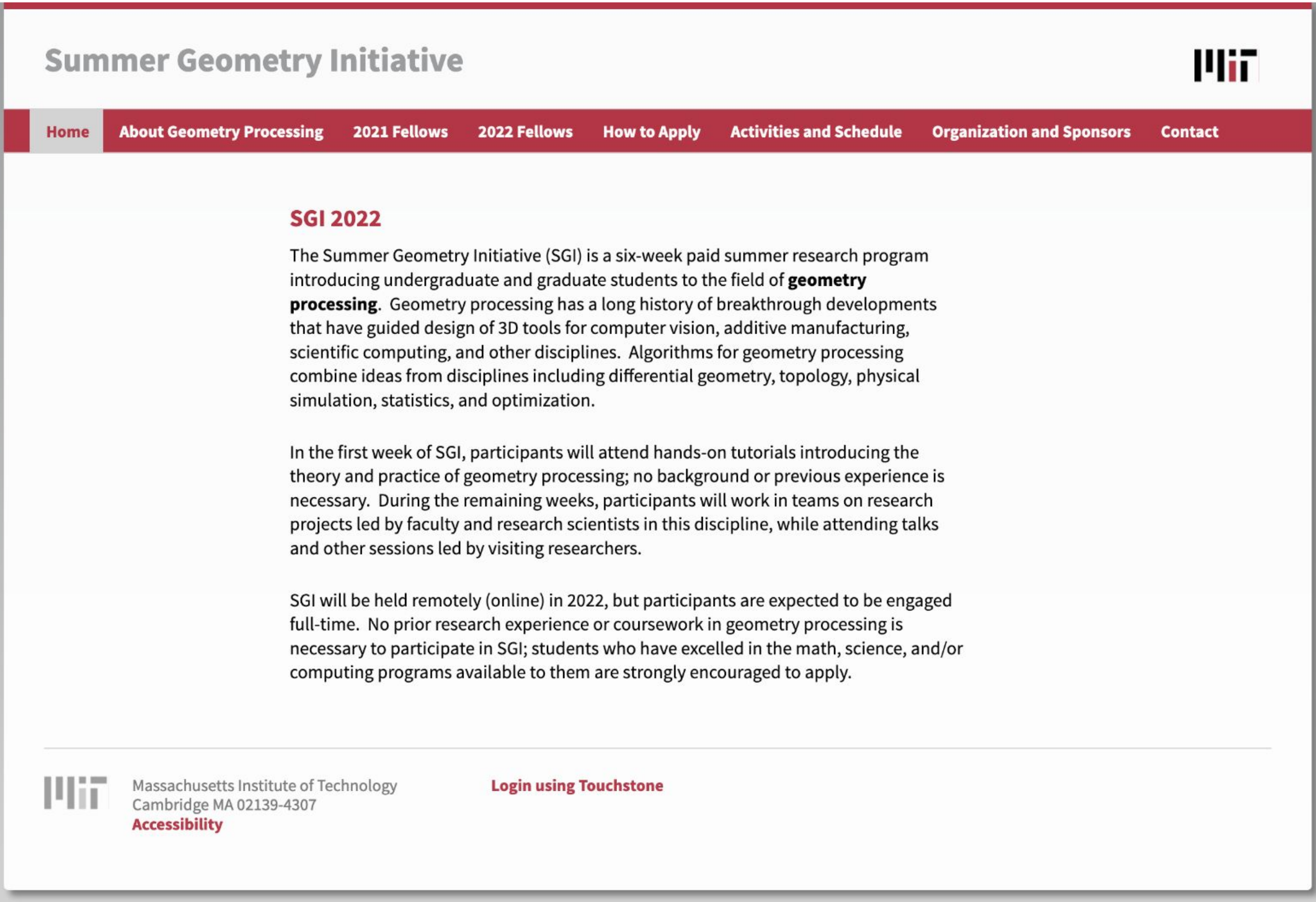


Summer Geometry Initiative

SGI 2022

The Summer Geometry Initiative (SGI) is a six-week paid summer research program introducing undergraduate and graduate students to the field of **geometry processing**. Geometry processing has a long history of breakthrough developments that have guided design of 3D tools for computer vision, additive manufacturing, scientific computing, and other disciplines. Algorithms for geometry processing combine ideas from disciplines including differential geometry, topology, physical simulation, statistics, and optimization.

In the first week of SGI, participants will attend hands-on tutorials introducing the theory and practice of geometry processing; no background or previous experience is necessary. During the remaining weeks, participants will work in teams on research projects led by faculty and research scientists in this discipline, while attending talks and other sessions led by visiting researchers.

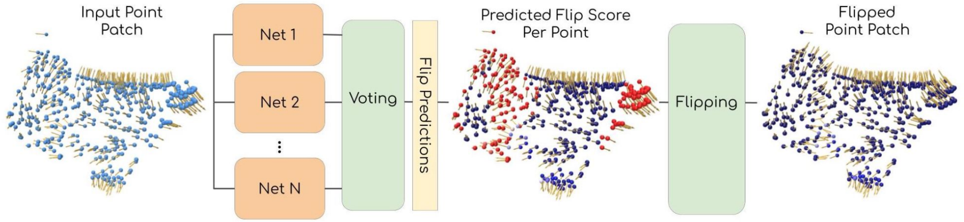
SGI will be held remotely (online) in 2022, but participants are expected to be engaged full-time. No prior research experience or coursework in geometry processing is necessary to participate in SGI; students who have excelled in the math, science, and/or computing programs available to them are strongly encouraged to apply.

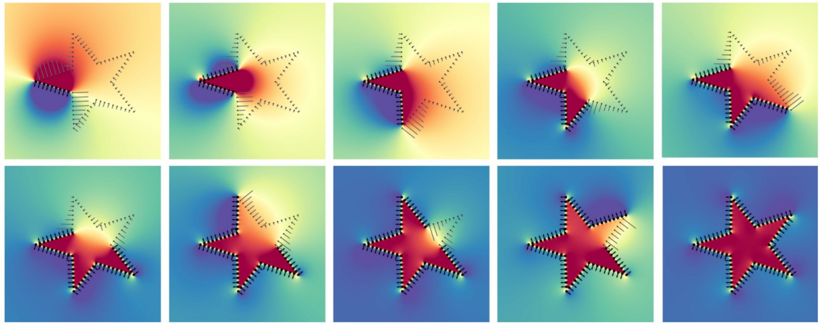
Massachusetts Institute of Technology
Cambridge MA 02139-4307
Accessibility

Login using Touchstone

Orienting Point Clouds (Kazhdan)

- Classic problem: how can we assign normals to elements of a point cloud?
- Increasingly less relevant as modern LiDAR scanning returns oriented points, but some older datasets may lack this data
- Inspiration: [Orienting Point Clouds with Dipole Propagation]() (2021)
  - Idea: come up with an algorithmic method of solving the "local" phase of the orienting problem (rather than using a neural network)





Orienting Point Clouds (Kazhdan)

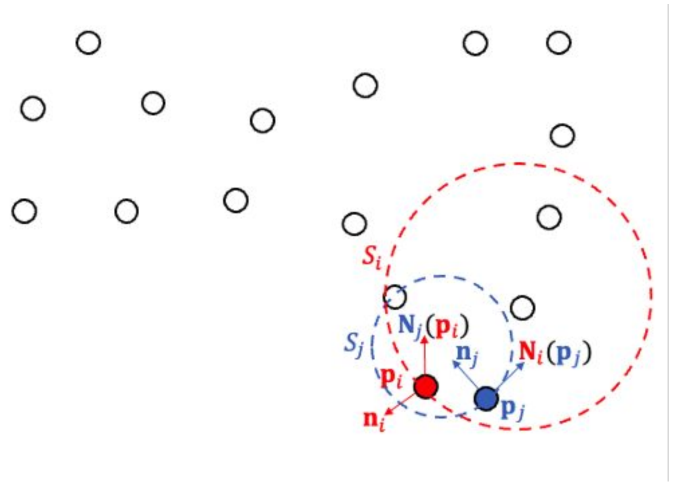
$$E(\alpha) = \alpha^T \cdot E \cdot \alpha$$

Local phase

- Instead of letting a neural network orient the points in each patch: fit an implicit function around each point to guide our decision
  - We spent the most time on this as SGI participants
- Clusters chosen by calculating k-nearest neighbors (k-d tree)
- The success of this orienting method depends on the goodness of all neighboring fits

$$\tilde{E_{ij}} = \frac{\langle N_i(p_i), N_j(p_i)\rangle + \langle N_i(p_j), N_j(p_j)\rangle}{2}$$

a : a vector assigning signs to each normal, $\{-1, +1\}^n$



Orienting Point Clouds (Kazhdan)

Global phase

- We'd rather not try all possible combinations of signs for the total set of points: clustering allows us to reduce the dimensionality of the problem at each step
- Greedy approach: try all 2^c combinations of signed clusters and see which one maximizes this energy
- Merge clusters up one step, and repeat at the next level
    - End with one coherently oriented cluster

$$E(\alpha) = \alpha^T N \alpha$$

$$a_i = \beta_i A_{c_i i}$$

Sign of point i

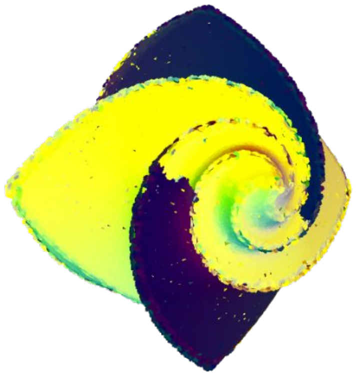
Assignment of signs to clusters

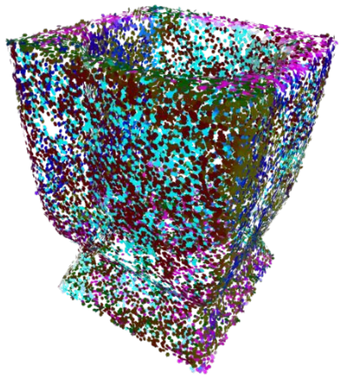
Assignment of points to clusters

$$E_A(\beta) = \beta^T A^T N A \beta$$

Our Work: New fits

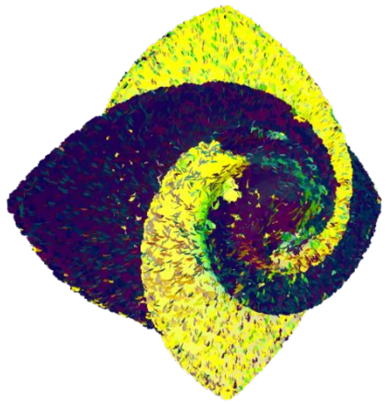
- Provided: linear fit, general quadratic fit
- Both work well for point clouds with certain features – also identified clear failure cases for both


Linear fit


Quadratic fit

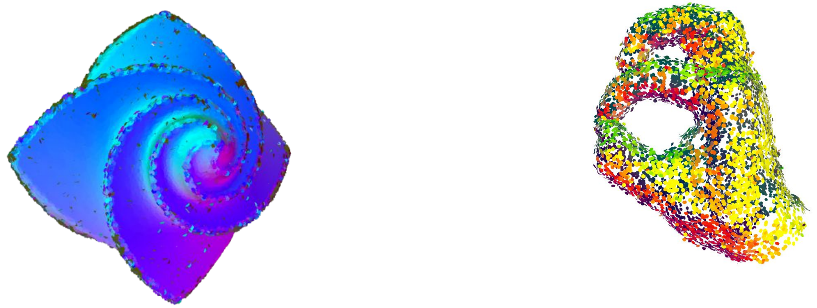
Our Work: New fits

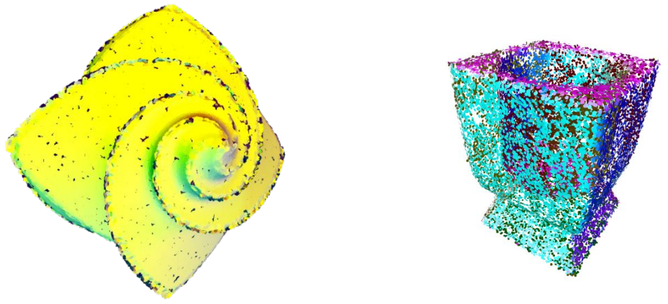
- First attempt: spherical fit using least squares approach
- Generally speaking, not very useful…



Our Work: New fits

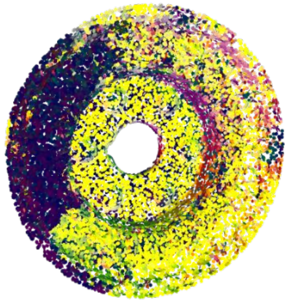
- Subsequent attempts at fitting local neighborhoods were much more successful
- Parabolic fit: limits quadratic fit to parabolas with evenly-scaled major and minor axes
- Associative fit: a “best of both worlds” approach that first attempts to use one fit, and falls back on a secondary fit type if the estimated goodness of fit doesn’t clear a threshold value
    - Templated C++ code makes it easy to pass in our other fits as parameters to the associative fit

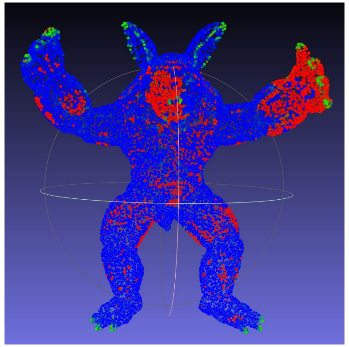

Parabolic Fit

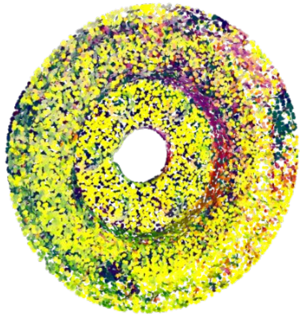

Associative<Linear, Quadratic> Fit

Our Work: Touch-ups

- Introduced use of distance weighting (diminish contribution of points farther from the cluster's center)
- Similarity scores for assessing performance: comparison with ground truth normal lines & orientations



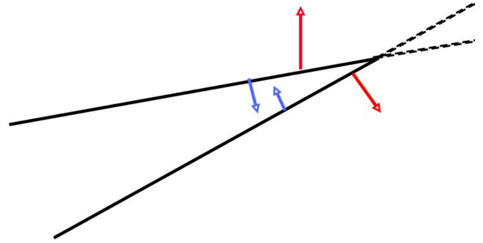


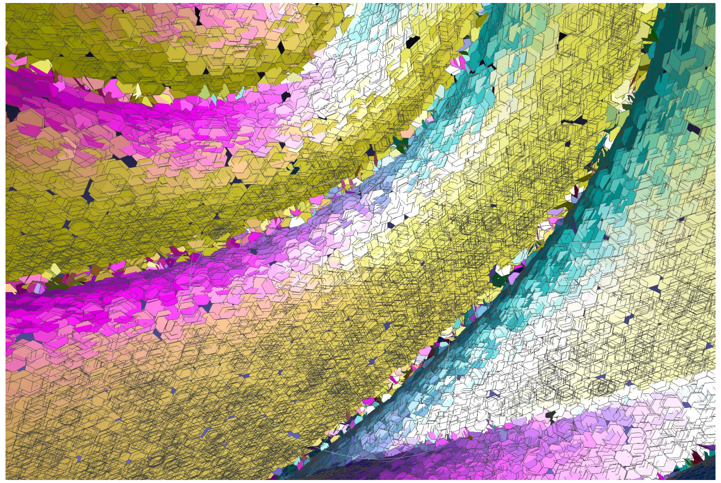


Pulley with distance weighting applied in construction of fits

Continuing work

- Accounting for non-uniform sampling & local changes in point cloud density (dynamic neighborhood size)
- New implicit fit types: “wedge” or intersection of planes, to more accurately capture sharp edges



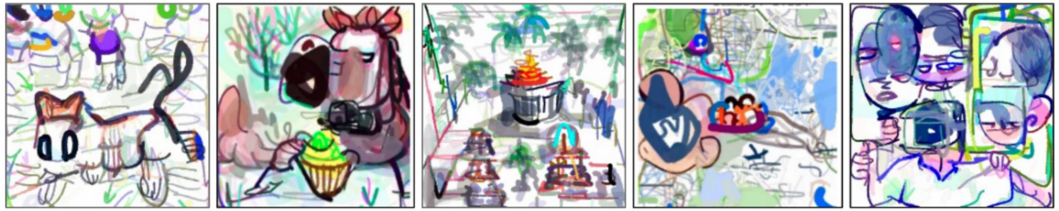


Interior of “flower” produced with preliminary wedge fit

Text-Guided Shape Assembly (Gadelha)

- CLIP is a neural network providing a joint embedding of text and images
- Goal: generate images from geometric primitives (curves, triangles, cuboids) using differentiable rendering
  - Reliance on CLIP means that we don't have to train a network ourselves: focus on constructing a sensible optimization problem instead
- Inspiration: CLIPDraw (2021), which constructs images from curves

```
Top predictions:

          dog: 13.28%
        puppy: 5.20%
          pet: 4.18%
        white: 2.96%
        owner: 2.31%
Top predictions:

          car: 18.31%
      taxicab: 10.27%
         polo: 2.40%
    insurance: 2.12%
       bumper: 1.55%
```

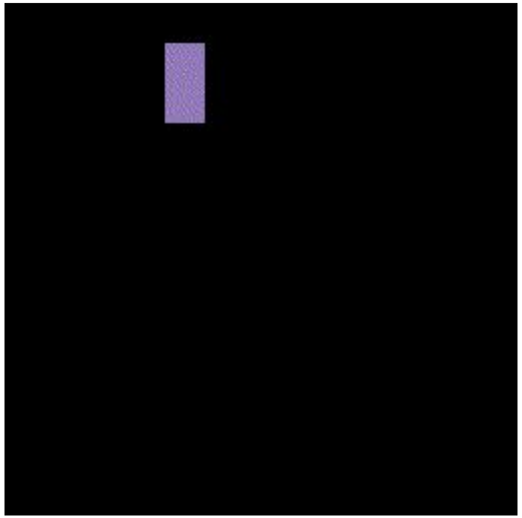


"A drawing of a cat". "Horse eating a cupcake". "A 3D rendering of a temple". "Family vacation to Walt Disney World". "Self".

Text-Guided Shape Assembly (Gadelha)

- Since text and images are embedded in the same vector space, we can use cosine similarity as a metric for assessing “closeness”
  - Loss function incorporates negative cosine similarity: we want to maximize this quantity to minimize the loss function
- By passing every draw operation through a differentiable rendering pipeline, we can keep track of gradients for each parameter & perform gradient descent to reach a locally optimal solution

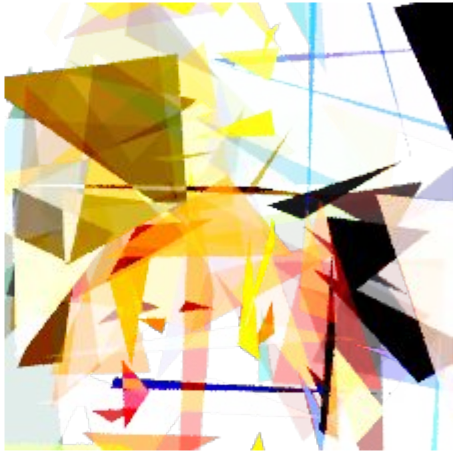
$$\text{loss} = -\langle \Phi(\text{Image}), \Phi(\text{Text}) \rangle$$



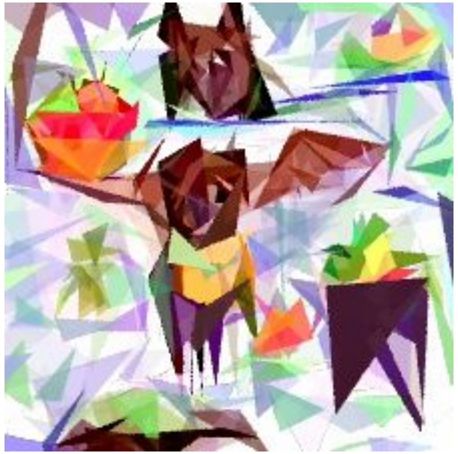
(Courtesy of diffvg)

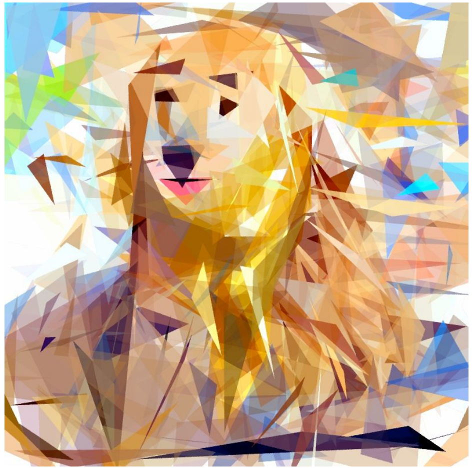
First attempts: gradient-based optimization of triangles

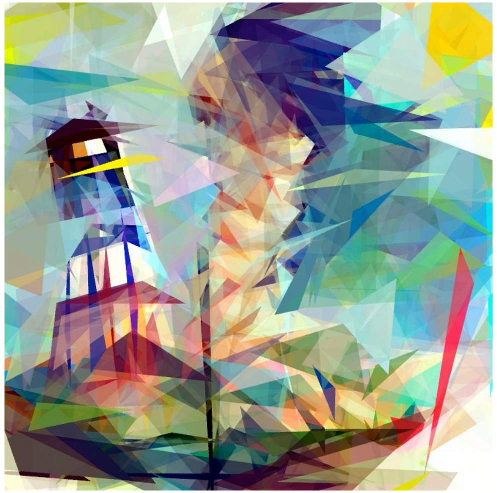
- Employing a similar approach to CLIPDraw, we attempted to make use of diffvg, adapting the approach to suit triangles
- We (very quickly) discovered the importance of applying data augmentations: otherwise, the dataset being fed to the model isn't comprehensive enough to produce sensible results
  - Random crop, rotate, perspective project enabled through PyTorch


"golden retriever" with no augmentations incorporated into the loss function

After adding augmentations…

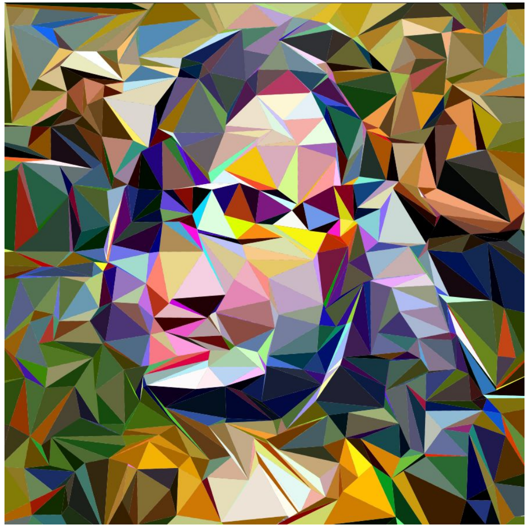

“fruit bat”

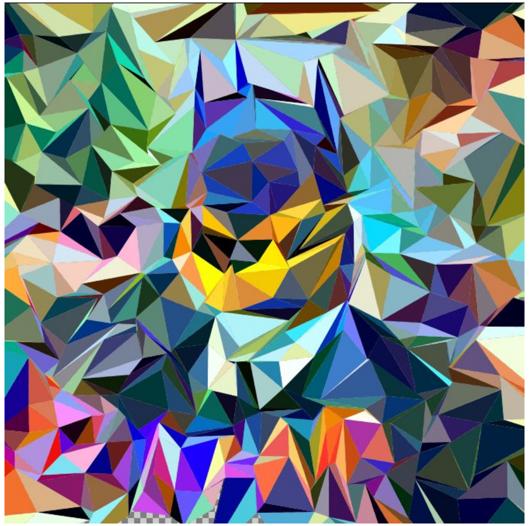

“golden retriever”

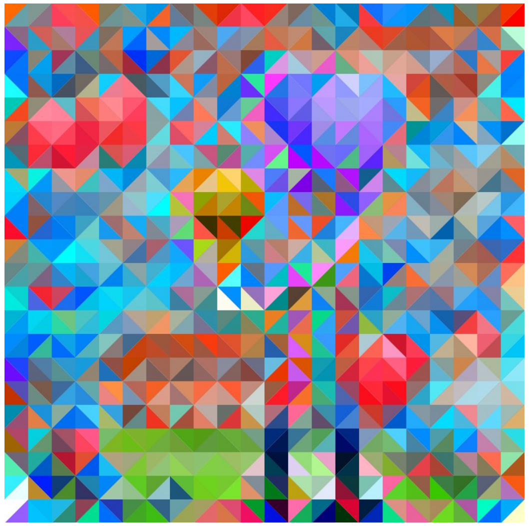

“lighthouse”

Changing which transformations are applied has profound effects on the results!

Meshing triangles

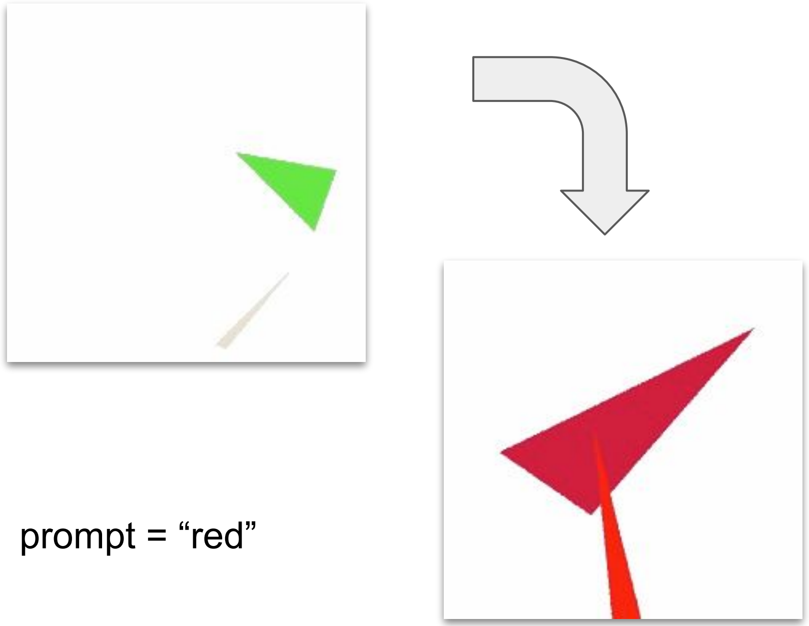

“Mona Lisa”


“Batman”


“Balloons” (fixed set of vertices)

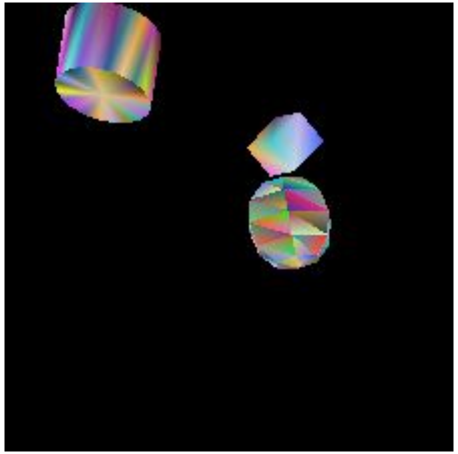
Other 2D approaches: evolutionary model

- Instead of following gradient descent (series of directed steps), introduce random changes & keep ones that improve the outcome
- Too slow unless random changes are calibrated toward the desired type of results



Next week: 3D shape assemblages

- We turned to a 3D differentiable renderer: Nvdiffrast
- Make use of the extra dimension to construct scenes of 3D primitives that are intelligible when viewed from multiple camera angles
- Major challenges posed by the fact that Nvdiffrast provides little in the way of out-of-the-box data structures



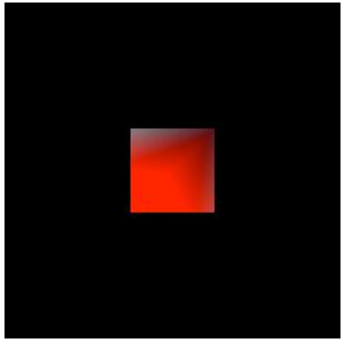
Next week: 3D shape assemblages

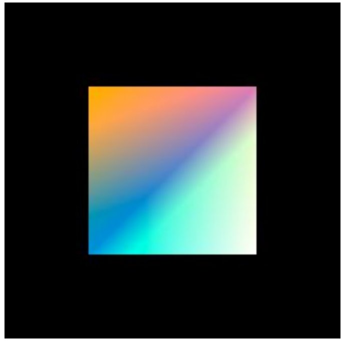
Milestones

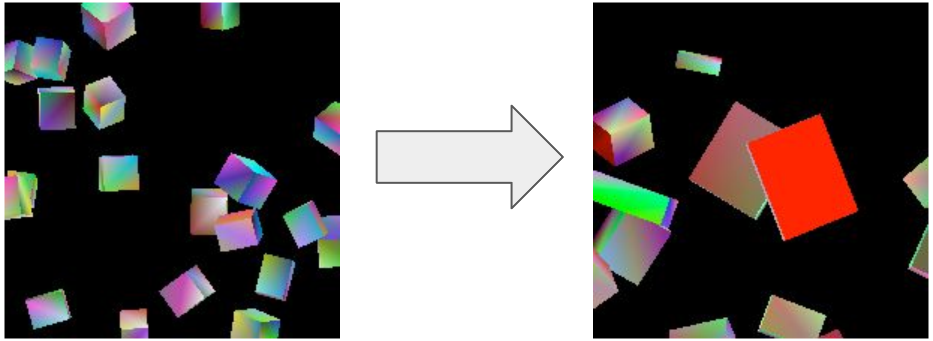
1. Define geometric primitives by constructing buffer objects
2. Construct means of transforming individual primitives
   - Goal: optimize parameters of each transformation, which means that each transform operation must be differentiable
3. Decomposition of primitives back into triangle elements, such that Nvdiffrast can properly render the scene

Result: the beginnings of a pipeline that seems responsive to certain features of each text prompt

- With more time, exploring the wide range of possibilities for 3D data augmentations would have produced more interesting results

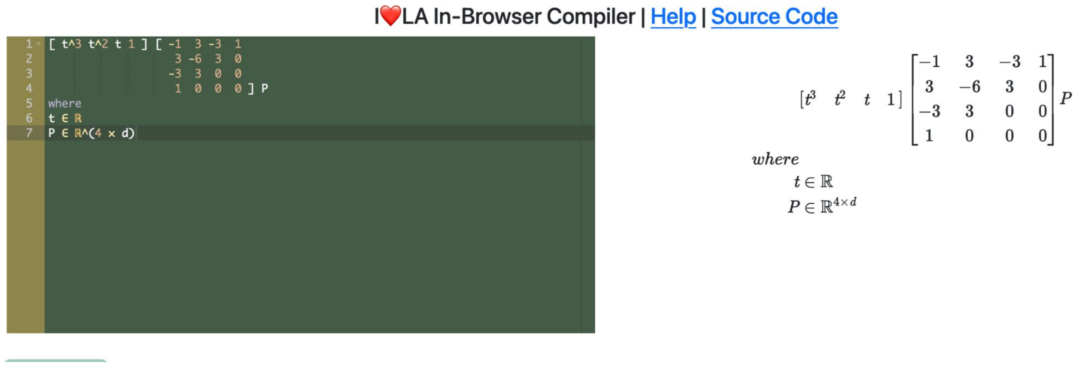

prompt = “red”


prompt = “rainbow”


prompt = “a centered red square”

I❤️LA: Compilable Markdown for Linear Algebra (Gingold, Jacobson)

- New language specification designed to make it easier to translate “chalkboard math” into working code (first paper, 2021)
  - User formulates a problem in the syntax of I❤️LA, which is then translated to code in one of several languages
    - C++ with Eigen
    - Python with NumPy
    - MATLAB
    - LaTeX
  - Supports unicode
- Compiler infrastructure that can be run on a web server or locally
  - Web


I❤️LA In-Browser Compiler | Help | Source Code

```
[ t^3 t^2 t 1 ] [ -1  3 -3  1
                   3 -6  3  0
                  -3  3  0  0
                   1  0  0  0 ] P
where
t ∈ ℝ
P ∈ ℝ^(4 × d)
```

$$\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} P$$

where

$t \in \mathbb{R}$

$P \in \mathbb{R}^{4 \times d}$

Enhancement proposals

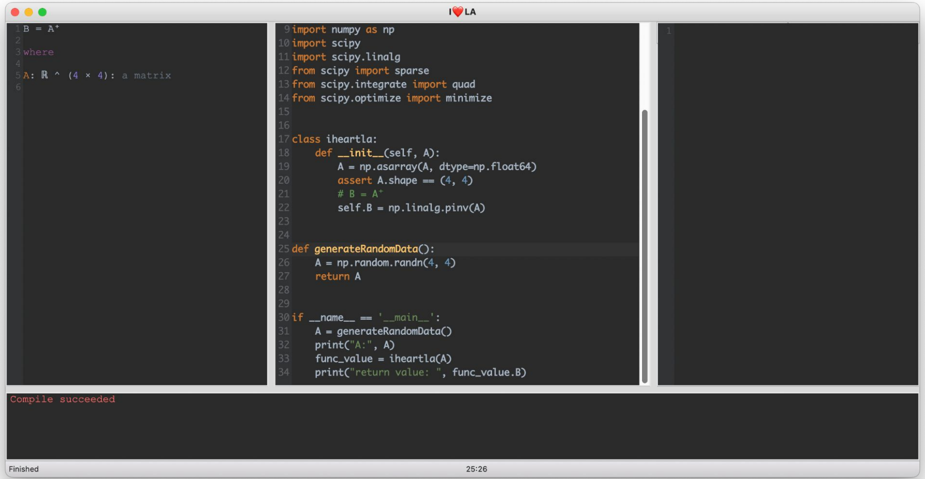
- Centered around a collection of optimization problems written by Alec Jacobson
- Goal: formulate the desired I❤️LA for each of these problems & work backwards to identify new language features
  - Assess alternative formulations of each problem that should be treated as equivalent by the compiler
- Many proposed features arose from these discussions
  - ∀ “for all” (throughout)
  - Slicing to produce submatrices (#3)
  - Identity matrix

New operator: Pseudoinverse

To add a new feature, the following must be modified:

- EBNF grammar (specified directly as input to Tatsu, which then generates a parser)
- type_walker functions
- Intermediate node representation
- Code generators (for each desired language)

Merged into official I❤️LA repository :)


I❤️LA

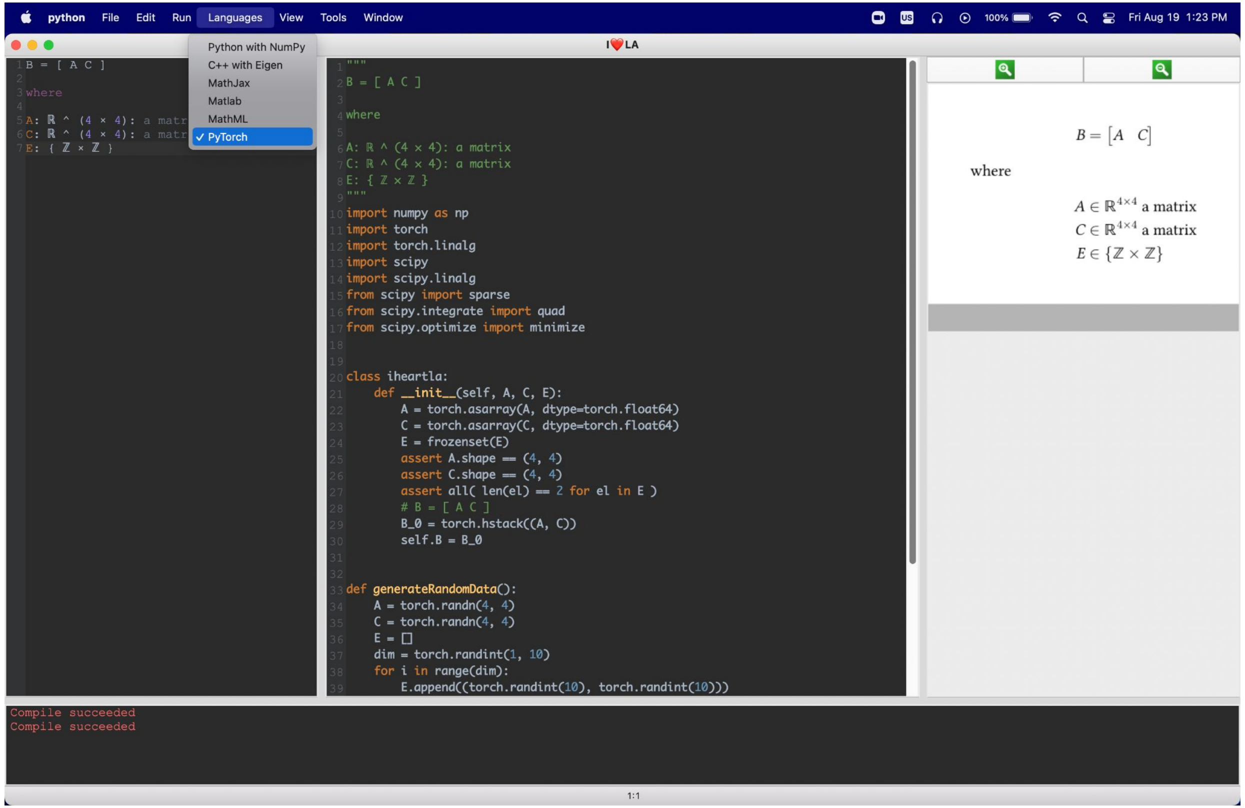
```
B = A⁺

where

A: ℝ ^ (4 × 4): a matrix
```

```python
import numpy as np
import scipy
import scipy.linalg
from scipy import sparse
from scipy.integrate import quad
from scipy.optimize import minimize


class iheartla:
    def __init__(self, A):
        A = np.asarray(A, dtype=np.float64)
        assert A.shape == (4, 4)
        # B = A⁺
        self.B = np.linalg.pinv(A)


def generateRandomData():
    A = np.random.randn(4, 4)
    return A


if __name__ == '__main__':
    A = generateRandomData()
    print("A:", A)
    func_value = iheartla(A)
    print("return value: ", func_value.B)
```

```
Compile succeeded
```

PyTorch backend

- Compatible with most of I❤️LA’s core features
    - Missing some sparse matrix operations due to lack of support in PyTorch itself



```
B = [ A C ]

where

A: ℝ ^ (4 × 4): a matr
C: ℝ ^ (4 × 4): a matr
E: { ℤ × ℤ }
```

```
"""
B = [ A C ]

where

A: ℝ ^ (4 × 4): a matrix
C: ℝ ^ (4 × 4): a matrix
E: { ℤ × ℤ }
"""
import numpy as np
import torch
import torch.linalg
import scipy
import scipy.linalg
from scipy import sparse
from scipy.integrate import quad
from scipy.optimize import minimize


class iheartla:
    def __init__(self, A, C, E):
        A = torch.asarray(A, dtype=torch.float64)
        C = torch.asarray(C, dtype=torch.float64)
        E = frozenset(E)
        assert A.shape == (4, 4)
        assert C.shape == (4, 4)
        assert all( len(el) == 2 for el in E )
        # B = [ A C ]
        B_0 = torch.hstack((A, C))
        self.B = B_0


def generateRandomData():
    A = torch.randn(4, 4)
    C = torch.randn(4, 4)
    E = []
    dim = torch.randint(1, 10)
    for i in range(dim):
        E.append((torch.randint(10), torch.randint(10)))
```

$$B = \begin{bmatrix} A & C \end{bmatrix}$$

where

$$A \in \mathbb{R}^{4\times 4} \text{ a matrix}$$
$$C \in \mathbb{R}^{4\times 4} \text{ a matrix}$$
$$E \in \{\mathbb{Z} \times \mathbb{Z}\}$$

```
Compile succeeded
Compile succeeded
```

Blog Posts

- Orienting Point Clouds
- Text-Guided Shape Assembly