Pre-class meet n' greet topics for your table:
What topic do you wish to do your project on?

Lecture 7:

Neurosymbolic Methods for Visual Content Generation

Visual Computing Systems
Stanford CS348K, Spring 2025

Key theme in the class:
picking the right representation for the job

Two examples:

- Frankencamera: chooses a timeline representation for telling an abstract camera machine to acquire shots

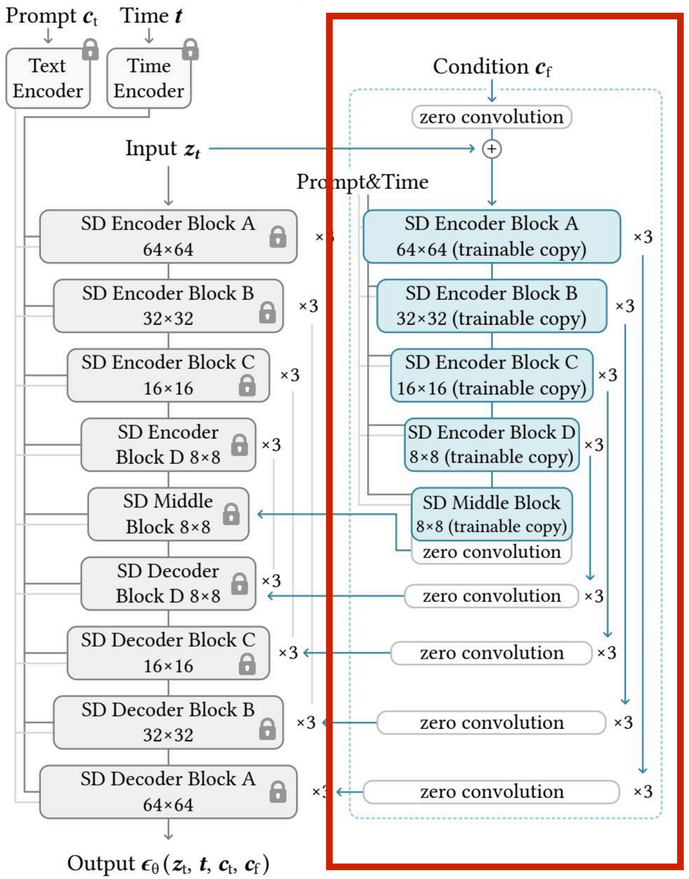
- Halide: provides set of primitives for scheduling loop nests (for computations on dense arrays)

What are signs that a system designer has wisely selected good representations for the task at hand?

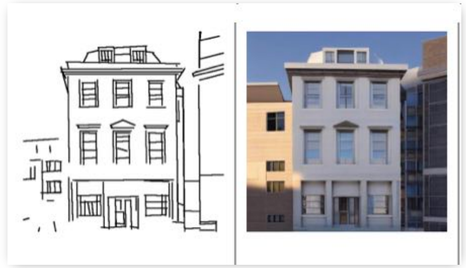
Learned representations

Examples from last time: exploit structure of learned representations to enhance control over generate AI output

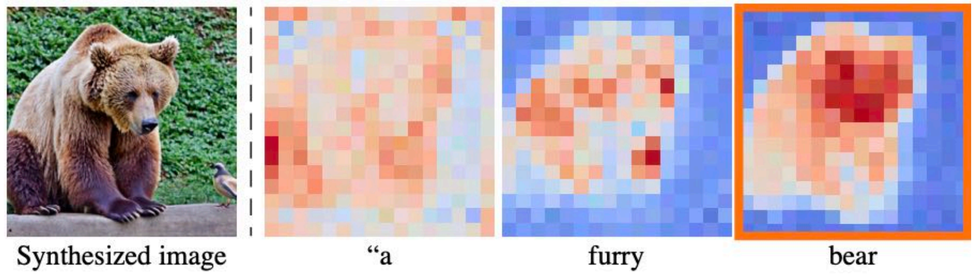
Example 1: ControlNet… add extra DNN layers "on the side" to capture intended control behavior

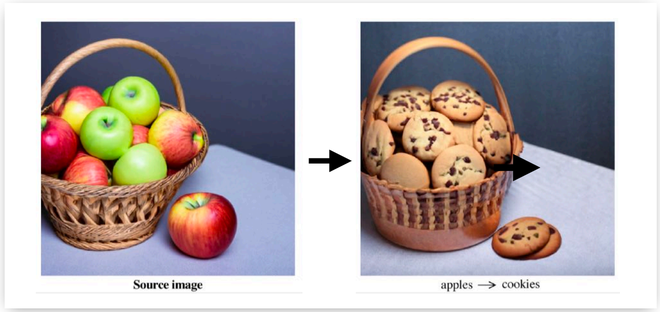


(a) Stable Diffusion (b) ControlNet

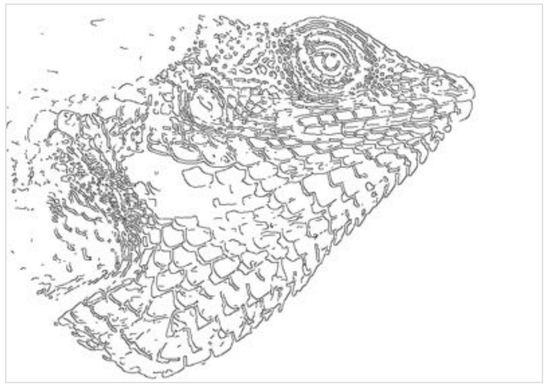


Example 2: prompt2prompt editing. Hold transformer's attention layer constant to maintain image coherence on edits.



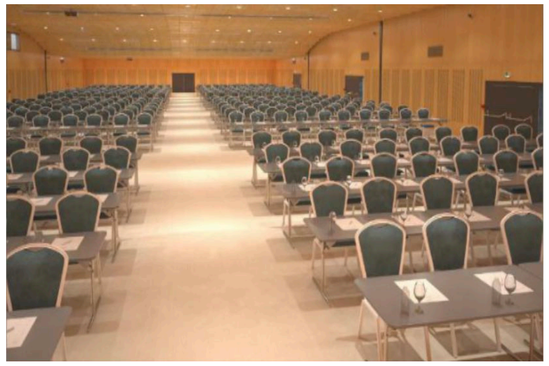


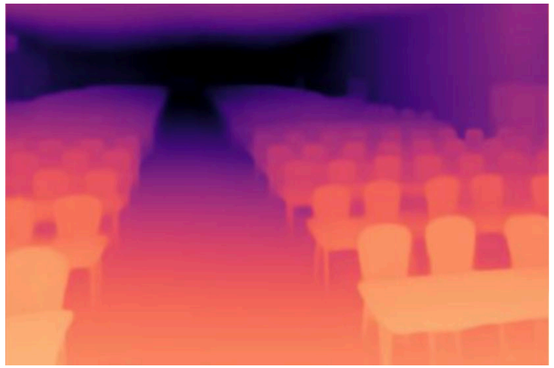
Common case: implicitly “teach” a model your control intent via providing paired data examples. (Let model learn largely uninterpretable weights that do the job)



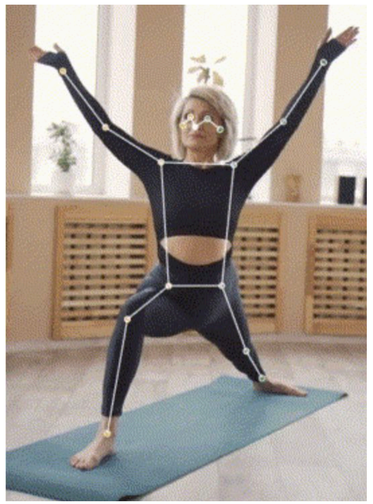
Edge detection

Segmentation





Depth



Another example:
Pose Estimation

Neurosymbolic methods: combining traditional symbolic representations with learned representations

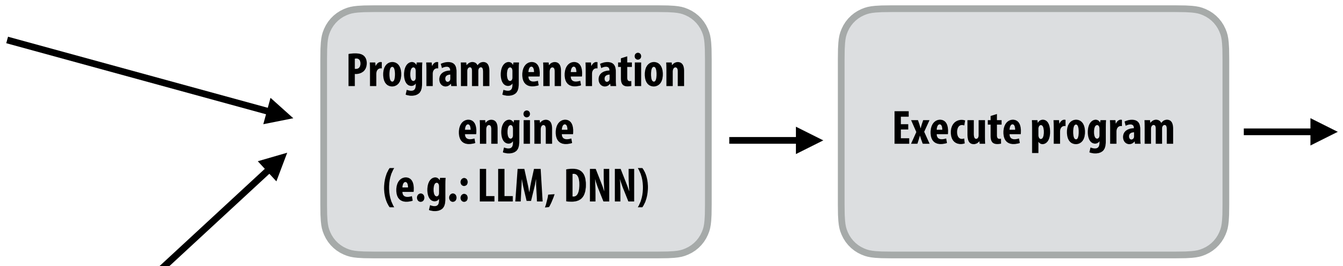
An increasingly common paradigm for generative AI

Reducing generation tasks to the act of writing programs

High-level specification

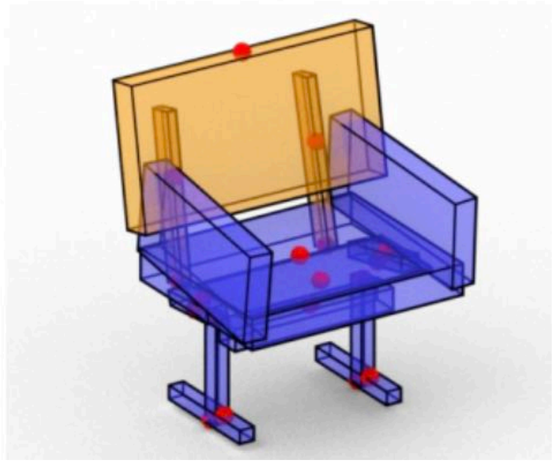
"Chair with a flat back"

Domain-specific language providing useful primitives (often with precise semantics)




```
def Chair():
  bbox = Cuboid(1.2, 1.4, 1, T)
  base = Base(.9, .5, .8, T)
  seat = Seat(1.1, .1, .9, T)
  back = Back(1.1, .9, .2, F)
  arm = Cuboid(.1, .4, .7, F)
  attach(base, bbox, .5, 0, .5, .5, 0, .5)
  squeeze(back, bbox, base, top, .5, .1)
  attach(seat, base, .5, 0, .5, .5, 1, .5)
  attach(arm, back, .5, .5, 0, .1, .3, .5)
  attach(arm, seat, .5, 0, .5, .1, .7, .5)
  reflect(arm, X)
...
def Back(l, w, h, aligned):
  bbox = Cuboid(l, w, h, aligned)
  surface = Cuboid(1.16, .64, .13, T)
  slat = Cuboid(.04, .76, .1, F)
  attach(surface, bbox, .5, 1, .5, .5, 1, .7)
  attach(slat, bbox, .5, 0, .5, .2, 0, .45)
  attach(slat, surface, .5, .6, .8, .2, .3, .2)
  reflect(slat, X)
```

Generated output



One benefit: a program is a human interpretable representation

We read and edit programs all the time!

```
def Chair():
  bbox = Cuboid(.82, 1.6, .85, T)
  base = Base(.75, .66, .66, T)
  seat = Seat(.8, .13, .85, T)
  back = Back(.8, .9, .1, T)
  attach(base, bbox, .5, 0, .5, .5, 0, .5)
  attach(back, bbox, .5, 1, .5, .5, 1, .05)
  attach(seat, base, .5, .0, .5, .5, 1, .5)
  attach(back, seat, .5, .0, .5, .5, .75, .05)

...

def Back(l, w, h, aligned):
  bbox = Cuboid(l, w, h, aligned)
  surface = Cuboid(.8, .4, .1, T)
  slat = Cuboid(.05, .5, .05, T)
  attach(surface, bbox, .5, 1, .5, .5, 1, .5)
  squeeze(slat, bbox, surface, bot, .1, .5)
  translate(slat, X, 3, 0.8)
```

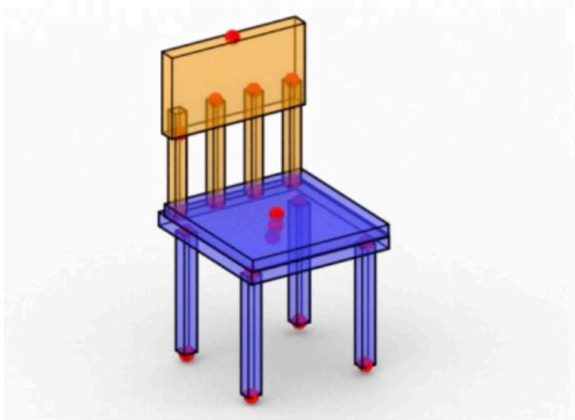
EDIT

```
def Chair():
  bbox = Cuboid(0.5, 2, 0.7, T)
  base = Base(0.5, .95, 0.7, T)
  seat = Seat(0.5, .05, 0.7, T)
  back = Back(0.5, 1, 0.05, T)
  attach(base, bbox, .5, 0, .5, .5, 0, .5)
  attach(back, bbox, .5, 1, .5, .5, 1, .05)
  attach(seat, base, .5, .0, .5, .5, 1, .5)
  attach(Back, seat, .5, .0, .5, .5, .75, .05)

...

def Back(l, w, h, aligned):
  bbox = Cuboid(l, w, h, aligned)
  surface = Cuboid(0.5, 0.1, 0.05, T)
  slat = Cuboid(.09, .9, .05, T)
  attach(surface, bbox, .5, 1, .5, .5, 1, .5)
  squeeze(slat, bbox, surface, bot, .1, .5)
  translate(slat, X, 2, 0.8)
```

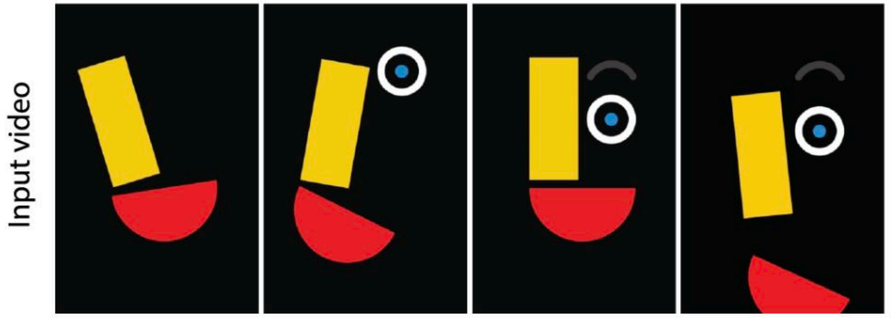
execute



execute

Editing motion graphics (vector graphics animations)

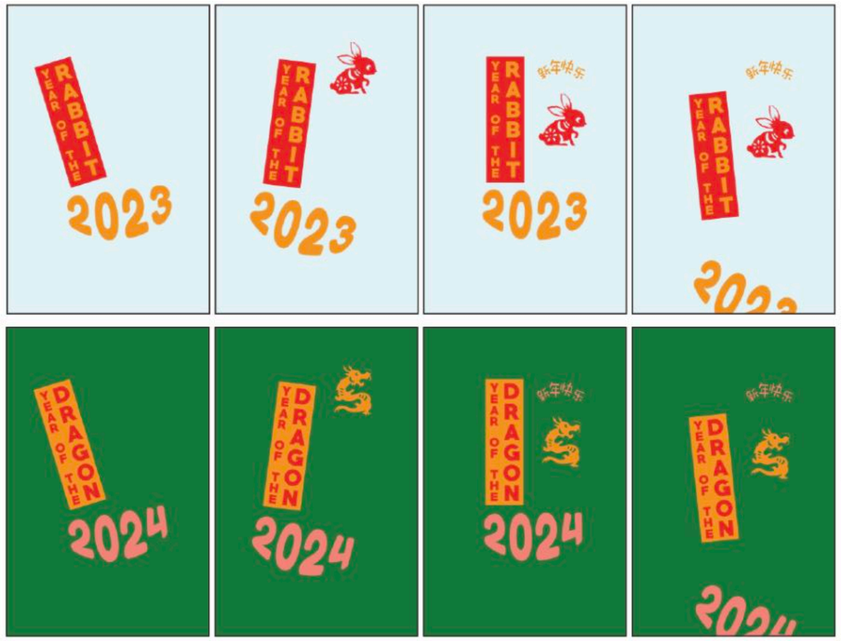
1. Video —> SVG keyframe animation (computer vision)
2. LLM edits the SVH file (e.g., change yellow rectangle to texture map)
3. Re-render animation



Motion vectorization

```
<svg w=200, h=200, dur="5s">
  <g id="0">
    <img href="red_ball.png" />,
    <scale val="1.0 1.0; 1.05 1.0; ..." />,
    <translate val="0.1 0.5; 0.15 0.5..." />,
    <rotate val="1.0; 1.05; ..." />,
    <z-index val="0; 0; ..." />,
    ...
  </g>
  <g id="1">...</g>
  ...
</svg>
```

SVG



SVG

SVG

Program transformer

```
NewYearsProgTransformer(P, args, [frmA, frmB]):
  // Set background color.
  setAppearance("bg", args.background)

  // OBJECT SELECTOR: Query for the red semicircle.
  selObjs = objSelector(P, propQuery, "color", "red", [frmA, frmB])
  // OBJECT TRANSFORMER: Change the appearance to the year.
  changeAppearanceObjTransformer(selObjs, args.year, [frmA, frmB])
  // Repeat obj selection and obj transformation for banner and animal.
  selObjs = objSelector(P, propQuery, "color", "yellow", [frmA, frmB])
  changeAppearanceObjTransformer(selObjs, args.banner, [frmA, frmB])
  selObjs = objSelector(P, propQuery, "color", "white", [frmA, frmB])
  changeAppearanceObjTransformer(selObjs, args.zodiac, [frmA, frmB])

  // OBJECT SELECTOR: Query for the gray curve.
  selObjs = objSelector(P, propQuery, "color", "gray", [frmA, frmB])
  changeAppearanceObjTransformer(selObjs, args.characters, [frmA, frmB])
  // OBJECT TRANSFORMER: Apply an oscillating scale.
  function pulse(t, [sx, sy]):
    return [sx + 0.5 * np.sin(t / 10), sy + 0.5 * np.sin(t / 10)]
  motionTexObjTransformer(selObjs, pulse, args.pulseArgs, [frmA,frmB])

  // OBJECT SELECTOR: Query for the blue circle.
  selObjs = objSelector(P, propQuery, "color", "blue", [frmA, frmB])
  // OBJECT TRANSFORMER: Remove the object.
  removeObj(selObjs, [frmA, frmB])
```

Another example

Python programs as "plans" to carry out tasks in 3D environments

Notice how the program queries for state of environment and exhibits simple conditional execution.

PROMPT for Planning

Import action primitives

```
from actions import walk <obj>,
grab <obj>, switchon <obj>,
open <obj>, standup, find
<obj>, putin <obj> <obj>, ...
```

Available objects list

```
objects=[clothesshirt' sink,
pie, apple, fridge, garbagecan,
tv, dishwashingliquid, bed,
bookshelf, salmon, stove, plate,
coffeepot, kitchentable,
wineglass, paper, microwave,
toothbrush, toothpaste,
bathroomcabinet, kitchen, lime,
painkillers, barsoap, ...]
```

Example task(s)

```
def throw_away_lime():
  # 0: find lime
  find('lime')
  ...
  # 5: close garbagecan
  assert('close' to 'garbagecan')
     else: find('garbagecan')
  assert('garbagecan' is 'opened')
     else: open('garbagecan')
  close('garbagecan')
  ...
  # 6: Done
```

Next task prompt

```
def microwave_salmon():
```

LLM code generator

Generated Plan

```
def microwave_salmon():
 # 0: walk to kitchen
 walk('kitchen')
 # 1: find microwave
 find('microwave')
 # 2: open microwave
 assert('close' to 'microwave'
    else: find('microwave')
 assert('microwave' is 'closed')
    else: close('microwave')
 open('microwave')
 ...
 # 5: put salmon in microwave
 assert('salmon' in 'hands')
    else: find('salmon')
    else: grab('salmon')
 assert('close' to 'microwave')
    else: find('microwave')
 assert('microwave' is 'opened')
    else: open('microwave')
 putin('salmon', 'microwave')
 ...
 close('microwave')
 ...
 switchon('microwave')
 # 8: wait for salmon to be done
       cooking 1-2 minutes
 # 9: turn off microwave
 ...
 switchoff('microwave')
 ...
 open('microwave')
 ...
 grab('salmon')
 # 12: put salmon on plate
 assert('salmon' in 'hands' )
 ...
 assert('close' to 'plate' )
    else: find('plate')
 putin('salmon', 'plate')
 ...
 close('microwave')
 # 14: Done
```

Comment: Summary

Action API + object(s): Plan Step

Comment - Action(s) Mapping

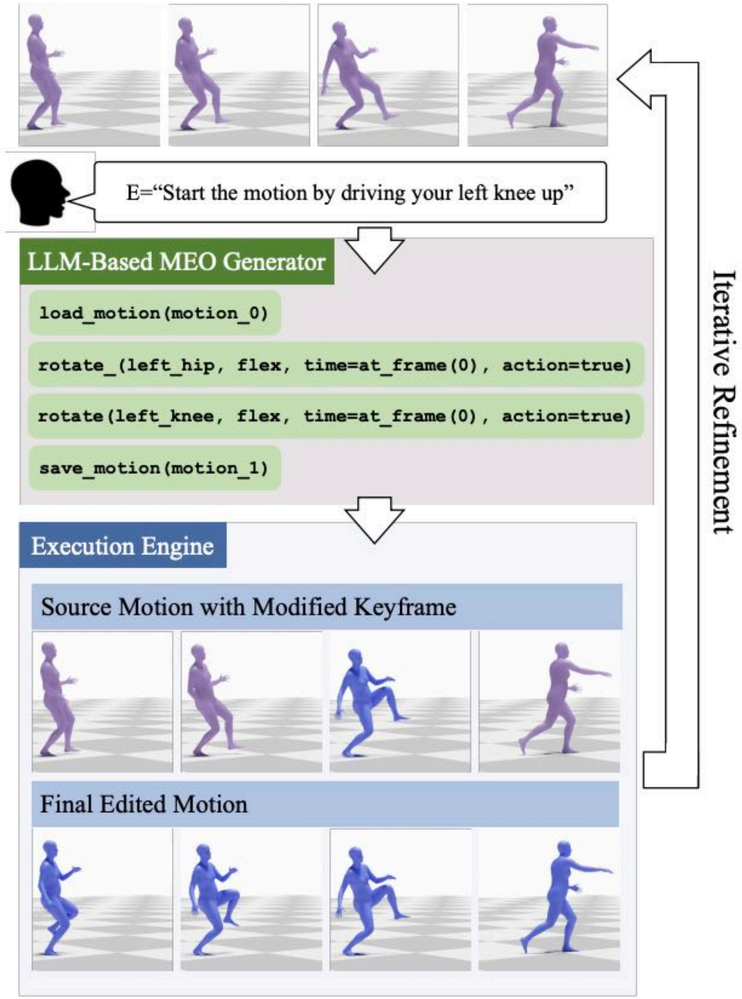
Assertions: State Feedback

Else: Recovery Actions

Optional Steps

Generating animations

High-level motion editing commands —> programs that perform keyframe edits —> use AI model (diffusion) to interpolate the keyframes



API for MEO Construction

```
from actions import translate_joint, rotate_joint,...
from timing import when_joint, at_frame,...
from motion_io import load_motion, save_motion
```

Available Parameters

```
relative_moments=["highest", "lowest",...]
translate_directions=["forward", "backward", "up",...]
rotation_directions=["abduct", "adduct", "extend",...]
joints_that_rotate=["right_knee", "left_knee",…]
```

In-context Learning Example(s)

```
# the person is jumping. Bring the right knee to chest
# as you jump
def right_knee_to_chest():
  # load the motion that needs to be edited
  load_motion("motion_0")
  ...
  # bend the right knee
  rotate_joint("right_knee", "flex",
    time=when_joint("waist", "highest"))
  # flex the right hip to bring the knee higher
  rotate_joint("right_knee", "flex",
    time=when_joint("waist", "highest"))
  ...
  # save the edited motion
  save_motion("motion_1")
```

Editing Instruction *E*

```
# A person is doing a side kick with the right leg.
# Can you get that kick higher out?
```

LLM completes code here

Another example

Programs as "plans" to answer questions

Q: Is the carriage to the right of a horse?

Large Language Model

```
answer = "no"  # default answer
horse_exists = query("Is there a horse?")
if horse_exists == "yes":
  x1, y1 = get_pos("carriage")
  x2, y2 = get_pos("horse")
  if x1 > x2:
    answer = "yes"
```

Object Localizer

Simple VQA Method

Note how the DSL contains primitives that themselves might be implemented as "Black-box" DNNs

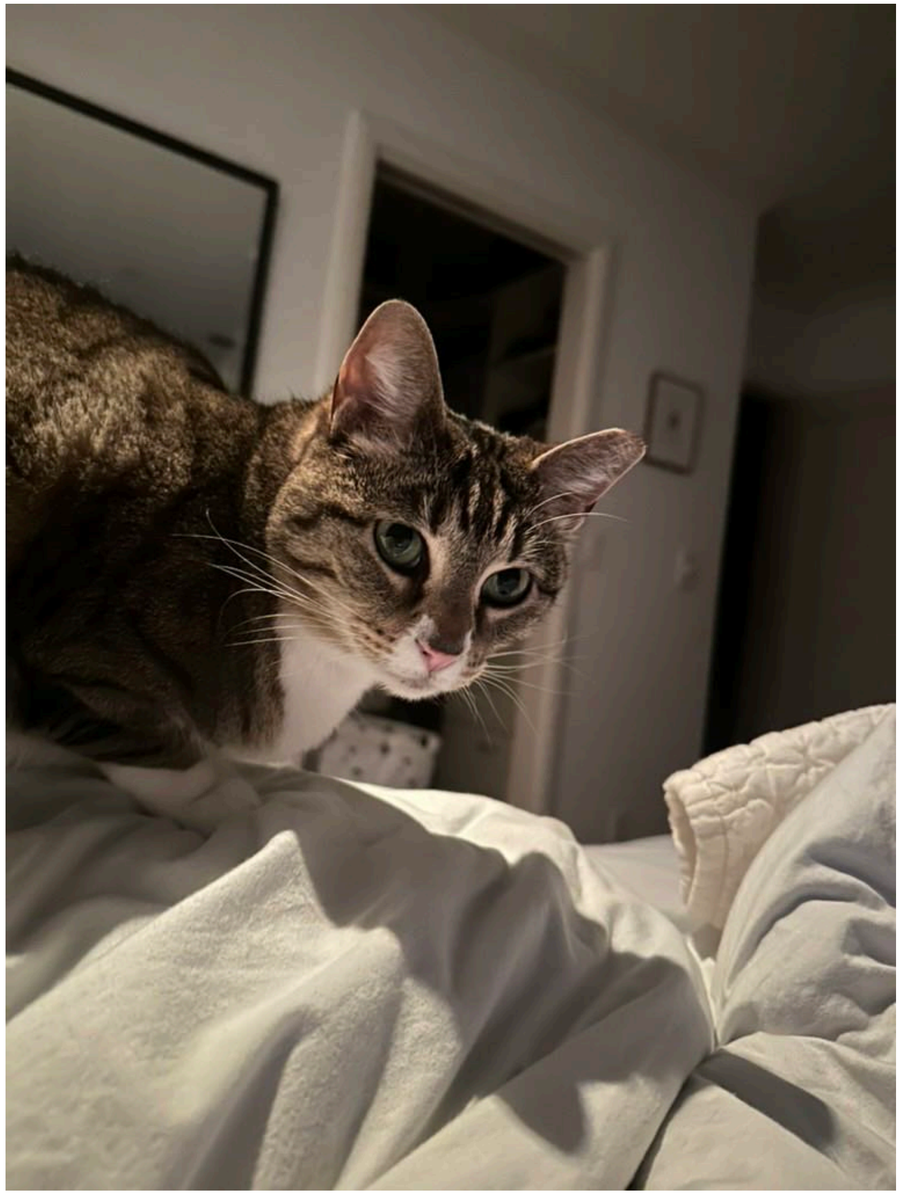
What is the DSL for the following edit?

- Given this photo, make the cat's tail curve back into the frame.

- Let's think about the pros and cons of this program-driven generation/editing approach:

  What are the benefits?

  What are limitations of this approach?



Challenges

- Designing a DSL can be challenging
  - What primitives to include? How to implement these primitives?
  - Problems must down into clearly defined, self-contained steps

- How do we know when a learned program generator produces a valid program (a program that performs the task specified in the controls)?
  - Can we predict when a program generator will fail?

MoVER: tonights reading (SIGGRAPH 2025)

Text prompt

<svg> ... </svg>
Move the orange circle above the rectangular shape.
In the meantime, rotate the letter H clockwise by 90 degrees.

MoVer program

```
o1 = to.clr(o,"orange")∧shp(o,"circle") # check for an orange circle
o2 = to.shp(o,"rectangle")              # check for a rectangle
o3 = to.id(o,"H")                       # check for the id "H"
# check for a translation by the circle to the top of the rectangle
m1 = tm.type(m,"trn")∧agt(m,o1)∧post(m,top(o1,o2))
# check for a clockwise rotation of 90 degrees by "H"
m2 = tm.type(m,"rot")∧agt(m,o3)∧dir(m,"cw")∧mag(m,90)
# assert that the translation and the rotation overlap in time
while(m1,m2)
```

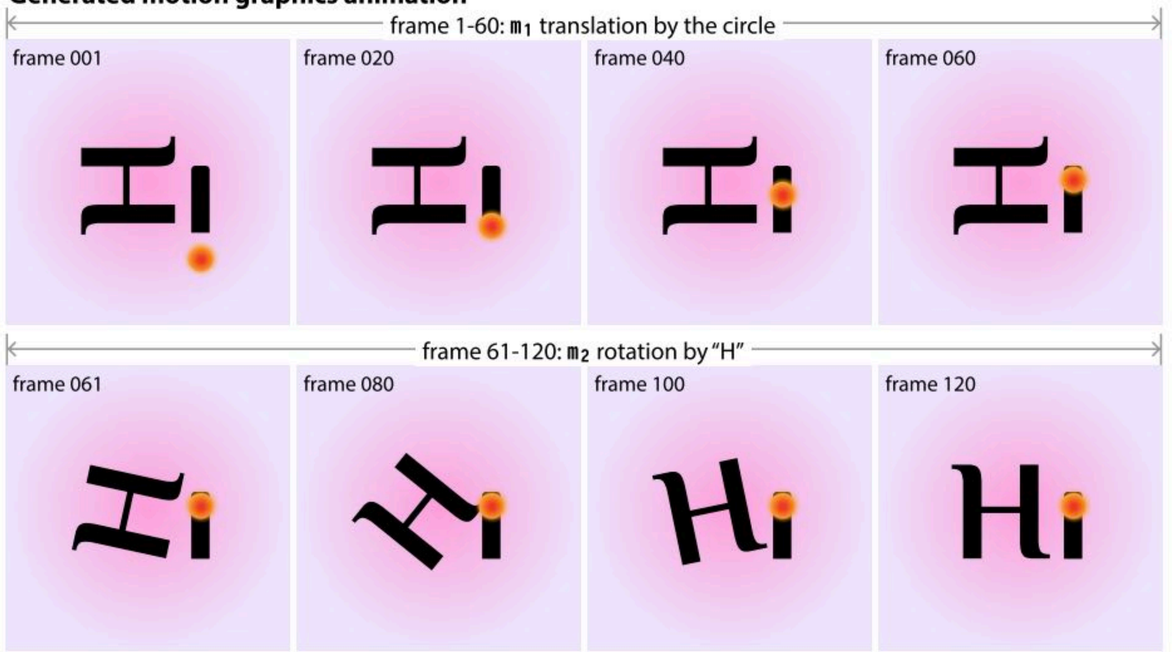
MoVer verification report

```
m1 = tm.type(m,"trn")∧agt(m,o1)∧post(m,top(o1,o2))          false
     top(o1,o2)                                             false
     post(m,top(o1,o2))                                     false

m2 = tm.type(m,"rot")∧agt(m,o3)∧dir(m,"cw")∧mag(m,90)       true

while(m1,m2)                                                false
```

Generated motion graphics animation



- Follows an emerging pattern in AI-based program generation
- Given an editing instruction, generate:
  - A program that performs the edit
  - A set of predicates that should be true if the edit was successfully performed (verifiers)
- If any of the predicates fail, have the program generator try again (given information about its prior failures)
- Questions:
  - What is the collection of verifiers?
  - Can the verifiers be "powerful enough" to provide useful checking?

Project Discussions