children = list(iter(children))

    def add(self, *drawables):
        """ Add drawables to this node, simply updating children list """
        self.children.extend(drawables)

    def draw(self, projection, view, model, **param):
        """ Recursive draw, passing down named parameters & model matrix. """
        # merge named parameters given at initialization with those given here
        param = dict(param, **self.param)
        model = ...   # what to insert here for hierarchical update?
        for child in self.children:
            child.draw(projection, view, model, **param)
```

- Children drawables can be added to a **Node** object through the `children` parameter of its constructor or the **add()** method.
- The `name` parameter is optionally useful for tagging an object, which we will use later on.
- The **draw()** method basically passes down all it receives to its children. The `projection` and `view` matrices are passed unchanged, the `model` matrix is hierarchically updated. It assumes all three matrices are multiplied in the vertex shader of children drawables to transform their object coordinates, as explained in Practical 1: 6. Projection Transform (practical1.html#projection).
- Any additional named parameters passed to **draw()** or to the node constructor is captured in their argument dictionaries `param` ; both are merged in the **draw()** method to be broadcast to all its descendants through their **draw()** call (usage will become clear during future exercises).

Exercises

1. Compute node model transform

On the highlighted line above in the **Node** class, how should the `model` matrix be computed from the passed parent `model` matrix transform and the `self.transform` relative to the parent, for the hierarchical transforms to be passed down the scene graph tree?

2. Using Node to rescale an object

To make a robot arm, we need to reshape our cylinder, to make thin cylinder arms, and a flat cylindric robot base. Introduce a Node with a scale transform to do so:

```
def main():
        viewer = Viewer()

        # construct our robot arm hierarchy for drawing in viewer
        cylinder = Cylinder()
        limb_shape = Node(transform=scale(...))     # make a thin cylinder
        limb_shape.add(cylinder)                    # scaled cylinder shape

        viewer.add(limb_shape)
```

For your robot arm, you need to make three such shapes, the arm, forearm, and base shapes. Use the knowledge from above for the three shapes:

```
def main():
        viewer = Viewer()

        # ---- let's make our shapes ------------------------------------
        # think about it: we can re-use the same cylinder instance!
        cylinder = Cylinder()

        # make a flat cylinder
        base_shape = Node(transform=scale(...))
        base_shape.add(cylinder)                    # shape of robot base

        # make a thin cylinder
        arm_shape = Node(transform=scale(...))
        arm_shape.add(cylinder)                     # shape of arm

        # make a thin cylinder
        forearm_shape = Node(transform=scale(...))
        forearm_shape.add(cylinder)                 # shape of forearm

        viewer.add(...)
        viewer.run()
```

3. The robot arm hierarchy

But wait a minute, all these cylinders are centered on the origin of the object coordinate system! If we want the rotation axis of the arm and forearm to be at the lower extremity of the cylinder, we need to apply a translation to realign the cylinder such that its lower extremity is at the origin:

```
def main():
        viewer = Viewer()

        # ---- let's make our shapes ------------------------------------
        # think about it: we can re-use the same cylinder instance!
        cylinder = Cylinder()

        # make a flat cylinder
        base_shape = Node(transform=scale(...))
        base_shape.add(cylinder)                    # shape of robot base

        # make a thin cylinder with lower base at origin
        arm_shape = Node(transform=translate(...) @ scale(...))
        arm_shape.add(cylinder)                     # shape of arm

        # make a thin cylinder with lower base at origin
        forearm_shape = Node(transform=translate(...) @ scale(...))
        forearm_shape.add(cylinder)                 # shape of forearm

        viewer.add(...)
        viewer.run()
```

Now we build our actual robot arm hierarchy by adding the code below. For this we create the transform nodes as they appeared in the hierarchy figure. Note that the forearm needs an additional translation because it needs to hinge at the tip of the arm:

```
def main():
    viewer = Viewer()

    # ---- let's make our shapes ------------------------------------
    base_shape = ...
    arm_shape = ...
    forearm_shape = ...

    # ---- construct our robot arm hierarchy ----------------------------
    theta = 45.0        # base horizontal rotation angle
    phi1 = 45.0         # arm angle
    phi2 = 20.0         # forearm angle

    transform_forearm = Node(transform=translate(...) @ rotate((...), phi2))
    transform_forearm.add(forearm_shape)

    transform_arm = Node(transform=rotate((...), phi1))
    transform_arm.add(arm_shape, transform_forearm)

    transform_base = Node(transform=rotate((...), theta))
    transform_base.add(base_shape, transform_arm)

    viewer.add(...)
    viewer.run()
```

- Fill in the rotations for the arm rotation parametrization, in `transform_forearm`, `transform_arm` and `transform_base`
- Try different rotations for your arm angle parameters and see if you get consistent mechanical poses

Optional Exercices

The exercises below can be performed in any order.

4. Keyboard control

Level: easy

Assign some key handlers to control each degree of freedom in rotation of your robot arm. A nice way of doing this is to create a new type of Node dedicated to keyboard control of a rotation, to use in your arm code:

```
class RotationControlNode(Node):
    def __init__(self, key_up, key_down, axis, angle=0, **param):
        super().__init__(**param)   # forward base constructor named arguments
        self.angle, self.axis = angle, axis
        self.key_up, self.key_down = key_up, key_down

    def draw(self, projection, view, model, win=None, **param):
        assert win is not None
        self.angle += 2 * int(glfw.get_key(win, self.key_up) == glfw.PRESS)
        self.angle -= 2 * int(glfw.get_key(win, self.key_down) == glfw.PRESS)
        self.transform = ...

        # call Node's draw method to pursue the hierarchical tree calling
        super().draw(projection, view, model, win=win, **param)
```

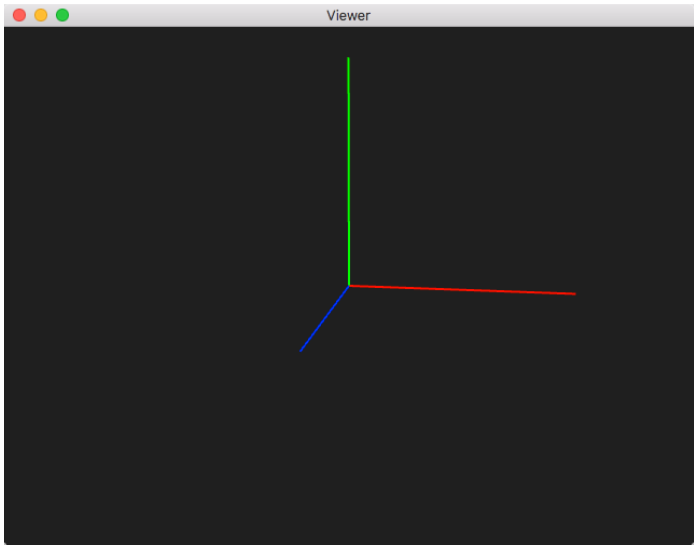
You can use this as you would use any node, by passing it the intended control keys, for example:

```
node = RotationControlNode(glfw.KEY_LEFT, glfw.KEY_RIGHT, vec(0, 1, 0))
```

5. Debug a hierarchy

Level: medium

It is quite useful to be able to display the basis vectors of a reference frame. For this you can create your own axis drawing object, using a **`VertexArray`** or deriving from **`ColorMesh`** . For this you can use `GL.GL_LINES` as primitive argument of your vertex array draw calls: this tells OpenGL to assemble vertices and attributes in the vertex array by forming a line for each consecutive pair of vertex attributes in the array. Draw each basis vector x, y, z with the corresponding unit line segment of color red, green, and blue.



(_images/axis.png)

Expected axis object drawn in clip coordinates, slightly rotated from its initial position so we can see the z axis in blue.

There is a very useful trick to debug a scene graph hierarchy and check that coordinate frames are in place, by adding an **`Axis`** object for every node. All you need to do is add this at the end of the **`Node`** constructor:

```
class Node:
    ...
    def __init__(self, name=None, children=(), transform=identity(), **param):
        ...
        self.add(Axis())
```

6. Refactoring

Level: easy

Hey, doesn't the **`Viewer`** class behave like a **`Node`** too, with its **`add()`** method and `for` loop to draw our drawables? Indeed the Viewer can be considered the root node of the scene graph. Refactor and simplify it by deriving it from node !

Back to top