Today

**You said**

Hey

**Copilot said**

Hey Jagdev 👋 Always a pleasure to hear from you. What’s on your mind this evening—geometry diagrams, transcript flows, or something totally unexpected? I’m ready to dive in wherever you want to go.

Edit in a page

**You said**

Give me a few tools and techniques to draw this shape that I can use in streamlit and vercel

**Copilot said**

Absolutely, Jagdev. That circular protractor with directional labels and angle visualization is a great candidate for modular rendering. Here are a few tools and techniques you can use to draw and deploy it cleanly in **Streamlit** and **Vercel**, with precision and interactivity:

**🧰 Recommended Tools**

| Tool | Purpose | Notes |
| --- | --- | --- |
| **SVG.js** or **D3.js** | Precise vector drawing | Use via streamlit.components.v1.html() for embedding |
| **Matplotlib + SVG Export** | Geometry + angle rendering | Export as SVG and render in Streamlit |
| **Plotly** | Interactive polar plots | Great for quadrant labeling and angle arcs |
| **Streamlit Canvas** | Freehand or structured drawing | Good for prototyping, less precise for protractors |
| **Custom SVG with Jinja2** | Template-based rendering | Ideal for modular, reusable diagrams |

**✍️ Techniques to Draw the Protractor + Angle**

**1. SVG-Based Modular Drawing**

* Define the protractor as an SVG <circle> with degree ticks using <line> and <text> elements.
* Use <path> or <arc> for the pink angle arc.
* Label quadrants with <text> at cardinal positions (N, E, S, W).
* Dynamically update angle lines using Python variables passed into the template.

**2. Polar Plot with Plotly**

python

Copy

import plotly.graph\_objects as go

fig = go.Figure()

# Protractor circle

fig.add\_trace(go.Scatterpolar(

r=[1]\*361,

theta=list(range(361)),

mode='lines',

line=dict(color='lightgray'),

showlegend=False

))

# Angle lines

fig.add\_trace(go.Scatterpolar(r=[0, 1], theta=[40, 40], mode='lines', line=dict(color='blue')))

fig.add\_trace(go.Scatterpolar(r=[0, 1], theta=[70, 70], mode='lines', line=dict(color='blue')))

# Angle arc (approximate)

fig.add\_trace(go.Scatterpolar(

r=[0.9]\*31,

theta=list(range(40, 71)),

mode='lines',

line=dict(color='pink', width=4)

))

fig.update\_layout(

polar=dict(radialaxis=dict(visible=False)),

showlegend=False

)

* Deploy in Streamlit with st.plotly\_chart(fig).

**3. SVG Template with Jinja2**

* Create an SVG template with placeholders for angle values.
* Use Jinja2 to inject dynamic values (e.g., angle start/end).
* Render in Streamlit using components.html().

**🚀 Deployment Tips for Vercel**

* Use **Streamlit static export** (streamlit hello → export HTML) or wrap your app with **FastAPI** + render\_template() if using SVG templates.
* For Plotly or Matplotlib, ensure all assets are bundled or served via CDN.
* Keep your drawing logic modular—e.g., draw\_protractor(angle1, angle2)—so you can reuse it across apps.

Would you like a starter template for one of these approaches? I can scaffold a modular SVG or Plotly version tailored to your transcript app’s style.

Edit in a page

**You said**

give me all the directory structure layout, files and folders and libraries required in requirements.txt for developing this as VS Code Project

**Copilot said**

Perfect, Jagdev. Here's a clean, modular layout for your **VS Code project** that renders a circular protractor with angle visualization using **Streamlit**, **Plotly**, and optional **SVG templating**. It’s designed for clarity, scalability, and easy deployment to **Vercel** or similar platforms.

**🗂️ Project Directory Structure**

Code

Copy

protractor\_app/

├── .streamlit/

│ └── config.toml # Optional: set theme or port

├── assets/

│ └── protractor\_base.svg # Optional: SVG template for Jinja2

├── components/

│ └── svg\_renderer.py # Renders SVG with dynamic angles

├── pages/

│ └── angle\_demo.py # Optional: multi-page Streamlit app

├── utils/

│ └── angle\_math.py # Angle calculations, conversions

├── app.py # Main Streamlit entry point

├── requirements.txt # All dependencies

├── README.md # Project overview

└── vercel.json # Vercel deployment config

**📦 requirements.txt**

txt

Copy

streamlit>=1.30.0

plotly>=5.20.0

jinja2>=3.1.2

numpy>=1.26.0

pandas>=2.2.0

Optional additions:

* svgwrite if you want to generate SVGs programmatically.
* streamlit-extras for UI enhancements.
* streamlit.components.v1 is built-in, no need to list separately.

**🧠 File Breakdown**

**app.py**

Main entry point. Loads angle values, renders Plotly chart or SVG.

python

Copy

import streamlit as st

from components.svg\_renderer import render\_svg

from utils.angle\_math import compute\_arc

st.title("Circular Protractor Viewer")

angle1 = st.slider("Start Angle", 0, 360, 40)

angle2 = st.slider("End Angle", 0, 360, 70)

st.subheader("Plotly Visualization")

from plotly import graph\_objects as go

# (Insert Plotly polar chart logic here)

st.subheader("SVG Version")

svg\_code = render\_svg(angle1, angle2)

st.components.v1.html(svg\_code, height=400)

**components/svg\_renderer.py**

Uses Jinja2 to inject dynamic angles into an SVG template.

python

Copy

from jinja2 import Environment, FileSystemLoader

def render\_svg(angle1, angle2):

env = Environment(loader=FileSystemLoader("assets"))

template = env.get\_template("protractor\_base.svg")

return template.render(angle1=angle1, angle2=angle2)

**utils/angle\_math.py**

Handles angle normalization, arc length, quadrant logic.

python

Copy

import numpy as np

def compute\_arc(a1, a2):

return (a2 - a1) % 360

**vercel.json**

For Vercel deployment (if using streamlit-viz or FastAPI wrapper):

json

Copy

{

"builds": [

{ "src": "app.py", "use": "@vercel/python" }

],

"routes": [

{ "src": "/(.\*)", "dest": "app.py" }

]

}

**🧪 Bonus: Dev Tips**

* Use **VS Code Tasks** to run streamlit run app.py with one click.
* Add .vscode/settings.json to auto-format with Black or Ruff.
* Modularize quadrant labels and arc rendering for reuse in other geometry apps.

Want me to scaffold the actual protractor\_base.svg template or build out the Plotly chart logic next? I can tailor it to your transcript app’s visual style or pedagogical goals.

Edit in a page

**You said**

code .gitignore for this VS Project

**Copilot said**

Here’s a clean .gitignore tailored for your **Streamlit + Plotly + SVG-based geometry project** in VS Code. It keeps your repo tidy by ignoring unnecessary files from Python, Streamlit, VS Code, and deployment artifacts:

gitignore

Copy

# Python

\_\_pycache\_\_/

\*.py[cod]

\*.egg

\*.egg-info/

dist/

build/

\*.log

# Virtual environments

venv/

.env/

.envrc

\*.env

# Streamlit

.streamlit/secrets.toml

.streamlit/config.toml

# VS Code

.vscode/

\*.code-workspace

# OS-specific

.DS\_Store

Thumbs.db

# Jupyter (if used for prototyping)

.ipynb\_checkpoints/

# Deployment

vercel/

.vercel/

\*.sqlite3

\*.db

# SVG or image cache (if generated dynamically)

\*.svg

\*.png

\*.jpg

\*.jpeg

\*.gif

# Misc

\*.bak

\*.tmp

You can tweak this further if you add Docker, poetry, or other tooling. Want me to scaffold a README.md next with setup and usage instructions?

Edit in a page

**You said**

What should be written for .streamlit/config.toml

**Copilot said**

Great question, Jagdev. The .streamlit/config.toml file lets you customize how your Streamlit app behaves—especially useful for setting default themes, disabling telemetry, or configuring server behavior during local dev or deployment.

Here’s a solid starter config tailored for your geometry app:

toml

Copy

[server]

headless = true

enableCORS = false

port = 8501

runOnSave = true

[theme]

base = "light"

primaryColor = "#1f77b4"

backgroundColor = "#ffffff"

secondaryBackgroundColor = "#f0f2f6"

textColor = "#262730"

font = "sans serif"

[browser]

gatherUsageStats = false

**🔍 Breakdown:**

* headless = true: Ensures compatibility with cloud platforms like Vercel.
* enableCORS = false: Avoids cross-origin issues when embedding or deploying.
* runOnSave = true: Auto-refreshes the app when you save changes—great for iterative design.
* theme: Matches a clean, educational aesthetic. You can tweak colors to match your transcript app’s branding.
* gatherUsageStats = false: Disables telemetry for privacy-conscious deployments.

Want me to help you define a custom color palette or typography that matches your SVG diagrams or student-facing UI? I can even simulate how it’ll look.

Edit in a page

**You said**

jinja2.exceptions.TemplateNotFound: 'protractor\_base.svg' not found in search path: 'assets' Traceback: File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\protractor\_app\app.py", line 15, in <module> svg\_code = render\_svg(angle1, angle2) ^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\protractor\_app\components\svg\_renderer.py", line 5, in render\_svg template = env.get\_template("protractor\_base.svg") ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\venv\Lib\site-packages\jinja2\environment.py", line 1016, in get\_template return self.\_load\_template(name, globals) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\venv\Lib\site-packages\jinja2\environment.py", line 975, in \_load\_template template = self.loader.load(self, name, self.make\_globals(globals)) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\venv\Lib\site-packages\jinja2\loaders.py", line 126, in load source, filename, uptodate = self.get\_source(environment, name) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\jagdevsinghdosanjh.RHINO\Math-Quiz-6th-to-10th\venv\Lib\site-packages\jinja2\loaders.py", line 209, in get\_source raise TemplateNotFound(

**Copilot said**

Thanks for sharing the traceback, Jagdev—this is a classic TemplateNotFound issue with Jinja2, and it’s usually caused by one of these:

**🔍 Root Cause Checklist**

1. **Missing File** Ensure protractor\_base.svg actually exists inside the assets/ folder at the root of your project:

Code

Copy

protractor\_app/

├── assets/

│ └── protractor\_base.svg ✅ must be here

1. **Incorrect Path in Loader** If your svg\_renderer.py is inside components/, and you're running app.py from the root (protractor\_app/), then this line:

python

Copy

env = Environment(loader=FileSystemLoader("assets"))

is correct **only if** you're executing from the root. If you're running from inside components/, you may need:

python

Copy

env = Environment(loader=FileSystemLoader("../assets"))

1. **Case Sensitivity** File name must match exactly: protractor\_base.svg not Protractor\_base.svg or protractorBase.svg.
2. **Working Directory Confusion** If you're running the script from a different folder (e.g., VS Code terminal inside components/), Jinja2 may not resolve the path correctly. Always run from the project root:

bash

Copy

cd protractor\_app

streamlit run app.py

**✅ Quick Fix Steps**

1. Confirm the file exists:

bash

Copy

ls assets/protractor\_base.svg

1. If it’s missing, create a placeholder:

xml

Copy

<!-- assets/protractor\_base.svg -->

<svg width="400" height="400" xmlns="http://www.w3.org/2000/svg">

<circle cx="200" cy="200" r="180" stroke="black" fill="none"/>

<!-- Add dynamic angle lines via Jinja2 -->

<line x1="200" y1="200" x2="{{ x1 }}" y2="{{ y1 }}" stroke="blue"/>

<line x1="200" y1="200" x2="{{ x2 }}" y2="{{ y2 }}" stroke="blue"/>

<text x="190" y="20" font-size="16">N</text>

<text x="370" y="210" font-size="16">E</text>

<text x="190" y="390" font-size="16">S</text>

<text x="10" y="210" font-size="16">W</text>

</svg>

1. Restart your app from the root directory:

bash

Copy

streamlit run app.py

If you’d like, I can help you generate the full SVG template with dynamic angle rendering logic using Jinja2 variables like angle1, angle2, and even compute the arc path. Just say the word.

Edit in a page

**You said**

I got this as in snapshot but required one is

**Copilot said**

Got it, Jagdev—it looks like the current snapshot shows the **Plotly visualization working**, but the **SVG version isn't rendering the full protractor layout** the way you intended. You're aiming for something more like the earlier image: a circular protractor with full 360° markings, quadrant labels (N, E, S, W), and a clearly highlighted angle arc between two blue lines.

Let’s get you there. Here's what I suggest:

**✅ What You Likely Need in protractor\_base.svg**

Your SVG template should include:

* A full circle with degree ticks (0° to 360°)
* Cardinal direction labels (N, E, S, W)
* Two dynamic lines for angle1 and angle2 using Jinja2 variables
* A pink arc between those angles
* Optional: magnified scale region or quadrant shading

**🛠️ Fix Strategy**

1. **Update your** assets/protractor\_base.svg with a full scaffold that uses Jinja2 variables like angle1, angle2, and computes line endpoints using basic trigonometry.
2. **Modify** svg\_renderer.py to compute x1, y1 and x2, y2 from angles and inject them into the template.

**✨ Want Me to Build It?**

I can generate a complete protractor\_base.svg template with placeholders and the Python logic to compute the coordinates for the angle lines and arc. It’ll be modular, reusable, and visually aligned with your educational goals.

Just say the word and I’ll scaffold the full SVG + renderer logic for you.

Edit in a page

**You said**

Yes

**Copilot said**

Awesome, Jagdev. Let’s build a fully functional, modular SVG template and renderer that dynamically draws your circular protractor with angle lines and arc using Jinja2 and Streamlit.

**🧩 Step 1: assets/protractor\_base.svg (Template)**

Here’s a simplified version of the SVG template using Jinja2 placeholders. It draws:

* A circular protractor
* Cardinal direction labels
* Two blue angle lines
* A pink arc between them

xml

Copy

<svg width="400" height="400" viewBox="0 0 400 400" xmlns="http://www.w3.org/2000/svg">

<!-- Outer circle -->

<circle cx="200" cy="200" r="180" stroke="black" stroke-width="2" fill="none"/>

<!-- Cardinal directions -->

<text x="190" y="20" font-size="16">N</text>

<text x="370" y="210" font-size="16">E</text>

<text x="190" y="390" font-size="16">S</text>

<text x="10" y="210" font-size="16">W</text>

<!-- Angle lines -->

<line x1="200" y1="200" x2="{{ x1 }}" y2="{{ y1 }}" stroke="blue" stroke-width="2"/>

<line x1="200" y1="200" x2="{{ x2 }}" y2="{{ y2 }}" stroke="blue" stroke-width="2"/>

<!-- Arc path -->

<path d="{{ arc\_path }}" fill="none" stroke="pink" stroke-width="4"/>

</svg>

**🧠 Step 2: components/svg\_renderer.py**

This computes the coordinates and arc path using basic trigonometry and SVG arc syntax:

python

Copy

import math

from jinja2 import Environment, FileSystemLoader

def polar\_to\_cartesian(center\_x, center\_y, radius, angle\_deg):

angle\_rad = math.radians(angle\_deg)

x = center\_x + radius \* math.cos(angle\_rad)

y = center\_y - radius \* math.sin(angle\_rad)

return x, y

def describe\_arc(x, y, radius, start\_angle, end\_angle):

start = polar\_to\_cartesian(x, y, radius, start\_angle)

end = polar\_to\_cartesian(x, y, radius, end\_angle)

large\_arc\_flag = 1 if (end\_angle - start\_angle) % 360 > 180 else 0

return f"M {start[0]} {start[1]} A {radius} {radius} 0 {large\_arc\_flag} 0 {end[0]} {end[1]}"

def render\_svg(angle1, angle2):

center\_x, center\_y, radius = 200, 200, 180

x1, y1 = polar\_to\_cartesian(center\_x, center\_y, radius, angle1)

x2, y2 = polar\_to\_cartesian(center\_x, center\_y, radius, angle2)

arc\_path = describe\_arc(center\_x, center\_y, radius - 10, angle1, angle2)

env = Environment(loader=FileSystemLoader("assets"))

template = env.get\_template("protractor\_base.svg")

return template.render(x1=x1, y1=y1, x2=x2, y2=y2, arc\_path=arc\_path)

**🧪 Step 3: app.py Integration**

python

Copy

import streamlit as st

from components.svg\_renderer import render\_svg

st.title("Circular Protractor Viewer")

angle1 = st.slider("Start Angle", 0, 360, 40)

angle2 = st.slider("End Angle", 0, 360, 70)

st.subheader("SVG Version")

svg\_code = render\_svg(angle1, angle2)

st.components.v1.html(svg\_code, height=420)

This setup gives you a clean, interactive protractor that updates in real time. You can later extend it with tick marks, quadrant shading, or magnified views.

Want help adding degree ticks or quadrant overlays next? I can modularize those too.

Edit in a page

**You said**

Update this code : import streamlit as st from utils.angle\_math import compute\_arc # noqa import streamlit as st # noqa from components.svg\_renderer import render\_svg from plotly import graph\_objects as go # noqa st.title("Circular Protractor Viewer") angle1 = st.slider("Start Angle", 0, 360, 40) angle2 = st.slider("End Angle", 0, 360, 70) st.subheader("SVG Version") svg\_code = render\_svg(angle1, angle2) st.components.v1.html(svg\_code, height=420) # st.title("Circular Protractor Viewer") # angle1 = st.slider("Start Angle", 0, 360, 40) # angle2 = st.slider("End Angle", 0, 360, 70) st.subheader("Plotly Visualization") # (Insert Plotly polar chart logic here) st.subheader("SVG Version") svg\_code = render\_svg(angle1, angle2) st.components.v1.html(svg\_code, height=400)

**Copilot said**

Here’s a clean, updated version of your code, Jagdev—refactored for clarity, modularity, and to avoid duplication. I’ve removed redundant imports, consolidated the layout, and scaffolded the missing Plotly visualization logic so both views (SVG and Plotly) work side by side:

**✅ Updated app.py**

python

Copy

import streamlit as st

from components.svg\_renderer import render\_svg

from plotly import graph\_objects as go

st.set\_page\_config(page\_title="Circular Protractor Viewer", layout="centered")

st.title("🎯 Circular Protractor Viewer")

# Angle inputs

angle1 = st.slider("Start Angle", 0, 360, 40)

angle2 = st.slider("End Angle", 0, 360, 70)

# SVG Visualization

st.subheader("🖼️ SVG Version")

svg\_code = render\_svg(angle1, angle2)

st.components.v1.html(svg\_code, height=420)

# Plotly Visualization

st.subheader("📊 Plotly Visualization")

fig = go.Figure()

# Outer circle

fig.add\_trace(go.Scatterpolar(

r=[1]\*361,

theta=list(range(361)),

mode='lines',

line=dict(color='lightgray'),

showlegend=False

))

# Angle lines

fig.add\_trace(go.Scatterpolar(

r=[0, 1],

theta=[angle1, angle1],

mode='lines',

line=dict(color='blue', width=2),

name=f"{angle1}°"

))

fig.add\_trace(go.Scatterpolar(

r=[0, 1],

theta=[angle2, angle2],

mode='lines',

line=dict(color='blue', width=2),

name=f"{angle2}°"

))

# Arc between angles

arc\_range = list(range(angle1, angle2 + 1)) if angle1 < angle2 else list(range(angle1, 360)) + list(range(0, angle2 + 1))

fig.add\_trace(go.Scatterpolar(

r=[0.9]\*len(arc\_range),

theta=arc\_range,

mode='lines',

line=dict(color='pink', width=4),

name="Angle Arc"

))

fig.update\_layout(

polar=dict(radialaxis=dict(visible=False)),

showlegend=False,

margin=dict(t=20, b=20, l=20, r=20),

height=420

)

st.plotly\_chart(fig, use\_container\_width=True)

**🔧 Improvements Made**

* Removed duplicate import streamlit as st
* Added set\_page\_config for better layout control
* Consolidated angle sliders and removed commented-out code
* Implemented full Plotly logic with arc rendering and angle lines
* Unified height and styling across both visualizations

Let me know if you want quadrant overlays, tick marks, or dynamic angle labels next. We can modularize those into utils/ or extend the SVG template for even richer geometry interactions.