Geospatial Agent-Based Modeling with Mesa-Geo

Boyu Wang

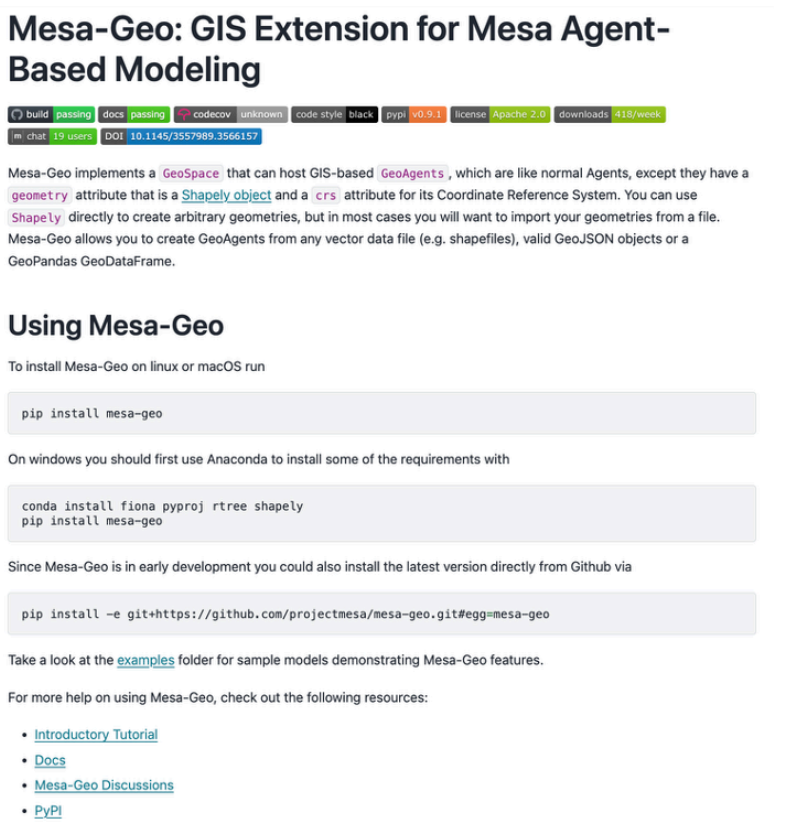
2026.01.04

Outline

1. **What are Mesa & Mesa-Geo?**
2. **Why Geospatial ABM?**
3. **ABM Frameworks with GIS Support**
4. **GIS Primer**
   - Spatial data models
   - File formats
   - Geospatial Python packages
5. **Mesa-Geo Architecture**
6. **Core building blocks**
   - GeoAgent
   - Cell
   - GeoSpace
7. **Visualizing GeoSpace**
8. **Example models**
9. **GSoC Idea 2025**
10. **Towards Mesa-Geo v1.0**
11. **Homework**
12. **Resources**

What are Mesa & Mesa-Geo?

- **Mesa** is an open-source agent-based modelling framework written in Python. Its modular design allows you to build, visualize and analyse ABMs quickly.
- **Mesa-Geo** is Mesa's GIS extension. It adds spatial data structures and functions so you can import, manipulate, visualize and export geographic information for ABMs.



Mesa-Geo: GIS Extension for Mesa Agent-Based Modeling

build passing | docs passing | codecov unknown | code style black | pypi v0.9.1 | license Apache 2.0 | downloads 418/week | chat 19 users | DOI 10.1145/3557989.3566157

Mesa-Geo implements a `GeoSpace` that can host GIS-based `GeoAgents`, which are like normal Agents, except they have a `geometry` attribute that is a Shapely object and a `crs` attribute for its Coordinate Reference System. You can use `Shapely` directly to create arbitrary geometries, but in most cases you will want to import your geometries from a file. Mesa-Geo allows you to create GeoAgents from any vector data file (e.g. shapefiles), valid GeoJSON objects or a GeoPandas GeoDataFrame.

Using Mesa-Geo

To install Mesa-Geo on linux or macOS run

```
pip install mesa-geo
```

On windows you should first use Anaconda to install some of the requirements with

```
conda install fiona pyproj rtree shapely
pip install mesa-geo
```

Since Mesa-Geo is in early development you could also install the latest version directly from Github via

```
pip install -e git+https://github.com/projectmesa/mesa-geo.git#egg=mesa-geo
```

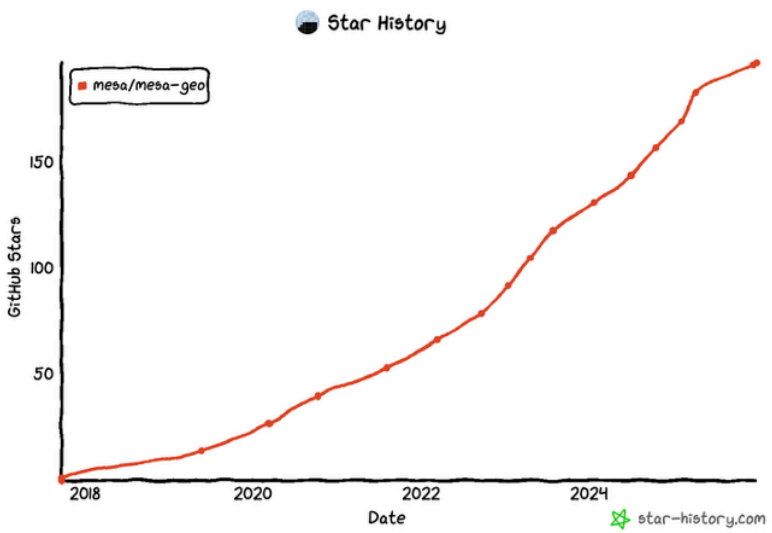
Take a look at the examples folder for sample models demonstrating Mesa-Geo features.

For more help on using Mesa-Geo, check out the following resources:

- Introductory Tutorial
- Docs
- Mesa-Geo Discussions
- PyPI

What are Mesa & Mesa-Geo?

- **Mesa-Geo:** open-source project created in 2017.
- Transferred into Mesa GitHub organization in 2022.
- ~2k downloads per month (PyPI statistics as of Dec 2025), about 5% to 10% of total Mesa downloads.

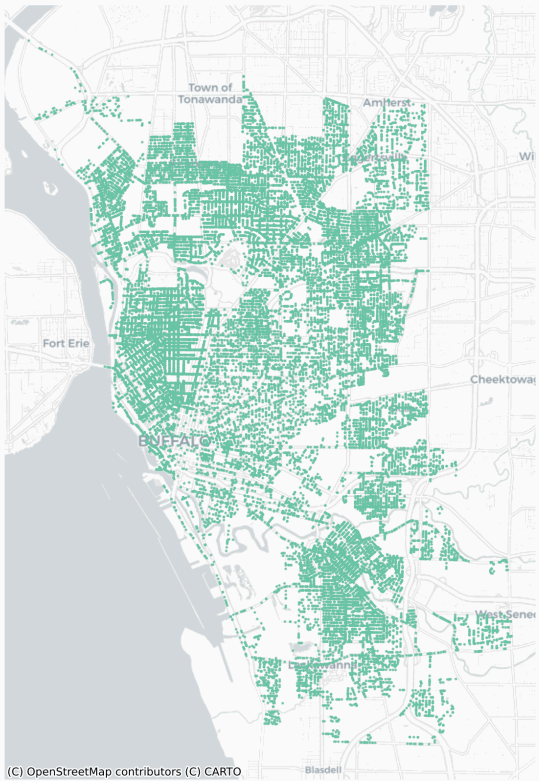


Why Geospatial ABM?

Many real-world phenomena (land use, population density, transport) are spatially heterogeneous. Classic ABMs can't capture real geography or handle map projections.

Integrating GIS into ABMs enables:

- **Basemaps** for real-world backgrounds
- **Spatial queries** (e.g., neighbourhood search, buffering)
- **Raster & vector overlay** to combine multiple data sources



An Urban Commuting Model with Synthetic Population in Buffalo, NY

ABM Frameworks with GIS Support

	NetLogo	Repast Simphony	AnyLogic	MASON	GAMA	Mesa	AgentScript
Initial Release Year	1999	2000	2000	2003	2009	2015	2018
License	GPL	BSD	Proprietary	Academic Free License 3.0	GPLv3	Apache 2.0	GPLv3
Implementation Language	Scala, Java	Java	Java	Java	Java	Python	JavaScript
Model Development Language or Interface	NetLogo	ReLogo (dialect of Logo), statecharts (point-and-click), Groovy, Java	GUI, Java, UML-RT	Java	GAML (GAma Modeling Language)	Python	JavaScript
GIS Support	Through gis-extension	Yes	Yes	Through GeoMason extension	Yes	Through Mesa-Geo extension	Yes

GIS Primer

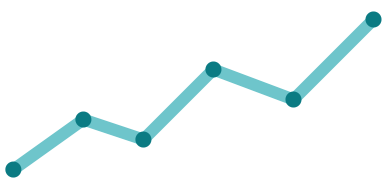
- Spatial Data Models
- File Formats
- Geospatial Python Packages

Spatial Data Model: Vector Data

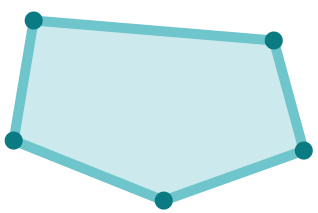
- Represent **discrete** features such as points (e.g., trees), lines (e.g., roads) and polygons (e.g., administrative boundaries).
- Useful to model individual objects (agents, roads, regions), compute spatial relationships (within / intersects / distance / buffer), and perform spatial operations such as spatial joins and overlays.

Point: a single coordinate, representing a location such as a tree.

Line: a sequence of coordinates forming a path, such as a road or river.

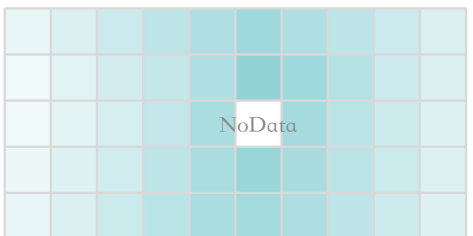


Polygon: a closed shape representing an area, such as a park or administrative boundary.



Spatial Data Model: Raster Data

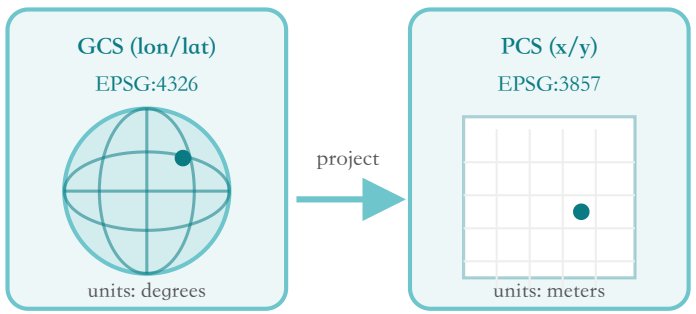
- Represent **continuous** surfaces as a regular grid of cells (pixels). Each cell stores a numeric value (e.g., elevation, land cover or population density).
- Useful for modelling spatial phenomena that vary smoothly across space, such as terrain elevation, rainfall or suitability surfaces.



cells (pixels) with values

Coordinate Reference Systems (CRS)

- A **coordinate reference system** defines how 2D coordinates correspond to real-world locations. Without a common CRS, overlaying layers from different sources leads to misalignment.
- **GCS (Geographic CRS)** uses angular units (**lat/lon**) on a reference ellipsoid (e.g., **WGS 84 / EPSG:4326**).
- **PCS (Projected CRS)** uses linear units (**meters/feet**) after projecting the Earth to a plane (e.g., **Web Mercator / EPSG:3857**, **UTM zones**).
- **Good practice:** use **one shared project CRS**. If multiple data sources come in different CRSs, **reproject** them into the project CRS before overlay/analysis.



ⓘ Learn more

Coordinate Reference Systems in QGIS Documentation:
https://docs.qgis.org/latest/en/docs/gentle_gis_introduction/coordinate_reference_systems.html

File Formats

- **Shapefile (.shp)**: A widely used vector format and de facto standard for GIS data. A shapefile is actually a collection of files (e.g., .shp, .shx, .dbf), each storing geometry, index and attribute data. Shapefiles store only one geometry type per file (points, lines or polygons).
- **GeoJSON**: A human-readable JSON format commonly used for web mapping. Supports multiple geometry types in a single file and is easy to share via REST APIs.
- **KML / KMZ**: An XML-based format developed for Google Earth and now an OGC standard. KML allows multiple geometries and even raster data when compressed as KMZ.
- **GeoPackage (.gpkg)**: A single-file SQLite database that can store multiple vector and raster layers. It is the default vector format for QGIS and increasingly popular.
- **GeoTIFF (.tif)**: The most common raster format; a TIFF image with embedded geographic metadata. Supports multiple bands, high bit depths and georeferencing tags.
- **GeoParquet (.parquet)**: A columnar, compressed format for vector data based on Apache Parquet. It is efficient for large datasets and works well with cloud/data-lake workflows.

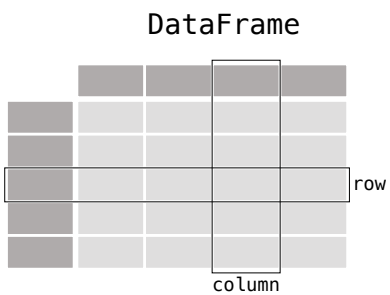
> ⓘ **Quick reference**
>
> Full list of GIS formats: https://gisgeography.com/gis-formats

Geospatial Python Packages

- **GeoPandas:** Extends pandas to handle **vector data.** Each row of a `GeoDataFrame` represents a feature with a geometry (Shapely object) and attribute columns; supports spatial joins, overlays and CRS transformations.
- **Rasterio:** Provides Pythonic access to **raster datasets.** Built on GDAL, it reads and writes formats such as GeoTIFF, and supports windowed reads, metadata access and creation of new rasters.
- **Shapely:** Performs geometric operations (intersection, union, buffering) on planar features.
- **PyProj:** Interfaces with PROJ to define and convert between coordinate reference systems (CRS).
- **Rtree:** Implements spatial indexing, enabling fast nearest-neighbour and intersection queries.

geopandas

pandas DataFrame

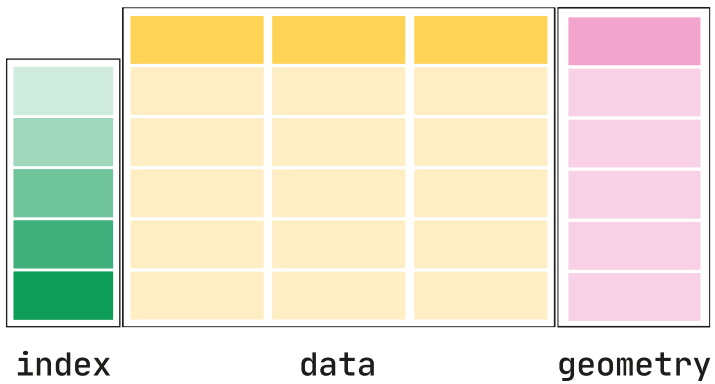


Source: pandas documentation

Tabular only

No `geometry` • No CRS • No spatial ops

geopandas GeoDataFrame



Source: geopandas documentation

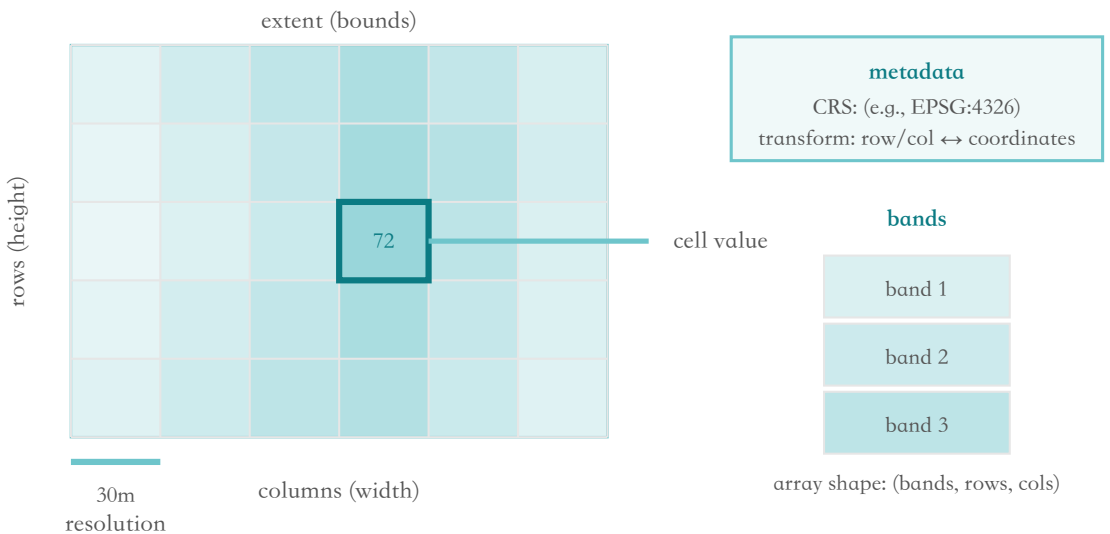
Geospatial table

`geometry` column • CRS-aware • spatial ops

(`sjoin`, `overlay`, `to_crs`, etc)

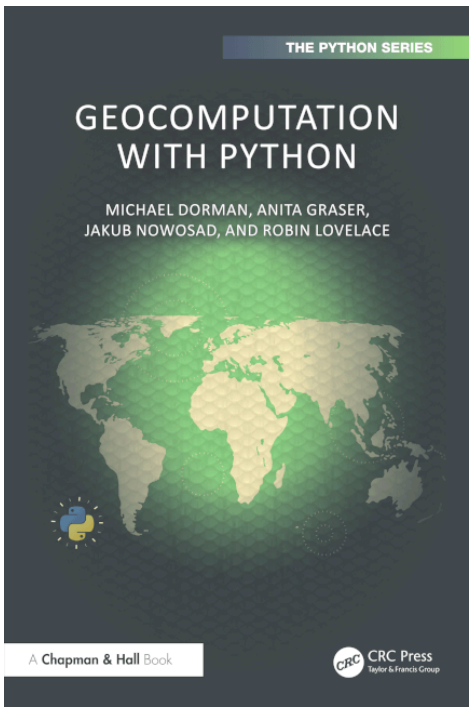
rasterio

- Read/write raster datasets (e.g., GeoTIFF) as **NumPy arrays**.
- Key concepts:
  - **rows × cols**: matrix shape
  - **cell**: one pixel in the grid
  - **cell value**: numeric attribute (e.g., elevation)
  - **resolution**: cell size (e.g., 30m × 30m)
  - **extent**: bounding box of the raster
  - **bands**: multiple layers (e.g., RGB / time / variables)
  - **CRS**: reference system that gives meaning to coordinates (e.g., EPSG code)

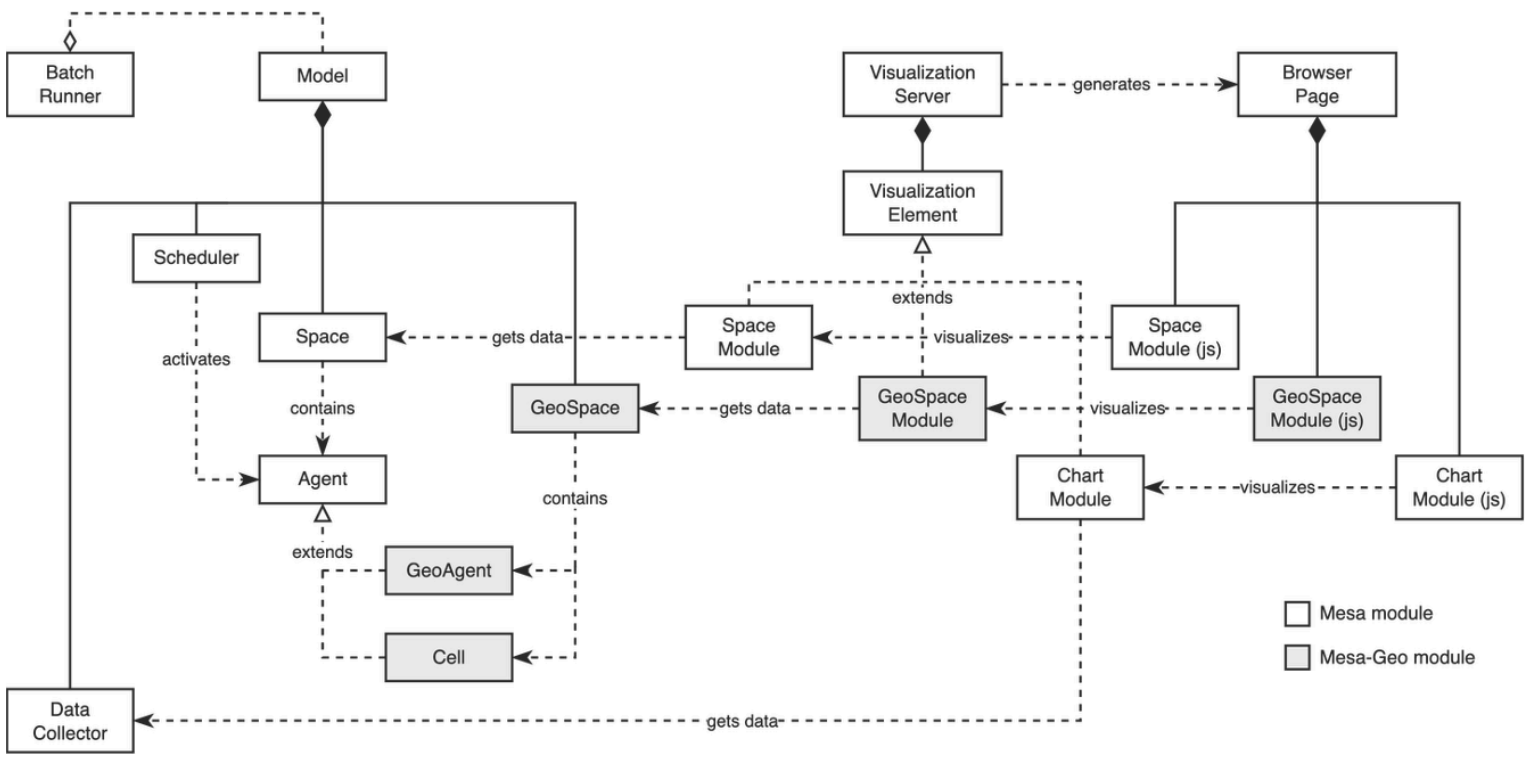


Further Reading

Geocomputation with Python: an open-source book on geographic data analysis with Python, publicly available at `py.geocompx.org`



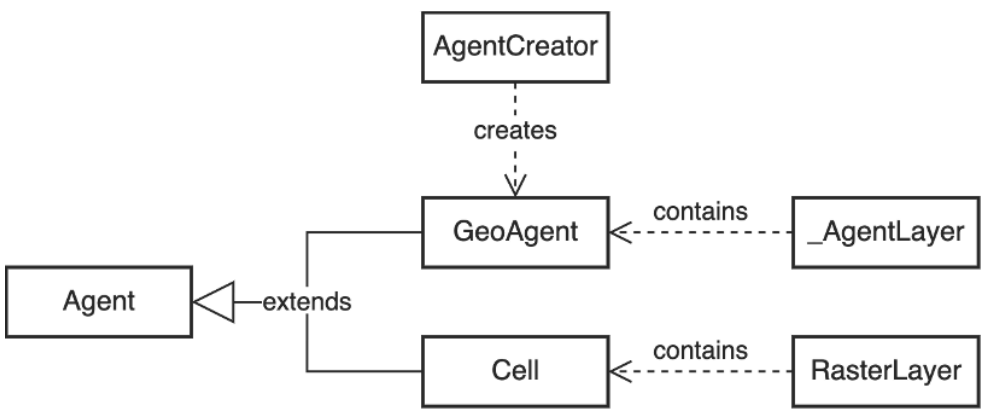
Mesa-Geo Architecture



High-level component diagram of Mesa & Mesa-Geo

GeoAgent & Cell

- A **GeoAgent** is a Mesa agent with a `geometry` (a Shapely shape such as a polygon or point) and a coordinate reference system (`crs`). It can perform geometric and topological operations (e.g., intersection tests) and knows its location in geographic space.
- A **Cell** is an agent that lives on a raster grid. It stores an `(x, y)` coordinate as well as `(row, col)` indices that map into a matrix representation.



Class diagram of the `Agent` , `GeoAgent` , and `Cell` classes

GeoAgent

Define a `GeoAgent` class

```python
# agents.py

import mesa
import mesa_geo as mg

class PersonAgent(mg.GeoAgent):
    def __init__(self, model, geometry, crs, income=0):
        super().__init__(model, geometry=geometry, crs=crs)
        self.income = income

    def step(self):
        ...
```

Create many agents from a GeoDataFrame

```python
# model.py

import geopandas as gpd
import mesa
import mesa_geo as mg

from .agents import PersonAgent

class MyModel(mesa.Model):

    def __init__(self, population_file, crs):
        ...

        # create agents from geodataframe file
        gdf = gpd.read_file(population_file).to_crs(crs)
        creator = mg.AgentCreator(PersonAgent, model=self)
        agents = creator.from_GeoDataFrame(gdf)

        # put agents into a geospace
        self.space = GeoSpace(crs=crs)
        self.space.add_agents(agents)
```

Cell

Define a `Cell` class

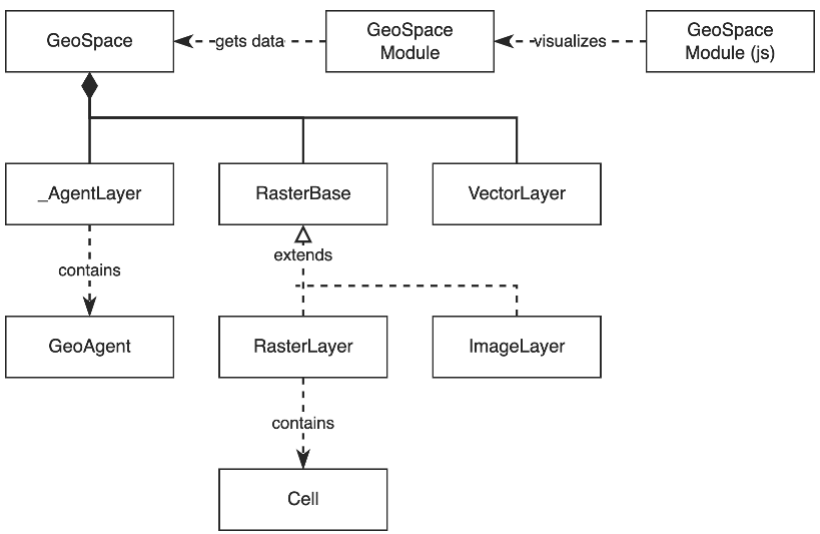
```python
# space.py

import mesa
import mesa_geo as mg

class LakeCell(mg.Cell):
    def __init__(
        self,
        model,
        pos: mesa.space.Coordinate | None = None,
        indices: mesa.space.Coordinate | None = None,
    ):
        super().__init__(model, pos, indices)
        self.elevation = None
        self.water_level = None
        self.water_level_normalized = None

    def step(self):
        ...
```

Create many cells from a raster layer

```python
# space.py

import gzip
import mesa_geo as mg
import numpy as np

class CraterLake(mg.GeoSpace):
    ...

    def set_elevation_layer(self, elevation_gzip_file, crs):
        layer = mg.RasterLayer.from_file(
            elevation_gzip_file,
            model=self.model,
            cell_cls=LakeCell,
            attr_name="elevation",
            rio_opener=gzip.open,
        )
        layer.apply_raster(
            data=np.zeros((1, layer.height, layer.width)),
            attr_name="water_level",
        )
        super().add_layer(layer)
```

GeoSpace

The **GeoSpace** object extends Mesa's `Space` concept to include both vector and raster layers. Each layer can contain agents or pixels, and agents can query the space for neighbours or search for cells that satisfy certain criteria. A simple class diagram is shown below:



Class diagram of `GeoSpace` and its related classes

Visualizing GeoSpace

You typically do **three things:**

1. **Write a portrayal function**
   - **GeoAgent (vector)** → return a *dict* of Leaflet style options (and optional popup text).
   - **Cell (raster)** → return an *(r, g, b, a)* tuple.
2. **Build a viz page** using `SolaraViz(...)`
3. **Add a map component** via `make_geospace_component(...)` (set `zoom`, optional `tiles`, etc.)

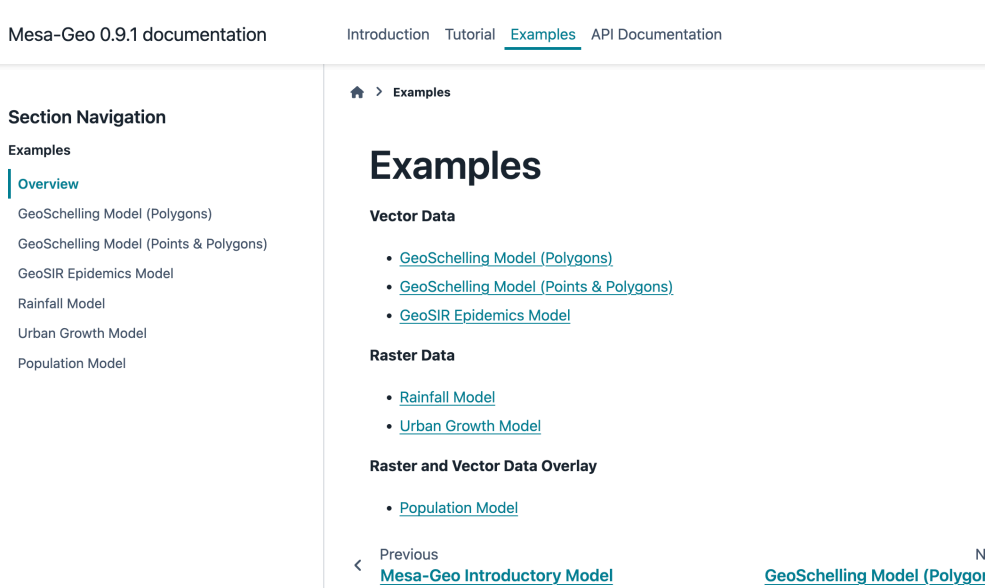
> ⚠ **Mesa version note**
>
> This workflow works with Mesa 3.2, but not 3.3 or 3.4, which introducted breaking changes to visualization API. Pin your Mesa version to 3.2 as a workaround.

Example usage

```
# (often in notebook or app.py; model class stays in model.py)

from mesa.visualization import SolaraViz
from mesa_geo.visualization import make_geospace_component

def draw(agent):
    # vector agents → Leaflet options dict
    if getattr(agent, "atype", None) == "infected":
        return {"color": "red"}
    return {"color": "green", "opacity": 0.6}

model_params = ...

page = SolaraViz(
    MyModel(),
    name="My Awesome Model",
    model_params=model_params,
    components=[
        make_geospace_component(draw, zoom=12),
        ...
    ],
)

page
```

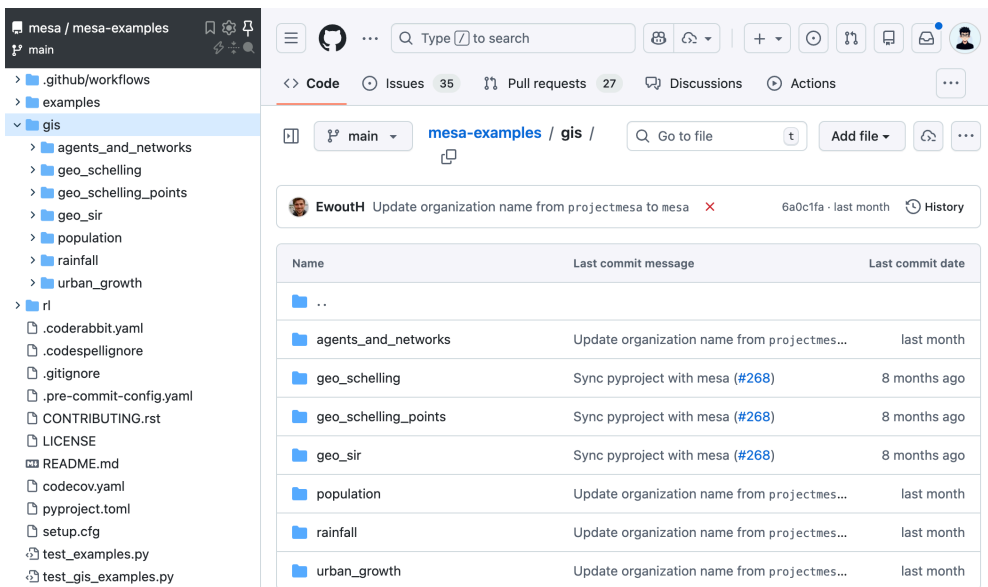
Example Models

Docs



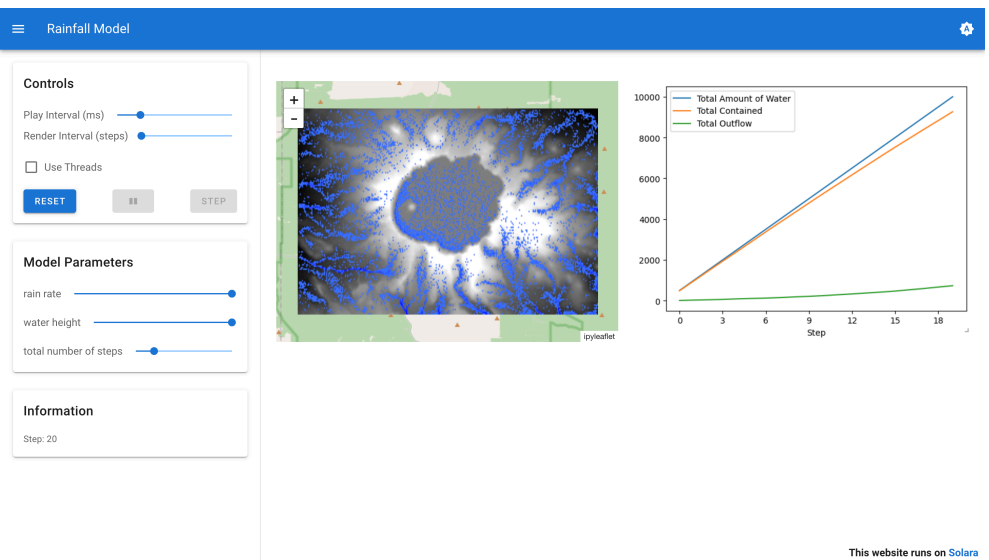
https://mesa-geo.readthedocs.io/stable/examples/overview.html

Source code (mesa-examples/gis)



https://github.com/mesa/mesa-examples/tree/main/gis

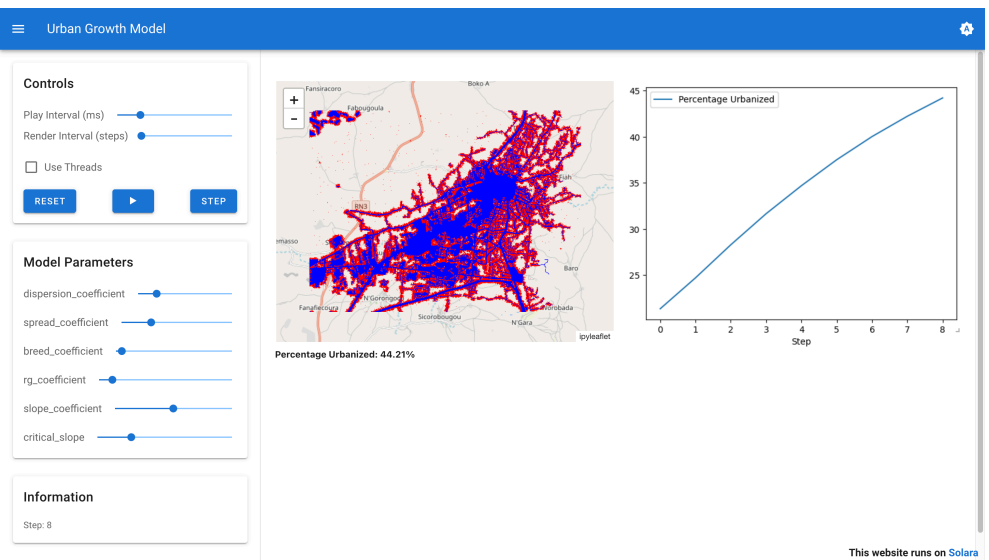
Example: Digital Elevation



Rainfall Model

- **GeoSpace:** a raster layer representing elevations.
- **GeoAgents:** raindrops.
- At each time step, raindrops are randomly created across the landscape to simulate rainfall.
- The raindrops flow from cells of higher elevation to lower elevation based on their eight surrounding cells (i.e., Moore neighbourhood).

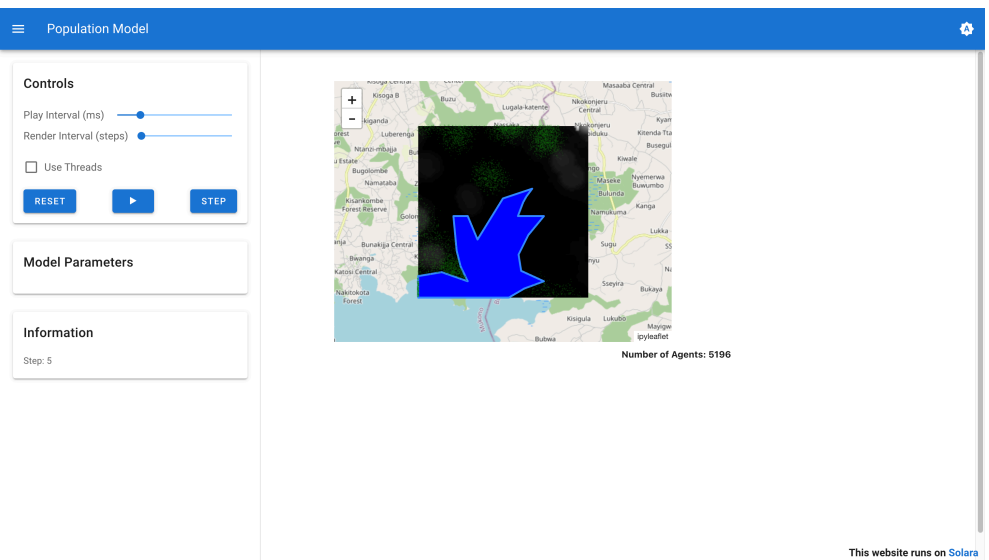
Example: Multiple Raster Layers



Urban Growth Model

- **GeoSpace:** multiple raster layers representing slope, road, land use, urban area, and so on.
- **Cells:** land parcels.
- At each time step, each land parcel is decided whether it is suitable to be urbanized, based on the input raster layers as well as the user defined coefficients.

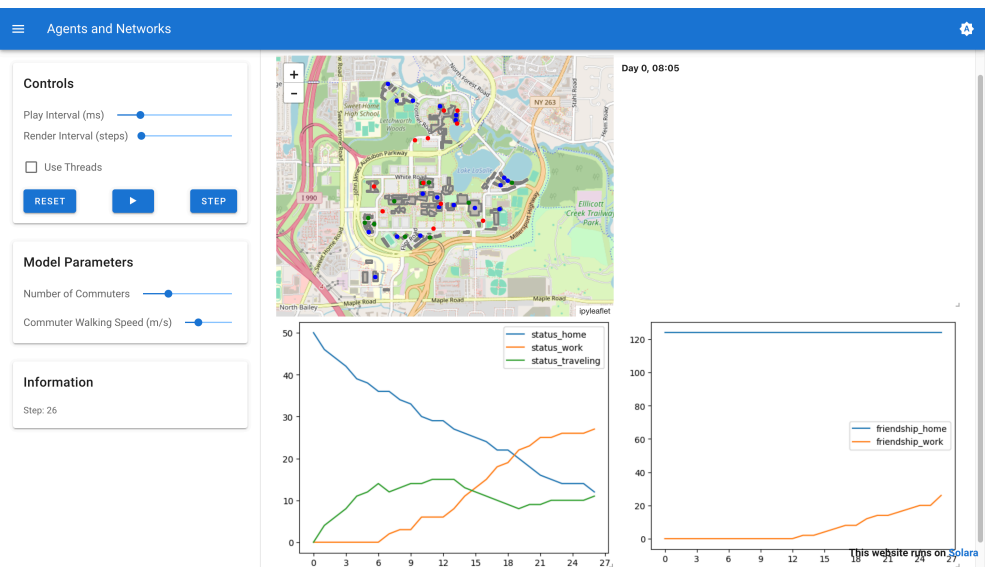
Example: Raster & Vector Overlay



Population Model

- **GeoSpace:**
  - a raster layer of population data for each cell.
  - a vector layer representing a lake.
- **GeoAgent:** people, created based on the population data.
- The agents move randomly to neighbouring cells at each time step.

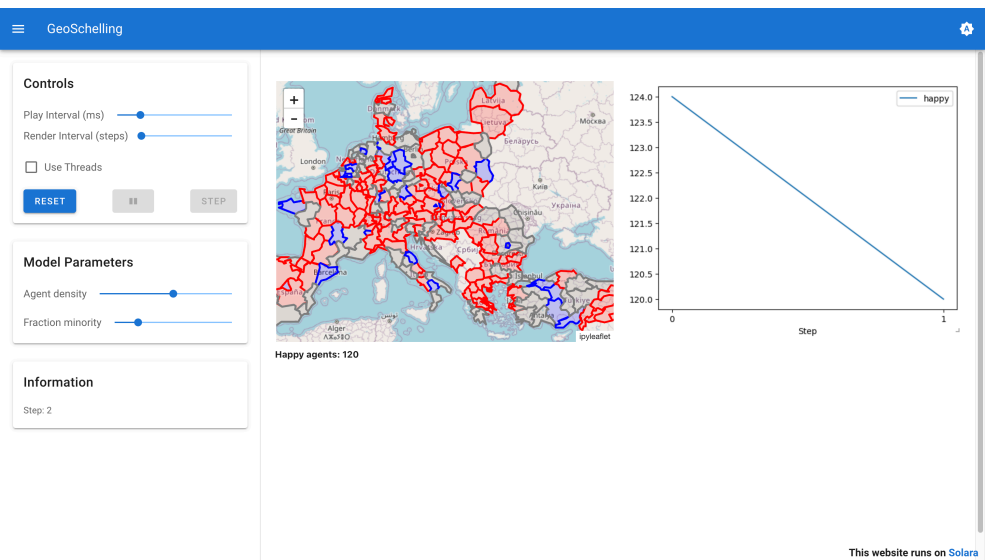
Example: Road Network



Agents and Networks Model

- **GeoSpace:** multiple vector layers, including buildings, lakes, and a road network. The road network is constructed from polyline data.
- **GeoAgent:** commuters.
- Buildings are randomly assigned to agents as their home and workplaces.
- Agents' commute routes can be found as the shortest path between entrances of their home and workplaces. Their movements are constrained on the road network.

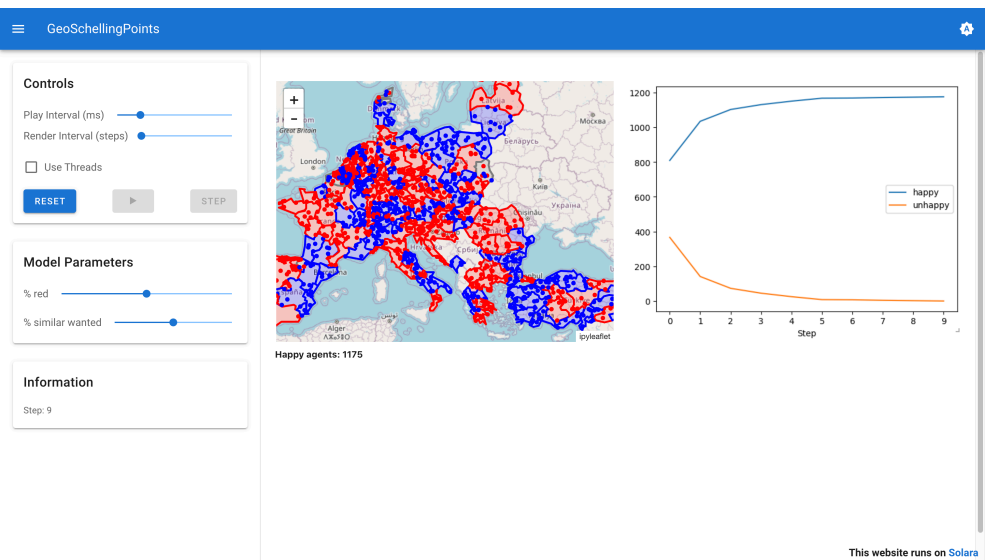
Example: Ploygons



GeoSchelling Model

- **GeoSpace:** only the agent layer containing GeoAgents.
- **GeoAgents:** the Level 2 European Nomenclature of Territorial Units for Statistics (NUTS-2) regions.
- During the running of the model, a polygon queries the colors of the surrounding polygon and if the ratio falls below a certain threshold (e.g., 40% of the same color), the agent moves to an uncolored polygon.

Example: Points & Ploygons



GeoSchelling Points Model

- **GeoSpace:** only the agent layer containing GeoAgents.
- **GeoAgents:**
  - NUTS-2 regions.
  - People residing in NUTS-2 regions.
- Each person resides in a randomly assigned region and checks the color ratio of its region against a pre-defined “happiness” threshold at every time step.
- If the ratio falls below a certain threshold (e.g., 40%), the agent is found to be “unhappy”, and randomly moves to another region.

GSoC Idea (carried over): Unifying Geospatial Support in Mesa-Geo

submitted last year.

- Source: Mesa Wiki: xiv - Google Summer of Code 2025

consistency, and make spatial modeling more accessible.

Motivation

Mesa and Mesa-geo have evolved separately, leading to duplicate implementations and compatibility challenges. As Mesa adopts new space abstractions like the experimental cell space and reimplements continuous space, there's an opportunity to unify spatial modeling in Mesa. This would simplify maintenance, ensure consistent APIs, and make GIS features a first-class citizen in Mesa.

Historical Context

Mesa's spatial modeling has evolved significantly. The current grid system is being replaced by the experimental cell space system, which provides more flexibility and better performance.

Mesa-geo was originally developed as a separate package to add GIS capabilities to Mesa. Recent discussions about moving to a monorepo and the development of Mesa's new conceptual model of Space suggest the time is right for integration.

Overall Goal

Create a unified spatial modeling framework in Mesa that handles both regular and geospatial spaces through a consistent API, while maintaining full GIS functionality.

Expected Outcomes

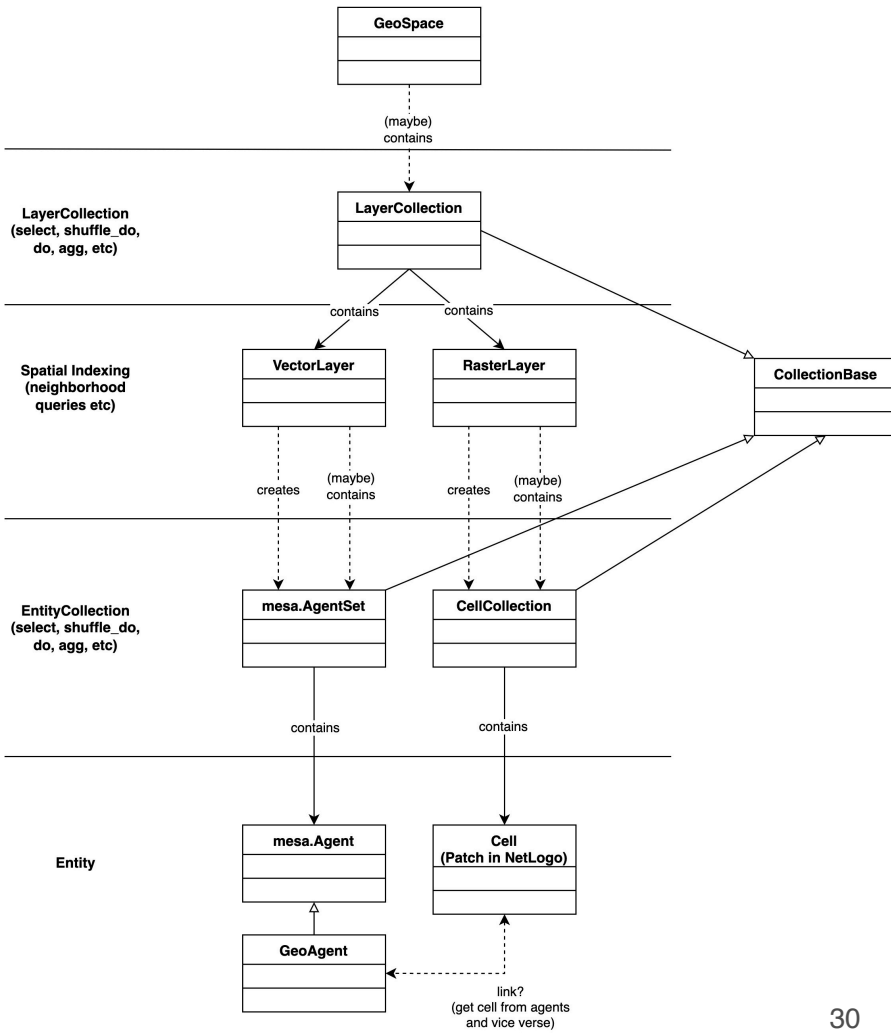
Core Features:

- Integrate Mesa-geo's core functionality (`GeoSpace` , `GeoAgent`) into Mesa as `mesa.geo` module
- Adapt Mesa-geo to use Mesa's new cell space system as its foundation
- Integration of GIS coordinate systems and transformations into Mesa's space framework
- Unified property layer system that works across all space types (addressing #2431)
- Migration path for existing Mesa-geo users

{.fit-figure .fit-figure-sm}

Towards Mesa-Geo v1.0

- Design sketch from last year's training program
- Discussion comment (context + original figure): link



Homework

Read and try the **Mesa-Geo Introductory Tutorial:**

https://mesa-geo.readthedocs.io/stable/tutorials/intro_tutorial.html

Goal: By the end, you should understand how the tutorial uses:

- GeoAgent & GeoSpace
- vector data loading (GeoJSON)
- the visualization component

> ⓘ **While you work through it, please share feedback if anything could be improved**
>
> - unclear wording / missing context, confusing setup steps (install, data download, versions)
> - broken links, typos, or code that doesn't run as written
> - suggestions to make the tutorial more beginner-friendly (what to explain earlier, what to move later)
>
> You can post feedback in our group chat, or open an issue / discussion in the Mesa-Geo repo.

Resources

You can explore Mesa-Geo further via the following links:

- **Source code:** https://github.com/mesa/mesa-geo
- **Documentation & examples:** https://mesa-geo.readthedocs.io
- **Example models repository:** https://github.com/mesa/mesa-examples/tree/main/gis
- **Join the conversation:**
  - Matrix chat room: https://matrix.to/#/#mesa-geo:matrix.org
  - Bi-weekly dev meeting: https://github.com/mesa/mesa/discussions
  - Discussion board: https://github.com/mesa/mesa-geo/discussions