Projection Comparisons

September 3, 2025

1 Projection Comparisons

Arthur Ryman, lasted updated 2025-09-03

1.1 Introduction

The goal of this notebook is to analyze the current semantics of some classes that govern the mapping from model space to scene space. My goal is to answer the following questions.

1.1.1 Q1: Should Projection take a camera location as an init parameter?

The init parameters for Projection class are called scene_x, scene_y, and camera_z, which makes them look dissimilar, but are they in fact components of a camera position vector in model space?

1.1.2 Q2: Does PuzzleCube3D need the init parameter cube_centre?

The init parameters for PuzzleCube3D includes cube_centre. Does it really give additional expressibility or can its effect be achieved by chaning the projection?

1.1.3 Q3: Does Puzzle3D need the init parameter cube_one_centre?

Similarly, the init parameters for Puzzle3D include projection and cube_one_centre, but can cube_one_centre be eliminated by chosing a different projection.

1.1.4 Manim Voiceover UserWarning

The manim_voiceover module is using a harmless deprecated feature which generates a distracting UserWarning when I import manim. We can safely ignore it.

[1]:
```
import warnings
warnings.filterwarnings("ignore", category=UserWarning,␣
  ↪module="manim_voiceover")
```

2 The Projection Class

Source code: projection.py

Here is its init method:

```
[2]: from listings.list_python import lst
     from instant_insanity.core.projection import Projection

     lst(Projection.__init__)
```

```
    def __init__(self, scene_x: float = 0.0, scene_y: float = 0.0, camera_z: float = 2.0,
        self.scene_x = scene_x
        self.scene_y = scene_y
        self.camera_z = camera_z
        self.scale = scale

        scene_origin: Point3D = scene_x * RIGHT + scene_y * UP + camera_z * OUT
        scene_per_model: float = scale
        self.conversion = ModelToSceneConversion(scene_origin, scene_per_model)

```

It is clear from the listing that the code is indeed treating the scene_x, scene_y, and camera_z as the components of a vector. It remains to confirm that the projection does in fact map this vector to the origin of scene space.

2.0.1 Using SymPy for Verification

The Projection class, and all other classes I have created for use with Manim, use NumPy to represent points in both model space and scene space. I could therefore write a suite of NumPy test cases to verify that the scene_origin vector in model space does indeed get mapped to the origin of scene space.

However, that would only verify the behaviour in a finite number of cases and would not lead to a clearer understanding of the code. Given that projections are fairly mathematical, it might be feasible to verify the behaviour in all cases by using symbolic exection of the code. Therefore, I am going to try using SymPy to verify the code symbolically.

2.1 Symbolic Projection

The Projection class and friends work on NumPy arrays. I have therefore created parallel versions of the code that work on SymPy objects.

Source code: symbolic_projection.py

For example, here's the init method for the symbolic version of Projection.

```
[3]: import instant_insanity.core.symbolic_projection as sp

     lst(sp.Projection.__init__)
```

```
    def __init__(self,
                 scene_x: Scalar = S.Zero,
                 scene_y: Scalar = S.Zero,
                 camera_z: Scalar = S.Zero,
                 scale: Scalar = S.One) -> None:
```

```
        self.scene_x = scene_x
        self.scene_y = scene_y
        self.camera_z = camera_z
        self.scale = scale

        scene_origin: Vector = scene_x * UNIT_I + scene_y * UNIT_J + camera_z * UNIT_K
        scene_per_model: Scalar = scale
        self.conversion = ModelToSceneConversion(scene_origin, scene_per_model)

```

Now let's create some SymPy variables to use as init parameters.

[4]:
```
from sympy import *
from instant_insanity.core.symbolic_projection import Scalar, Vector

def scalar(name: str) -> Scalar:
    return symbols(name, real=True)

def positive_scalar(name: str) -> Scalar:
    return symbols(name, real=True, positive=True)

def vector(name: str) -> Vector:
    return Matrix(symbols(name + '1:4', real=True))

scene_x = scalar('scene_x')
scene_y = scalar('scene_y')
camera_z = scalar('camera_z')
scale = positive_scalar('scale')

Matrix([scene_x, scene_y, camera_z, scale]).T
```

[4]: $\begin{bmatrix} scene_x & scene_y & camera_z & scale \end{bmatrix}$

Projection is an abstract base class so we cannot instantiate it directly. We need to use one of its concrete subclasses. Here's the init method for the symbolic version of OrthographicProjection:

[5]:
```
lst(sp.OrthographicProjection)
```

```
class OrthographicProjection(Projection):
    """This class models an orthographic projection.

    Attributes:
        u: A unit Vector that specifies the direction of the projection.
    """

    u: Vector

    def __init__(self, u: Vector, **kwargs) -> None:
        """Initializes an orthographic projection object.

        Args:
            u: A unit vector that specifies the direction of the projection.

        Raises:
            TypeError: if u is not a unit Vector.
            ValueError: if the z-component of u is zero.
        """
        assert isinstance(u, Matrix)
        assert u.shape == (3, 1)

        super().__init__(**kwargs)

        norm_u: Scalar = simplify(u.norm())
        assert norm_u == S.One

        u_z: Scalar
        _, _, u_z = u
        if u_z == S.Zero:
            raise ValueError('unit vector z-component is zero')

        self.u = u

    def compute_u(self, model_point: Vector) -> Vector:
        assert isinstance(model_point, Matrix)
        assert model_point.shape == (3, 1)

        return self.u

```

The model defines a standard orthographic projection.

[6]:
```
lst(sp.mk_standard_orthographic_projection)
```

```
def mk_standard_orthographic_projection() -> OrthographicProjection:
    direction: Vector = Matrix([Rational(3, 2), S.One, Integer(5)])
    u: Vector = direction / direction.norm()
    projection: OrthographicProjection = OrthographicProjection(u,
                                                                scale=Rational(1, 2),
                                                                scene_x=Integer(2),
                                                                scene_y=Integer(-3),
                                                                camera_z=S.One)
    return projection

```

Create an instance of the standard orthographic projection.

[7]:
```
sop = sp.mk_standard_orthographic_projection()

sop.u.T
```
[7]:
$$\begin{bmatrix}\frac{3\sqrt{113}}{113} & \frac{2\sqrt{113}}{113} & \frac{10\sqrt{113}}{113}\end{bmatrix}$$

[8]:
```
sop.conversion.scene_origin.T
```
[8]:
$$\begin{bmatrix}2 & -3 & 1\end{bmatrix}$$

Map the scene origin to scene space.

[9]:
```
model_camera_origin = sop.project_point(sop.conversion.scene_origin)

model_camera_origin.T
```
[9]:
$$\begin{bmatrix}0 & 0 & \frac{1}{2}\end{bmatrix}$$

The putative camera origin in scene space does not get mapped to the origin of model space. I think this is confusing.

Let's look at a totally generic OrthographicProjection and see where it maps the putative camera origin.

An OrthographicProjection takes a unit vector as an init parameter. We need to create a generic unit vector. The best way to do that is to specify its spherical polar coordinates.

[10]:
```
theta = scalar('theta')
phi = scalar('phi')

Matrix([theta, phi]).T
```
[10]:
$$\begin{bmatrix}\theta & \phi\end{bmatrix}$$

[11]:
```
u_x = sin(theta) * cos(phi)
u_y = sin(theta) * sin(phi)
u_z = cos(theta)
u = Matrix([u_x, u_y, u_z])

u.T
```
[11]:
$$\begin{bmatrix}\sin(\theta)\cos(\phi) & \sin(\phi)\sin(\theta) & \cos(\theta)\end{bmatrix}$$

[12]:
```
u.norm()
```
[12]:
$$\sqrt{\sin^2(\phi)\sin^2(\theta) + \sin^2(\theta)\cos^2(\phi) + \cos^2(\theta)}$$

[13]:
```
simplify(u.norm())
```
[13]:
1

[14]:
```
trigsimp(u.norm())
```
[14]:

1

[15]:
```
op = sp.OrthographicProjection(u, scene_x=scene_x, scene_y=scene_y,␣
 ↪camera_z=camera_z, scale=scale)

op.u.T
```

[15]:
$$\begin{bmatrix}\sin(\theta)\cos(\phi) & \sin(\phi)\sin(\theta) & \cos(\theta)\end{bmatrix}$$

[16]:
```
op.conversion.scene_origin.T
```

[16]:
$$\begin{bmatrix}scene_x & scene_y & camera_z\end{bmatrix}$$

[17]:
```
scene_origin = op.conversion.scene_origin

projected_scene_origin = op.project_point(scene_origin)

projected_scene_origin.T
```

[17]:
$$\begin{bmatrix}0 & 0 & camera_z scale\end{bmatrix}$$

In general, the projection of the scene origin has a nonzero z-component. This seems wrong. I think it would be more intuitive if the model space scene origin mapped to the origin of scene space. Furthermore, what we call the scene origing in model space should be called the camera origin as an extension of using camera_z.

I also think that when we create 3D objects, their natural centres should be located at the origin of model space, and that further positioning and scaling should be done by the projection by setting the camera origin.

2.2 PerspectiveProjection

Create a generic perspective projection and see where the scene origin gets mapped to.

Here's the initializer:

[18]:
```
lst(sp.PerspectiveProjection.__init__)
```

```
    def __init__(self, viewpoint: Vector, **kwargs) -> None:
        """Initialize the projection.

        Args:
            viewpoint: The position of the viewpoint.

        Raises:
            TypeError: if viewpoint is not a Vector.

        """
        assert isinstance(viewpoint, Matrix)
        assert viewpoint.shape == (3, 1)

        super().__init__(**kwargs)
        self.viewpoint = viewpoint

```

[19]:
```
viewpoint = vector('v')

viewpoint.T
```

[19]: $\begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$

[20]:
```
pp = sp.PerspectiveProjection(viewpoint, scene_x=scene_x, scene_y=scene_y,␣
 ↪camera_z=camera_z, scale=scale)

pp.viewpoint.T
```

[20]: $\begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$

[21]:
```
model_scene_origin_pp = pp.project_point(scene_origin)

model_scene_origin_pp.T
```

[21]: $\begin{bmatrix} 0 & 0 & camera_z scale \end{bmatrix}$

[22]:
```
pp.conversion.scene_origin.T
```

[22]: $\begin{bmatrix} scene_x & scene_y & camera_z \end{bmatrix}$

At least the projections are consistent.

2.3 Action

Look at the code and decide if it make sense to interpret the so-called scene origin as the point in model space that maps to the origin of scene space. Should this point be called the camera origin?