Assignment 3: Matrices, Frames, Scene Graphs

Weightage: ***3.6%*** (1/11 of the 40% assignment component)

Deadline: Friday 25^{th} September 2026, **11:59 pm**

Introduction

A2 moved points one at a time. That does not scale once a shape has to sit somewhere, turned and scaled, with its parts coming along. This week you move *space* instead, in form of a **scene graph**.

Objectives

- Representing 2D transforms as 3×3 matrices, read by their columns
- Using homogeneous coordinates to make translation a matrix
- Composing transforms in the right order, the rightmost acts first
- Inverting an affine transform to change frame of reference
- Traversing a hierarchy so parts inherit their parents' motion

Collaboration and AI Policy

All work is individual. Students are NOT permitted to look at or copy each other's code, code structure, or scenes. If caught, both parties receive a 0 in A3 and the case is reported.

Per the course policy: AI assistants may be used to *explain* concepts and to *debug code you have already written*, but not to generate solutions or scenes. Whatever you use, disclose it in the header of `mat3.py`, `shapes.py` and `scene.py`. You must be able to explain, and modify on request, any line you submit.

Getting Started

A) File Setup

Download the starter folder and keep **all files in one directory**:

`mat3.py`	(edit this) your transform library — **T1–T8 go here**
`shapes.py`	(edit this) your transformed geometry — **T9–T12 go here**
`scene.py`	(edit this) your articulated scene — **all Part B work goes here**
`vec2.py`	*do not edit* — from A2
`canvas.py`	*do not edit* — the `Canvas`/`Layer` classes
`curves.py`	*do not edit* — from A2
`painter.py`	*do not edit* — from A1
`encode.py`	*do not edit* — turns your frames into a video
`check.py`	*do not edit* — the autograder; run it as often as you like
`expected_a3.py`	*do not edit* — reference data used by `check.py`

Three of those files are the reference solutions to A1 and A2. Read them and compare against what you submitted, quietly, for your own benefit. You are not graded on them and you must not edit them.

As in A1 and A2, there is **nothing to install**: A3 is standard-library Python throughout — no `pip`, no virtual environment, no third-party packages. Any Python 3.9+ interpreter works, **PyPy** included, and PyPy is worth using here: Part B renders around a hundred frames of a scene graph, and its JIT makes that several times faster. The one external tool is `ffmpeg`, exactly as in A2:

```
python3 encode.py --check          # confirm ffmpeg is still reachable
```

B) The Matrix Model

Everything in Part A below works with the same two data types:

- A **vector** is a `Vec2`, exactly the class you wrote in A2 — `.x` and `.y`, `a + b`, `v * k`, `v.dot(w)`, `v.normalized()`, `v.perp()`, `v.rounded()`. The finished version is given to you this week.
- A **matrix** is a `Mat3`, holding **9 floats in row-major order**:

$$\texttt{(a, b, c, d, e, f, g, h, i)} \quad \Longleftrightarrow \quad \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

so entry (row, col) is `m[3*row + col]`, and the given `m.at(row, col)` does that arithmetic for you. Both spellings read the same nine numbers; use `at` when you are thinking in rows and columns and plain indexing when you are thinking in flat offsets.

Like `Vec2`, a `Mat3` is **immutable**: every operation returns a new matrix. The constructor, indexing, `at`, `format_matrix` and printing are **given**; every operation that does transform mathematics is yours, and that is what T1–T8 are.

The `Mat3` API at a glance. The left column is **given** and already works; the right column is **yours**.

Given (plumbing)	Yours (T1–T8)
`Mat3(a,b,c, d,e,f, g,h,i)`	`Mat3.identity()  m @ n`
`m[i]  m.at(row, col)`	`Mat3.translation(tx, ty)`
`m.format_matrix()`	`Mat3.scaling(sx, sy)`
`m == n  repr(m)`	`Mat3.rotation(theta)  Mat3.shear(..)`
`mat3.transform_shape(m, s)`	`m.apply_point(p)  m.apply_vector(v)`
`Node(transform, draw, name)`	`m.apply_points(pts)  m.inverse()`
`node.add(child)`	`mat3.compose(A, B, C, ...)`
	`Mat3.rotation_about(theta, pivot)`
	`Mat3.scaling_about(sx, sy, pivot)`
	`m.to_local(p)  m.point_in_box(..)`
	`mat3.world_matrix(node)`
	`mat3.draw_tree(node, img)`

Note which names are `Mat3.something` and which are `mat3.something`. The **static methods** on the class build a new matrix out of numbers (`Mat3.rotation(0.5)`); the **free functions** in the module work on several matrices or on a scene graph (`compose`, `world_matrix`, `draw_tree`). Everything else is a method on a matrix you already have.

Part A — Transforms and Frames (70 pts)

Implement T1–T8 in `mat3.py` and T9–T12 in `shapes.py`. Each function's docstring in the starter repeats its exact contract. Check yourself anytime:

```
python3 check.py
```

The checker runs **value cases** (comparing what your functions return, within 10^{-6}) and **pixel cases** (comparing what your functions draw, exactly), as in A2. **Grading runs additional hidden cases**: passing the checker is necessary, not sufficient — hardcoding its expected values earns a 0 for the task.

T1–T8 below are written in `mat3.py`. Open that file now — the stubs are in the same order as the tasks here, and each docstring repeats the contract you are reading. Nothing in this file draws anything or imports anything beyond `math`: it is pure arithmetic on nine floats.

T1 · `identity` and `@` · 6 pts

`Mat3.identity()` is the transform that changes nothing — ones down the diagonal, zeros elsewhere. It is the "empty" transform, and it is what you start from when folding a list of matrices together. It is a `@staticmethod`, so you call it on the class: `Mat3.identity()`, not `m.identity()`.

`m @ n` is the matrix product mn, written as Python's matrix multiplication operator (you implement `__matmul__`). Entry (r, c) of the result is the dot product of row r of `m` with column c of `n`:

$$\texttt{result.at(r, c)} = \sum_{k=0}^{2} \texttt{m.at(r, k)} \cdot \texttt{n.at(k, c)}$$

Write it as a double loop, not as nine typed-out expressions.

T2 · the basic transforms · 5 pts

Four constructors, one matrix each:

$$T = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

plus `shear(shx, shy)`, which maps $x' = x + \texttt{shx}\, y$ and $y' = y + \texttt{shy}\, x$.

Implement these as four `@staticmethod`s on `Mat3`: `Mat3.translation`, `Mat3.scaling`, `Mat3.rotation` and `Mat3.shear`. They are static because they build a matrix out of thin air rather than modifying an existing one.

T3 · `apply_point` and `apply_vector` · 6 pts

Both multiply the matrix by a column, and the only difference is what goes in the third slot, but that difference is the entire task:

$$\underbrace{\begin{pmatrix} x \\ y \\ \mathbf{1} \end{pmatrix}}_{\text{a } \textbf{point}\text{: a location}} \qquad \underbrace{\begin{pmatrix} x \\ y \\ \mathbf{0} \end{pmatrix}}_{\text{a } \textbf{vector}\text{: a direction}}$$

With $w = 1$ the last column is picked up and the result translates. With $w = 0$ it is not, and the result does *not* translate.

A2 told you that a point and a direction are the same type, and that the difference lives in your head. **Here it becomes real.** "The car is at $(300, 120)$" moves when the world moves. "The car is heading north-east" does not. Translating a heading is a bug, and it is a quiet one, your object ends up facing a direction that drifts as it travels.

Rule of thumb: positions and vertices are points; headings, normals, velocities and axes are vectors.

T4 · `compose` · 5 pts

Fold any number of matrices into one, left to right:

$$\texttt{compose(A, B, C)} = \texttt{A @ (B @ C)} = ABC$$

so `C` is applied to the point **first** and `A` last. **Read a compose call right to left and you are reading the order things happen in.** With no arguments, return `identity()` — the empty product, exactly as an empty sum is 0.

`compose` stays a *free function* in `mat3.py` rather than a method, because it takes any number of matrices — including none.

T5 · `inverse` · 8 pts

`m.inverse()` returns the matrix that undoes `m`.

You may **assume `m` is affine** — bottom row exactly $(0, 0, 1)$. Write `m` as a linear part L and a translation $\mathbf{t}$, and with $\det = ae - bd$ return:

$$L = \begin{pmatrix} a & b \\ d & e \end{pmatrix} \qquad L^{-1} = \frac{1}{\det} \begin{pmatrix} e & -b \\ -d & a \end{pmatrix} \qquad \texttt{m.inverse()} = \begin{pmatrix} L^{-1} & -L^{-1}\mathbf{t} \\ 0\ 0 & 1 \end{pmatrix}$$

Required behaviour:

- If $|\det| <$ `EPSILON`, return `Mat3.identity()`. Do not raise, do not divide.
- Otherwise `m @ m.inverse()` must equal `Mat3.identity()` to within 10^{-6}.

Do not implement general 3×3 inversion; the affine case above is what is graded.

T6 · transforms about a pivot · 5 pts

`Mat3.rotation()` spins the plane about $(0, 0)$. To spin about some other point, use the **sandwich** — and once you see it here you will see it everywhere in graphics:

1. translate the pivot to the origin $T(-\mathbf{p})$
2. do the transform there $R(\theta)$
3. translate back $T(+\mathbf{p})$

$$R_{\mathbf{p}}(\theta) \;=\; T(\mathbf{p})\, R(\theta)\, T(-\mathbf{p})$$

```
compose(Mat3.translation(px, py), Mat3.rotation(theta),
        Mat3.translation(-px, -py))
```

Get the two translations backwards, or drop them, and your object will orbit the origin instead of spinning in place — a distinctive and memorable bug, and the one Figure 1 warns about.

`Mat3.scaling_about` is the identical sandwich with S in the middle. Both are static methods, like the transforms they wrap.

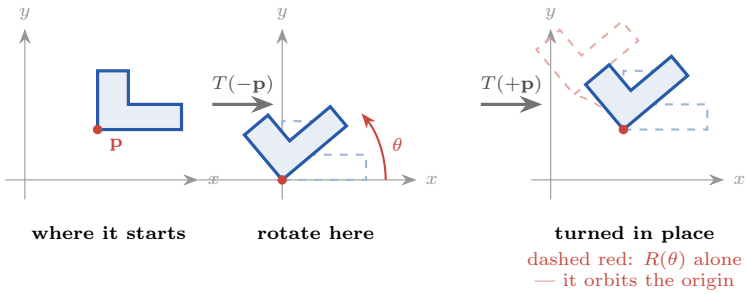

Figure 1: The pivot sandwich $T(\mathbf{p})\, R(\theta)\, T(-\mathbf{p})$.

T7 · `to_local` and `point_in_box` · 7 pts

A matrix `m` maps an object's **local** coordinates into **world** coordinates. `to_local(m, p)` goes the other way: it takes a world point and asks *where is that, as far as this object is concerned? (hint: one line, given T5).*

Now `point_in_box`: is a world point inside a box that has been rotated, scaled and moved? You *could* transform the four corners into world space and write a general point-in-quadrilateral test. **Don't.** Convert the *point* into the box's frame instead, and the test collapses to two comparisons, because in that frame the box is axis-aligned again:

```
abs(local.x) <= half_w    and    abs(local.y) <= half_h
```

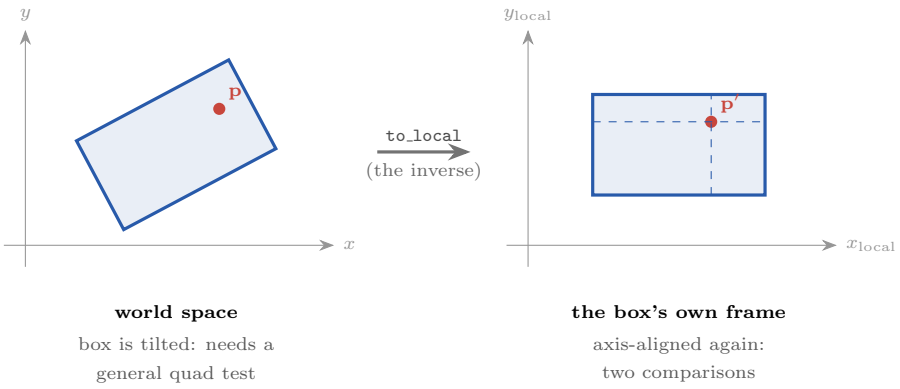

Figure 2: Do not move geometry, move the *question* into the frame

T8 · the scene graph · 7 pts

A `Node` is one part of your scene. It stores a **local** transform (where it sits *relative to its parent*), an optional `draw` callback, and a list of children. `__init__` and `add` are given. Write these two functions:

1. `world_matrix(node)` — returns where this node sits in **world** coordinates. Walk *up* the parent chain to the root and compose, with the root leftmost and the node itself rightmost:

```
compose(root, ..., parent, node.transform)
```

(The node's own transform is rightmost because it acts first.) A node with no parent returns its own transform.

2. `draw_tree(node, img, parent_world)` — draws the node and everything below it. For each node, in this order:

1. `world = parent_world @ node.transform` (`parent_world` is `identity()` at the root)
2. if the node has a `draw`, call `node.draw(img, world)`
3. recurse into each child, passing `world` as *their* `parent_world`

Required: `draw_tree` **must not call** `world_matrix`**.** Passing `world` down computes each product once, $O(n)$. Walking up from every node recomputes the same products over and over, $O(n \cdot \text{depth})$.

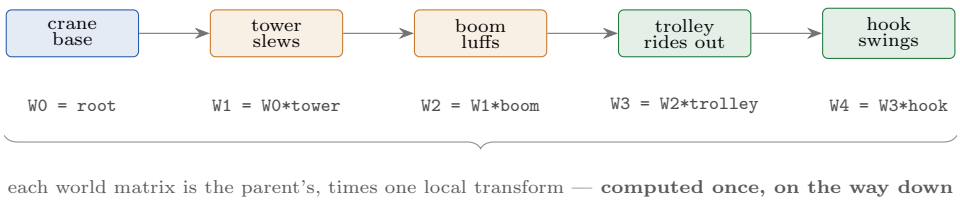

Figure 3: `draw_tree` accumulating world matrices down the exemplar's hierarchy. Move the tower and the boom, trolley and hook all follow — their world positions were never written down in the first place.

`mat3.py` **is done — T9–T12 below are written in** `shapes.py`**.** That file is where transforms meet geometry: it defines shapes in *local* coordinates, transforms them with your matrices, and hands the results to A2's `curves` and A1's `painter` to draw. The layering is now three deep, and each layer knows only the one below it.

T9 · `unit_square` and `regular_polygon` · 4 pts

1. `unit_square()` — returns the 1×1 square **centered at the origin**, vertices in exactly this order:

```
[(-0.5,-0.5), (0.5,-0.5), (0.5,0.5), (-0.5,0.5)]
```

(Centered, because rotation and scale act about the origin: a shape centered there spins and grows in place with no pivot sandwich.)

2. `regular_polygon(sides, radius)` — returns a list of `sides` vertices, vertex k at angle $2\pi k/$`sides` on a circle of `radius`. Use `curves.circle_point` to place each vertex. If `sides < 3`, return an empty list.

T10 · `draw_shape` and `fill_shape` · 6 pts

1. `draw_shape(img, m, shape, color, closed)` — transform every point of `shape` by `m`, then stroke the result with `curves.draw_polyline`. Two lines.

2. `fill_shape(img, m, shape, color)` — transform the points, then fill the polygon as a **triangle fan**: for vertices $p_0 \dots p_{n-1}$, fill the triangles $(p_0, p_1, p_2), (p_0, p_2, p_3), \dots$ with `painter.fill_triangle`. Fewer than 3 points fills nothing.

Required behaviour:

- Transform in float space; round coordinates **only** at the drawing call. (Rounding first makes shapes wobble as they spin — a pixel case catches this.)
- A fan is only correct for **convex** shapes; everything you are asked to fill here is convex. The given `star()` is deliberately concave: stroke it, don't fill it.

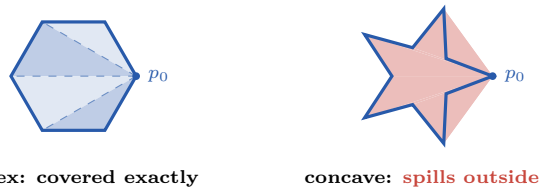

Figure 4: A fan from p_0 tiles a convex polygon exactly; across the concave `star()` it paints the red regions outside the outline.

T11 · `look_at` · 6 pts

`look_at(origin, target, scale_factor)` returns the local-to-world matrix for an object that **sits** at `origin` and **points its local** $+x$ **at** `target`, scaled by `scale_factor`.

Build it from columns, not from `atan2`. Compute the axes and write them straight into the matrix:

$$\hat{\mathbf{x}} = \text{normalize}(\text{target} - \text{origin}) \qquad \hat{\mathbf{y}} = \text{perp}(\hat{\mathbf{x}}) \qquad M = \begin{pmatrix} s\,\hat{x}_x & s\,\hat{y}_x & o_x \\ s\,\hat{x}_y & s\,\hat{y}_y & o_y \\ 0 & 0 & 1 \end{pmatrix}$$

Required behaviour: if `target` and `origin` are closer than `EPSILON`, there is no direction to face — fall back to $\hat{\mathbf{x}} = (1, 0)$, $\hat{\mathbf{y}} = (0, 1)$.

Draw a fish once in local coordinates, nose along $+x$, and this matrix lets it swim anywhere, facing wherever it is going.

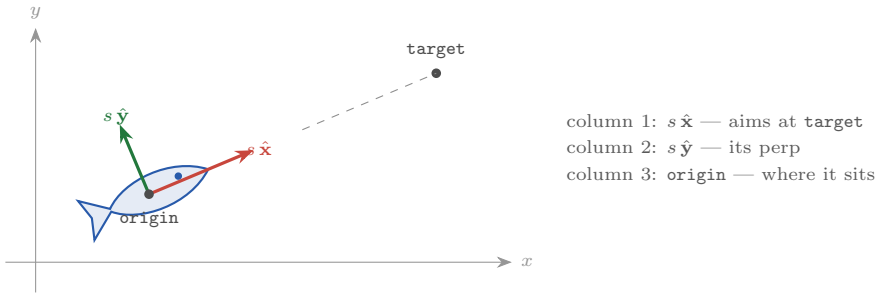

Figure 5: `look_at` read off geometrically: the three columns of M are the two scaled basis vectors and the position. The fish is drawn once in local coordinates, nose along local $+x$; the matrix places it at `origin`, facing `target`.

Notes (not graded). You have just built a small camera matrix — Lecture 13 builds the 3D version of exactly this. The same column-reading trick also tames the y-down convention: the **view matrix** V = `compose(translation(0, H), scaling(1, -1))` composed onto the outside of everything you draw makes the whole world y-up with the origin at the bottom-left, without touching one line of `mat3.py`. The lesson to keep: when an orientation looks wrong, never flip a sign inside `rotation` or `look_at` — change the frame, not the algebra. If you want y-up in your Part B scene, V goes in exactly one place: the top of your tree, as a camera node that everything else hangs from. Never fold it into individual parts.

T12 · `TransformStack` · 5 pts

A stack of matrices — the scene graph's information expressed as a traversal instead of a tree. Implement three methods:

1. `current()` — the transform in effect now, the top of the stack. One line.

2. `push(m)` — enter a new frame *relative to the current one*: the new top is `current @ m`, **not `m` by itself**. Push `translation(10, 0)` twice and you must end up 20 across, not 10.

3. `pop()` — remove and return the top, **but never empty the stack**: if only one matrix remains, leave it in place and return `identity()` instead.

Historical note: OpenGL shipped `glPushMatrix`/`glPopMatrix` on precisely this idea for two decades. Modern APIs make you manage matrices yourself — which is to say, they make you write what you are writing this week.

Part B — Your Articulated Scene (30 pts)

Using your toolkit, design and render an animated scene in `scene.py` whose moving parts are **articulated**: positioned relative to each other rather than relative to the screen. Render it as a sequence of frames and encode it to `scene.mp4`.

What you build is up to you, and we mean that. It does not have to be a machine, it does not have to be realistic, and it does not have to obey physics. One creature or a whole ecosystem; cute, eerie, funny, abstract or flatly impossible — all of it is welcome. The B1 checklist below is the only hard constraint.

Subject, style, palette, canvas shape, length and tone are yours, and B2 rewards ambition. If you have an idea that seems too strange for a graphics assignment, that is the one to make.

The rule that defines this assignment: nothing that moves in your scene should know its own screen position. A wheel knows where it sits on its car; the car knows where it sits in the world. If you are typing screen coordinates for a sub-part, stop — that part belongs in its parent's frame.

The test of whether you got it right is one line long. **Change the root node's transform and everything in your tree must move, turn or grow as one rigid object**, with no other edits. B1 requires exactly that, and the grader will try it. Two things this does *not* rule out:

- **A root that draws nothing.** The root can be an invisible stage, or a camera that pans, zooms and shakes, with any number of independent things hanging off it — a whole circus, not just one cart.
- **Static scenery outside the tree.** Your painted backdrop, and any prop that never moves, may stay in screen coordinates. Anything that *animates* belongs in the tree.

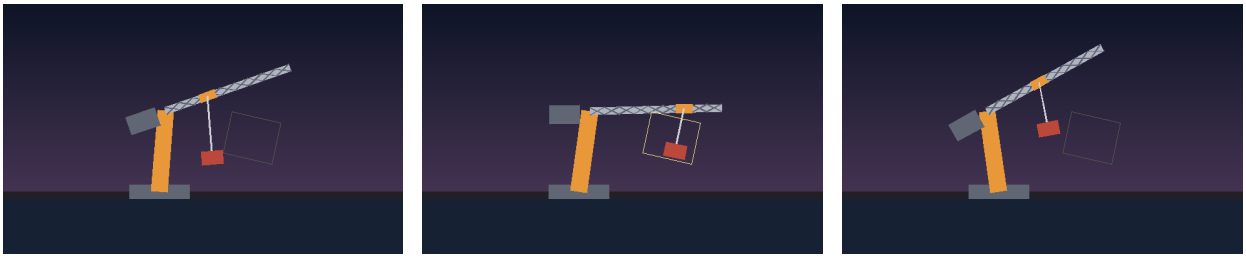

Figure 6: Three frames from the instructor exemplar, "The Harbour Crane" (frames 10, 40 and 98 of 120). Five levels: a base bolted to the quay, a tower that slews, a boom that luffs, a trolley that rides out along the boom, and a hook that hangs from the trolley — swinging with a deliberate lag, and counter-rotating so the load hangs true no matter what its ancestors are doing. The tilted gate lights up only while the load is genuinely inside it, tested with `point_in_box` in the gate's own frame. Your scene should be *yours*, not this one — and it need not be a machine at all.

Where you could take it

The crane is deliberately conventional. A scene graph can do far stranger things, and each row below leans on something the hierarchy is uniquely good at. **None of it is required and none of it is a template** — it is here to show how big the space is. Borrow a technique, not a subject.

- **Recursion.** A function that adds a branch node and then calls itself twice grows a whole tree: eight levels deep is 255 branches from about ten lines. Sway each level slightly out of phase and the tree breathes. Coral, lightning, antlers, a fractal creature.
- **Long chains.** Thirty segments, each bending a little behind its parent's phase: a tentacle, a snake, a dragon's tail, a whip, a strand of seaweed. The lag travels down the chain by itself.
- **Instancing.** Write one function that builds a subtree, then call it under six rotated parents: a kaleidoscope, a mandala, a flower, a snowflake, a Ferris wheel.
- **A camera as root.** Everything below a camera node inherits the cinematography: a slow push-in, a whip pan, a shake on impact, a zoom from a galaxy down to one moon.
- **`look_at`.** Eyes that follow a fly. A sunflower tracking the sun. A crowd of heads all turning to watch the same thing go past.

- **Re-parenting.** A juggler throws a ball to a second juggler; a bird lands on a moving train. Moving a node to a new parent without a visible jump means giving it the local transform `P.inverse() @ W`, where `P` is the new parent's world matrix and `W` is the node's old one — the inverse item, earned honestly.
- **Squash and stretch.** Non-uniform scale and shear: a blob that bounces, a jelly that wobbles, a building that sways as if it is dancing.

Write whatever helpers you like in `scene.py` — your own shapes, Bézier outlines from A2's `curves`, particles, a palette that shifts with `u`. Two practical limits: everything has to live in `scene.py` (it is the only scene file you submit), and it stays standard-library Python. `fill_shape` fills convex shapes only, so fill anything concave as several convex pieces, or stroke it.

Workflow — follow this order

1. `python3 encode.py --check` — confirm ffmpeg still works.
2. Set `FRAMES = 1` and get a **single frame** right. Iterating on one frame is seconds; on 120 it is minutes.
3. **Turn on** `DEBUG_AXES` and look at your frames before you trust them.
4. Raise `FRAMES`, render, encode, watch it. Then adjust.

```
python3 scene.py       # writes frames/frame_0000.ppm, frame_0001.ppm, ...
python3 encode.py      # combines them into scene.mp4
```

Use the debug axes. `shapes.draw_axes` draws a frame's local $+x$ in red and $+y$ in green, and the starter wires it to a `DEBUG_AXES` flag that draws them for every node. A transform bug is invisible in the final picture and obvious the moment you can see the frames: you can tell at a glance whether a part is rotated the way you meant, scaled the way you meant, and hinged where you meant.

If your render is slow

It should not be. The starter's 90 frames render in under a second, and even a 500-node recursive tree takes only a couple of seconds, so do not hold back on complexity for speed's sake. If yours takes a minute, you are almost certainly **repainting your background every frame**.

Remember what `painter.set_pixel` really is: one pixel, one Python function call. A 640×400 gradient is 256,000 of them, and doing that once per frame across 90 frames is 23 million calls spent redrawing an image that is *identical every time*.

The starter shows the fix. Anything that does not depend on `u` is painted once into a module-level `BACKGROUND`, and each frame starts from a copy:

```
BACKGROUND = make_background()      # painted ONCE, at import

def render_frame(i):
    img = BACKGROUND.copy()         # not a repaint - a copy
    ...
```

Note the `.copy()`. Draw onto `BACKGROUND` directly and every frame accumulates the previous one on top of it.

The general rule is worth more than the trick: work that does not depend on `u` does not belong inside `render_frame`. Put your starfield, skyline or texture in `make_background()`. That single habit is the difference between iterating on your animation and waiting for it.

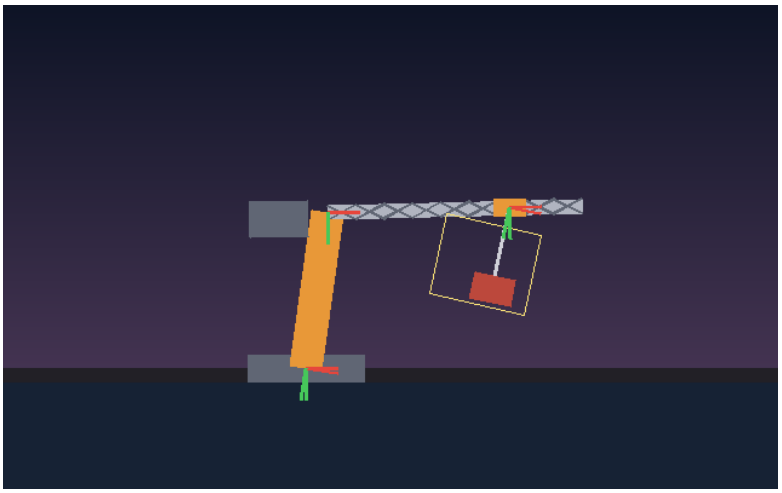

Figure 7: The exemplar with `DEBUG_AXES = True`. Every joint's local frame is visible — base, tower, boom pivot, trolley, hook. Note that a node carrying a large scale gets long axes, because the axes are drawn *in* that frame: the debugger is telling you the truth about what your children will inherit. Turn this off before submitting.

B1 · Compliance Checklist · 16 pts

Binary items, checked by the grader exactly as written. The checklist also appears as comments in the starter `scene.py`.

Item	Pts
`python3 scene.py` runs from a clean folder and writes `frames/frame_0000.ppm` …	2
At least 60 frames; canvas at least 480 × 320	1
`python3 encode.py` produces a playable `scene.mp4` from your frames	2
A scene graph at least **3 levels deep** (root → child → grandchild)	3
Drawn via `mat3.draw_tree` or `shapes.TransformStack` — not by hand	2
A grandchild has its **own animated** local transform, so it inherits its parent's motion *and* adds its own	2
Changing **only** the root transform moves the whole scene as one	2
One use of `to_local` / `point_in_box` / `inverse` that **affects what is drawn**, marked with a comment	1
Any randomness is seeded — the same movie on every run	1
Total	**16**

Two of these are faked every year, so to be explicit. The **grandchild** item means a part whose motion is genuinely *compounded*: if its parent stopped moving, the grandchild would still move, and while the parent moves, the grandchild does both at once. A grandchild whose local transform is a constant does not count. And the **inverse** item, like A2's dot-product item, must change something visible — computing a value and ignoring it scores 0.

The minimums are only minimums. A longer film, a bigger or oddly shaped canvas (square, tall, letterboxed) and a tree fifty levels deep are all fine; the only ceiling is the upload limit.

B2 · Craft Rubric · 14 pts

Criterion	0 — absent	1 — attempted	2 — competent	3 — excellent
Hierarchy design (worth 0/1/3/5)	Flat; parts positioned in screen coordinates	A tree exists, but pivots are arbitrary and some parts are placed by hand	Sensible depth; each part's local origin sits at its actual joint	Hierarchy mirrors how the thing moves, real or imagined; adding a part would be trivial
Motion quality	Parts teleport or jitter	Movement happens but is uniformly linear and abrupt	Smooth, continuous motion; easing or varied pacing used deliberately	Motion has character — lag, follow-through, secondary movement in the leaves, or a stylised rhythm that is plainly deliberate
Composition & design	Elements scattered; no framing	Some structure; foreground and background undifferentiated	Clear staging; the articulation is easy to read	Deliberate composition that holds up *throughout* the motion, not just on frame 0
Originality & ambition	Reproduces the starter cart or the handout crane	Derivative with small changes	Clearly original; minor artifacts (popping, gaps, stray pixels)	Original and polished — or a genuinely bold idea whose rough edges are the price of its reach; the loop or ending is considered

Hierarchy design counts extra (it is what this assignment is about): its four columns are worth 0, 1, 3 and 5 points. The other three criteria score as labelled, 0–3. Total 14.

Calibration: *competent* on every axis earns 9/14 — a solid outcome. Top marks are reserved for work that would headline the class gallery. Submitting the starter's cart, lightly recolored, caps *Originality* at 1.

Ambition is rewarded, not punished. Physics is optional and realism is optional; only intent is required. If you attempt something genuinely strange or hard and a few seams show, the marker weighs the reach, not only the finish: a safe idea polished to a shine and a wild idea with a few rough edges can both earn a 3 on *Originality & ambition*.

Two cheap upgrades, both nearly free. First, **put each part's local origin at its joint** — a mast hinged at its base needs only `rotation()`, while a mast whose origin is at its middle needs a pivot sandwich and will look wrong the first three times. Second, **give the leaves a lag**: offset a child's phase slightly behind its parent's ($\sin(2\pi u - 0.9)$ instead of $\sin(2\pi u)$) and the whole assembly stops looking welded together.

Submission

Submit **one zip file** named `RollNo_A3.zip` (e.g. `231234567_A3.zip`) on the course website, containing exactly:

`mat3.py` `shapes.py` `scene.py` `scene.mp4`

Do **not** include any of the following: the `frames/` folder (it is hundreds of frames), `vec2.py`, `curves.py`, `canvas.py`, `painter.py`, `encode.py`, `check.py`, `expected_a3.py`, `__pycache__`, or the ffmpeg binary. We

run your submission against our own copies of the given files. All three `.py` files must have the header block (name, roll number, AI disclosure) filled in.

If `scene.mp4` exceeds the upload limit, reduce your canvas size or frame count and re-encode — do not submit a partial video.

The deadline is hard: no late submissions, per the course outline. A missed deadline scores zero. Documented medical or university grounds are the only exception.

Appendix A: The Transform Cheat Sheet

Every matrix in this assignment, and what its columns are telling you.

You want	Matrix	Read it as
move by (t_x, t_y)	$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$	basis unchanged; origin goes to (t_x, t_y)
scale about origin	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$(1,0) \mapsto (s_x, 0)$, $(0,1) \mapsto (0, s_y)$
rotate about origin	$\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}$, $c = \cos\theta$	$(1,0) \mapsto (\cos\theta, \sin\theta)$ — that *is* column one
rotate about $\mathbf{p}$	$T(\mathbf{p})\, R(\theta)\, T(-\mathbf{p})$	go there, turn, come back
undo a transform	`m.inverse()`	world → local; the frame-changing tool
face a target	`look_at(o, t, s)`	columns are the axes you chose
flip to y-up	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & H \\ 0 & 0 & 1 \end{pmatrix}$	$(0,1) \mapsto (0,-1)$; origin to bottom-left — the *view matrix*

The single rule to carry away: in $ABC\mathbf{p}$, matrix C acts first. Read products right to left and they read as a sequence of events.

Appendix B: Debugging Transforms

Transform bugs are invisible in the output and obvious in the frame. When something is wrong, work down this list.

Symptom	Almost always
Object *orbits* when it should spin	product order reversed, or a missing pivot sandwich (T6)
Everything piles up at the origin	`push` storing `m` instead of `current @ m`, or children ignoring their parent
Shape wobbles or shimmers as it turns	rounding before transforming, not after (T10)
A direction drifts as the object moves	`apply_point` used where `apply_vector` was meant (T3)
Child moves but ignores the parent	`draw_tree` not passing `world` down
Scene explodes to nothing, or vanishes	singular matrix — a scale of 0 somewhere (T5)
Everything is mirrored	`perp` sign, or forgetting y points *down*
Render takes minutes, not seconds	the background is being repainted inside `render_frame` — cache it (see Part B)

Two tools make all of these quick. `m.format_matrix()` prints a matrix over three lines — the last column is the translation and the top-left 2×2 is the rotation/scale, and reading those two facts off the numbers finds most bugs faster than staring at the picture. `shapes.draw_axes(img, m)` draws the frame itself.

Appendix C: A Scene Graph in Twenty Lines

The skeleton every Part B scene shares. Everything interesting is in the local transforms; the traversal never changes.

```
def build_scene(u):                 # u runs 0.0 -> 1.0
    # ROOT - the only node that knows a screen position
    root = mat3.Node(Mat3.translation(220.0, 300.0), name="cart")
    root.draw = lambda img, w: draw_body(img, w)

    # LEVEL 2 - placed in the ROOT's frame, not the screen's
    wheel = root.add(mat3.Node(
        mat3.compose(Mat3.translation(-38.0, 24.0),
                     Mat3.rotation(spin_angle(u))),
        draw=lambda img, w: draw_wheel(img, w)))

    # LEVEL 2 - the mast, hinged at its base on the cart
    mast = root.add(mat3.Node(
        mat3.compose(Mat3.translation(0.0, -17.0),
                     Mat3.rotation(lean_angle(u))),
        draw=lambda img, w: draw_mast(img, w)))
    # LEVEL 3 - inherits the cart's motion AND the mast's, adds its own
    mast.add(mat3.Node(
        mat3.compose(Mat3.translation(19.0, -80.0),
                     Mat3.rotation(flutter(u))),
        draw=lambda img, w: draw_flag(img, w)))
    return root

def render_frame(i):
    u = i / max(FRAMES - 1, 1)
    img = BACKGROUND.copy()                 # painted once, copied per frame
    mat3.draw_tree(build_scene(u), img)     # one call draws everything
    return img
```

Rebuilding the tree every frame is the simplest thing that works, and it keeps all the time-dependence in one place: each node's local transform is a function of `u`, and the hierarchy does the rest.

Note that `draw_wheel` and `draw_flag` never learn where they ended up. They are handed a world matrix and draw in local coordinates. That is the property worth protecting — the moment a draw function starts reasoning about screen positions, you have lost the thing this assignment was teaching.