Tessellations and Sweeping Nets: Advancing the Calculus of Geometric Logic

Parker Emmerson

December 2023

1 Introduction

This paper focuses on the optimal arrangement of reflecting points for efficient ray tracing given limited sweep time. We examine spatial configurations, employing our core concept of a sweeping subnet and defining a causal barrier to capture constraints imposed by time.

We will also discuss the influence of these constructions on the design of an algorithm for approximating optimal tessellations.

I have provided code for each of the graphs, as the mathematics is demonstrated unequivocally by their implementation. The reader can test out the reality of this system by visualizing the graphs themselves using Python in a suitable environment like Google Colaboratory.

2 Fundamentals

2.1 Sweeping Subnet

A sweeping subnet refers to the set of reachable points on a surface from a light source within a time constraint. To formalize:

$$\|\vec{r}_i - \vec{x}_i\| \leq 2\|\vec{n}(X_i)\| < 2\xi. \tag{1}$$

This equation establishes the geometrical constraints required to capture the notion of sweeping efficiency rigorously.

2.2 Causal Barrier

The causal barrier is the spatial limit reachable by rays within a defined temporal boundary:

$$r_{\text{barrier}}(t) = \max_{x,y \in C} G(x,y) \cdot P(y,t). \tag{2}$$

It delineates the boundary of influence for any point within our geometric configuration.

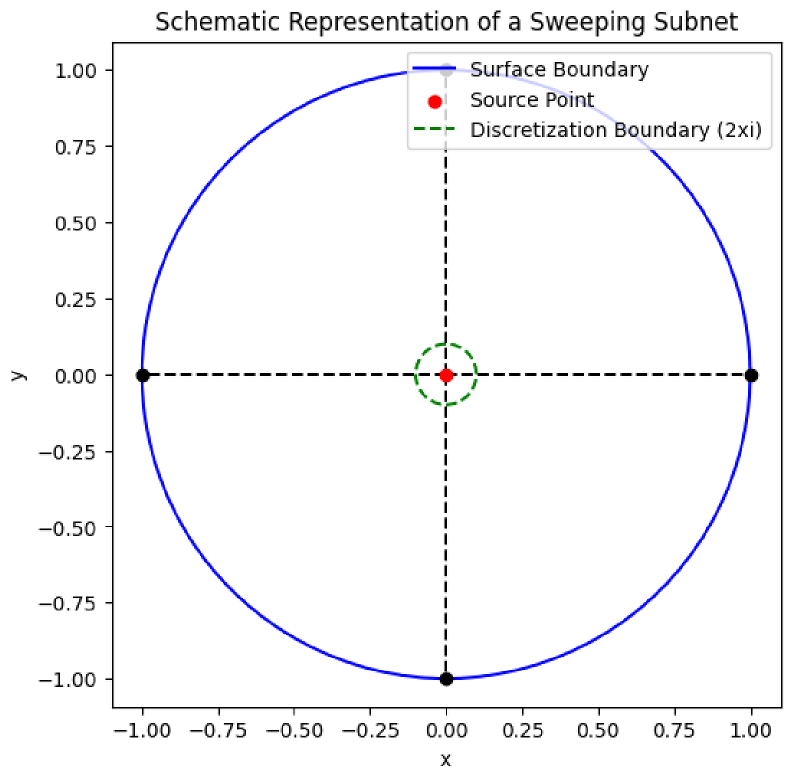


Figure 1: Schematic representation of a sweeping subnet.

2.2.1 Causal Barrier Dynamics

A pivotal factor in tracing rays within a limited timeframe is the concept of a causal barrier. This barrier represents the set of points that are unreachable by light within a given period, restricted by past events and influences. The causal barrier is a manifest constraint shaped by the maximum sweep time τ and the network of reflecting points across the surface Ω.

The causal barrier's radius, $r_{\text{barrier}}(t)$, quantifies the spatiotemporal limit of causal influence for light propagating in the medium, and is expressed as:

$$r_{\text{barrier}}(t) = \max_{(x,y)\in C} G(x,y) \cdot P(y,t), \tag{3}$$

where $G(x,y)$ characterizes a geometric factor dependent on the spatial coordinates of the medium, and $P(y,t)$ signifies the probability of light or causal influence reaching point y at time t.

Notably, the causal barrier encompasses not just the physical impediments to light's movement but also integrates the historical dependencies influencing its progress. Factors such as prior ray paths and reflecting surface orientations are embodied within this structure. For a discretized model:

$$k_{i_1 \dots i_n} := \lim_{x \to p_0^i} \mathfrak{s}(x), \tag{4}$$

which signifies the link between the sites' tessellation patches and the causal barrier. Here, the function $\mathfrak{s}(x)$ describes the geometric state at each point x along a path p_0^i, converging on the reflective elements' configurations. The optimized tessellation, $q_{i_1 \dots i_m}$, thus not only examines spatial coverage but also adapts to temporal dimension constraints imposed by the causal barrier.

Figure 2 illustrates the causal barrier's structure, highlighting its variation with different discretization parameters and tessellation configurations. As the discretization granularity refines, the causal barrier grows dynamically, underscoring the coupling between discretization strategies and the operational envelope of the tessellated reflective elements. In essence, it becomes a data structure encoding the system's global dynamics, encompassing discretized ray paths, reflective interactions, and time-sensitivity constraints. This characterization permits an informed assessment of tessellation costs and provides insight into the computational complexity of potential paving schemes across the surface Ω.

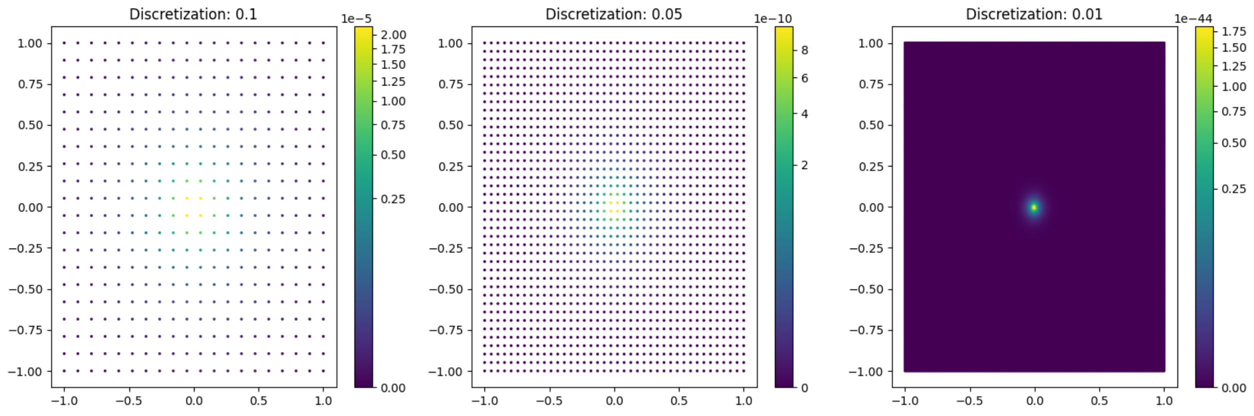


Figure 2: Visualization of Causal Barrier with varying discretization parameters.

```
Code for visualizing the Causal Barrier

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import PowerNorm

# Function to simulate the causal barrier
def causal_barrier(t, xi):
    # Represents a simulation of barrier values in the 2D space
    # For simplicity, returning a placeholder array mimicking barrier values
    X, Y = np.meshgrid(np.linspace(-1, 1, int(2/xi)), np.linspace(-1, 1, int(2/xi)))
    Z = np.exp(-((X**2 + Y**2)**0.5 + t) / xi)  # Exponential decay as an example
    return X, Y, Z
```

```
# Define time variable and discretization parameters
t = 1  # Fixed point in time
discretization_params = [0.1, 0.05, 0.01]  # Varying discretization parameters

# Create figure
fig, axs = plt.subplots(1, len(discretization_params), figsize=(15, 5))

# Loop through varying discretization parameters and plot
for ax, xi in zip(axs, discretization_params):
    X, Y, Z = causal_barrier(t, xi)
    # Using scatter plot for performance and visualization of discretization points
    scatter = ax.scatter(X, Y, c=Z, cmap='viridis', norm=PowerNorm(0.3), s=2)
    ax.set_title(f'Discretization: {xi}')
    plt.colorbar(scatter, ax=ax)

plt.tight_layout()
plt.show()
```

Further complicating the causal barrier's role is its influence on evolving system constraints. It guides how computational processes unfold within the discretized model, calculating emerging constraints through time-evolving tessellations. The causal barrier, therefore, has a symbiotic relationship with the tessellation process: each contributing to and shaped by the logical calculus of ray propagation.

Let $A = \bigcup_i A_i$ and W_i, we define *larger* residue sets, R_i, where each subset's union within W_i is A_i, ensuring their existence due to the infinite cardinality of the Cartesian product space.

In summary, the causal barrier serves as a critical component in the tessellation problem, reflecting the complexities of both spatial arrangement and temporal evolution. It encapsulates the ultimate bounds of light's propagation and provides a comprehensive framework for calculating ray trajectory efficiencies across the discretized landscape of Ω.

3 Optimal Tessellation Framework

We now introduce the formalism for our tessellation strategy starting with the radius of spheres $\mathcal{S}_r$.

This section demonstrates the method for approximating surfacing singularities of saddle maps using a sweeping net. The method involves constructing a densified sweeping subnet for each individual vertex of the saddle map, and then combining each subnet to create a complete approximation of the singularities. The authors also define two functions f_1 and f_2, which are used to calculate the charge density for each subnet. The resulting densified sweeping subnet closely approximates the surfacing saddle map near a circular region.

$$\{\langle \partial\theta \times \vec{r}_\infty \rangle \cap \langle \partial\vec{x} \times \theta_\infty \rangle\} \to \left\{ (A_r \oplus B_r) \cap \mathcal{S}_r^+ \right\}. \tag{5}$$

This is the implication of a calculus structure combining spatio-temporally to form a oneness denoted $1_{E_{\{\langle \partial\theta \times \vec{r}_\infty \rangle \cap \langle \partial\vec{x} \times \theta_\infty \rangle\} \to \left\{ (A_r \oplus B_r) \cap \mathcal{S}_r^+ \right\}}}$.

Here $\mathcal{S}_r^+$ is the right half of the unit circle, defined as

$$\mathcal{S}_r^+ \left\{(x,y) \in \mathbb{R}^2 \,|\, x^2+y^2=r^2, x \geq 0\right\}, \tag{6}$$

and A_r, B_r are specified as follows

$$A_r \left\{(\tilde{x},\tilde{y})|\tilde{x} \geq 0, \tilde{y} \geq 0, \tilde{x}^2+\tilde{y}^2=1, \arcsin\tilde{x} \geq f_1(\arcsin(r^{-1}\tilde{x}))\right\}, \tag{7}$$
$$B_r \left\{(\tilde{x},\tilde{y})|\tilde{x} \geq 0, \tilde{y} \geq 0, \tilde{x}^2+\tilde{y}^2=1, \arcsin\tilde{y} \geq f_2(\arcsin(r^{-1}\tilde{y}))\right\}, \tag{8}$$
$$\tag{9}$$

In the above, $\oplus$ indicates the direct sum of two sets and $r_+ = r$. $\vec{x}$ is a curve where the slope of tangent line is greater than the vertex in the line function (See Fig.(**??**)b upper line). In the same way, $\partial\vec{x}$ is the vertex set of $\vec{x}$ (single point set). θ_∞ is a direct sum of line $l_{mn} := \{(x,y) \in \mathbb{R}^2|x+ry=n\}$ (n is constant) and the line with infinite slope.

We define $f_1, f_2 : [0,\pi/2] \to [0,\pi/2]$ as follows

$$\frac{\partial \arcsin(\sin\theta)}{\partial\theta} = \ldots\frac{1}{\sqrt{1-\sin^2\theta}}\left(\int_0^1 \frac{\mathrm{d}}{\mathrm{d}\theta}\sin\theta\mathrm{d}s\right) = \ldots\frac{\cos\theta}{\sqrt{1-\sin^2\theta}}$$

When we take $\theta = \frac{\pi}{2}$, $f_1(0) = f_2(0) = 0$. It implies that f_1 and f_2 continuously connect with straight line to positively going. The ω calculates as follows

$$\omega\Big|_{\mathcal{S}_r^+} = \int_0^{\frac{\pi}{2}} \left\{\left(\mathcal{K}^{-1}f_i'(s)\partial s\right) \times \left(\tilde{x}(s,l)-\tilde{x}(0,l)\right)\right\},\ i=\{1,2\} \tag{10}$$

where $\mathcal{K}$ and charge density ∂s are constant and expressed as

$$\tilde{x}(s,l)\tilde{x}^{(0)} + r\sin s\tilde{Y}(l), \tag{11}$$
$$\tilde{x}(0,l)\tilde{x}^{(0)} + r\tilde{Y}(l), \tag{12}$$

respectively. Here $\tilde{x}^{(0)} = (1,1)^{\mathrm{t}}$, and $\tilde{Y}(l) = (\cos l, \sin l)^{\mathrm{t}}$ normalize. Consequently, the net (5) approximates the surfacing saddle map around the right circle, when $r > 0$ is sufficiently small(Since only around a right circle), approximately satisfying charge density of sweeping generic singular saddle case around the right circle.

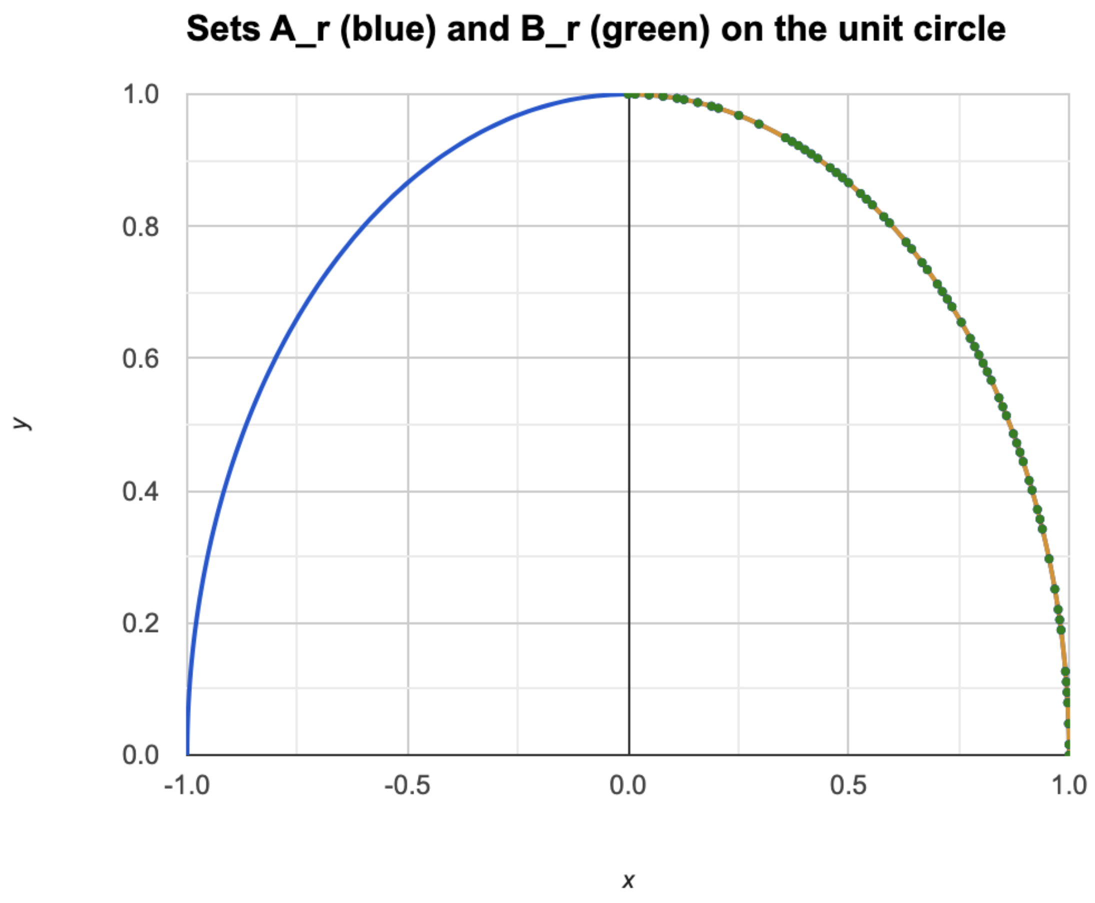
4 Graphing the System

Graphing this system yields two different graphs depending on whether you use Python or Mathematica.

4.1 Python Code

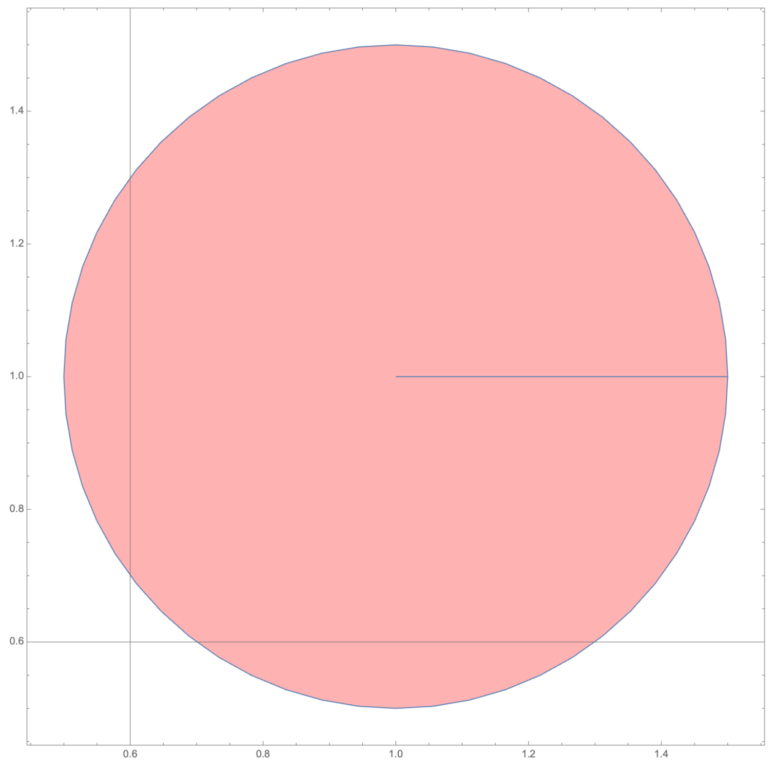
```
    import matplotlib.pyplot as plt
import numpy as np

# Define the functions f1 and f2
def f1(theta):
    return np.arcsin(np.sin(theta)) + np.pi/2 * (1 - np.pi / (2 * theta))

def f2(theta):
    return np.arcsin(np.cos(theta)) + np.pi/2 * (1 - np.pi / (2 * theta))

# Define the unit circle and right half circle
theta = np.linspace(0, np.pi, 200)
x_unit = np.cos(theta)
y_unit = np.sin(theta)
x_right = x_unit[theta <= np.pi/2]
y_right = y_unit[theta <= np.pi/2]

# Define the sets A_r and B_r
r = 0.5  # Set the radius
A_r = []
B_r = []
for theta in np.linspace(0, np.pi/2, 100):
    # Convert theta to x and y coordinates on the unit circle
    x = np.cos(theta)
    y = np.sin(theta)

    # Check if (x, y) is in A_r
    if x >= 0 and y >= 0 and x**2 + y**2 == 1 and np.arcsin(x) >= f1(np.arcsin(r * x)):
        A_r.append((x, y))

    # Check if (x, y) is in B_r
    if x >= 0 and y >= 0 and x**2 + y**2 == 1 and np.arcsin(y) >= f2(np.arcsin(r * y)):
        B_r.append((x, y))

# Plot the unit circle, right half circle, sets A_r and B_r
fig, ax = plt.subplots()
ax.plot(x_unit, y_unit, label='Unit circle')
ax.plot(x_right, y_right, label='Right half circle')

for point in A_r:
    ax.plot(point[0], point[1], marker='o', color='b', alpha=0.5)

for point in B_r:
    ax.plot(point[0], point[1], marker='o', color='g', alpha=0.5)
```

```
# Set labels and title
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Sets A_r (blue) and B_r (green) on the unit circle')
ax.legend()

# Show the plot
plt.show()
```



4.2 Mathematica Code

```
(*Define the constants and functions*)
r = 0.5;(*Radius of the \
circle*)K = 1;(*Constant K*)
f1[\[Theta]_] :=
 ArcSin[Sin[\[Theta]]] + \[Pi]/2 (1 - \[Pi]/(2 \[Theta]));
f2[\[Theta]_] :=
  ArcSin[Cos[\[Theta]]] + \[Pi]/2 (1 - \[Pi]/(2 \[Theta]));

x0 = {1, 1};(*Initial point*)
Y[l_] := {Cos[l],
  Sin[l]};(*Normalized vector*)(*Define the sets Ar and Br*)Ar =
 ImplicitRegion[
  x^2 + y^2 == 1 && x >= 0 && y >= 0 &&
   ArcSin[x] >= f1[ArcSin[r^-1 x]], {x, y}];
Br = ImplicitRegion[
   x^2 + y^2 == 1 && x >= 0 && y >= 0 &&
    ArcSin[y] >= f2[ArcSin[r^-1 y]], {x, y}];

(*Visualize the sets*)

RegionPlot[{Ar, Br}, PlotRange -> {{0, 1.2}, {0, 1.2}},
  BoundaryStyle -> {Red, Blue}, PlotLegends -> {"Ar", "Br"}];

(*Define the curves x(s,l) and x(0,l)*)

x[s_, l_] := x0 + r Sin[s] Y[l];
x0l = x0 + r Y[l];

(*Parametric plot of the curves*)
ParametricPlot[{x[s, l], x0l} /.
  l -> t, {s, 0, \[Pi]/2}, {t, 0, 2 \[Pi]},
 PlotStyle -> {{Red, Thick}, {Blue, Dashed}}]
```



$$\begin{aligned}A_r &:= \{\vec{x} \in \partial\Omega \colon \exists \theta \text{ such that } \|\partial\theta \times \vec{r}\| \leq 2\xi, \|\vec{r} - \vec{x}\| < r\},\\ B_r &:= \{\vec{r} \in \partial\Omega \colon \exists \vec{x} \text{ such that } \|\partial\vec{x} \times \theta\| \leq 2\xi, \|\vec{r} - \vec{x}\| < r\},\end{aligned} \tag{13}$$

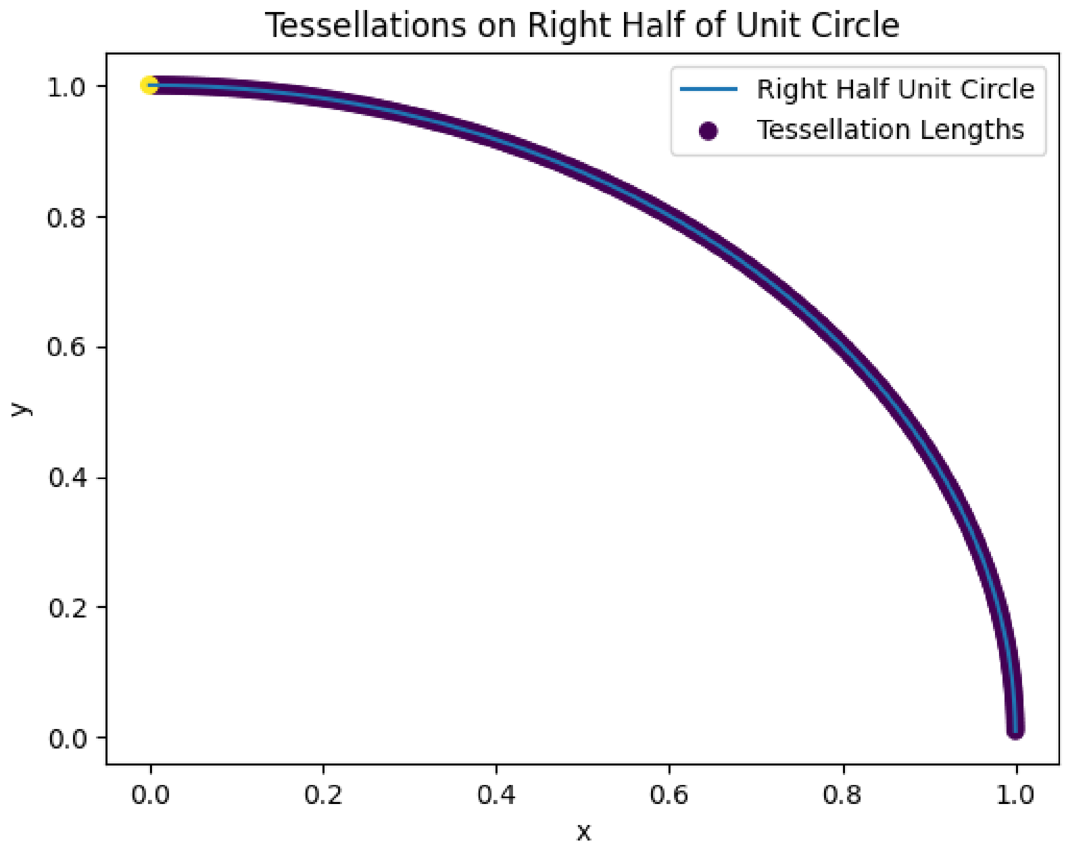
The strategic overlay of A_r and B_r yields a tessellation conducive to an optimal sweeping subnet.

5 Algorithmic Approach

I used the sweeping net concept to generate tessellations along the curve dictated by the form of the notated calculus singularity as above. The tessellations lengths follow the curve of this function:

```
import numpy as np
import matplotlib.pyplot as plt

# Constants and definitions based on LaTeX input and provided data
r = 0.5  # Radius value from data
theta_inf = 2 * np.pi  # Infinity angle

# Function f1 as defined in provided text
def f1(theta):
    return np.arcsin(np.sin(theta)) + (np.pi / 2) * (1 - (np.pi / (2 * theta)))
    if theta != 0 else 0
```

```
# Function f2 as defined in provided text
def f2(theta):
    return np.arcsin(np.cos(theta)) + (np.pi / 2) * (1 - (np.pi / (2 * theta)))
    if theta != 0 else 0

# Placeholder for tessellation related function based on LaTeX interpretation
def tessellation_length(phi, psi, theta):
    if phi != 0:
        return phi / np.cos(theta)
    elif psi != 0:
        return psi / np.sin(theta)
    else:
        return 0

# Define the plotting function for tessellation
def plot_tessellation():
    # Generate theta values for the right half of the circle
    theta_values = np.linspace(0.01, np.pi / 2, 300) # Avoid division by zero

    # Compute tessellation length for these theta values
    tessellation_lengths = [tessellation_length(f1(theta), f2(theta), theta)
    for theta in theta_values]

    # Plot the right half unit circle
    x = np.cos(theta_values)
    y = np.sin(theta_values)

    fig, ax = plt.subplots()
    ax.plot(x, y, label="Right Half Unit Circle")
    ax.scatter(x, y, c=tessellation_lengths, cmap='viridis',

    label='Tessellation Lengths')

    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('Tessellations on Right Half of Unit Circle')
    ax.legend()
    plt.show()

plot_tessellation()
```



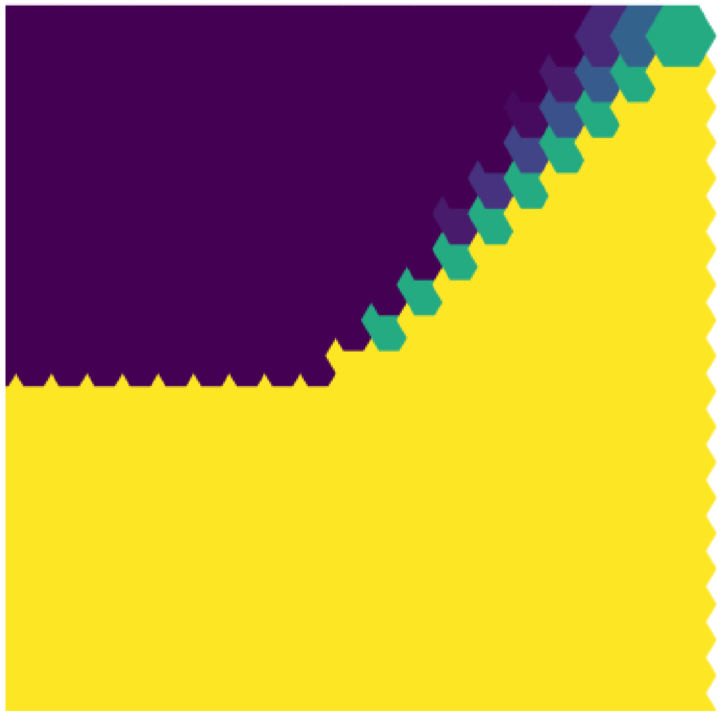
We outline an approximation algorithm aimed at minimizing error while constructing an efficient sweeping path:

1. Choose a starting point on the boundary $\partial\Omega$.
2. Initialize $\xi > 0$ as the discretization parameter.
3. Calculate the final position and orientation using the specified dynamical system.
4. Construct a sequence of points that form an approximate sweeping path subject to discretization constraints.

This algorithm is geometrically inspired and heuristics-based, ensuring computational efficiency for real-time applications.

6 Example Tessellations from the Method

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import networkx as nx

# Constants
theta_inf = 2 * np.pi
r = 0.5

# Functions f1 and f2
def f1(theta):
    return np.arcsin(np.sin(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

def f2(theta):
    return np.arcsin(np.cos(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

# Define decision graph for logic modulation
def create_decision_graph():
    G = nx.DiGraph()

    # Define logical nodes for quantifiers and logic operations
    logical_nodes = ['phi_eq_psi', 'some_other_node']  # Adjust as needed
    for node in logical_nodes:
        G.add_node(node, value=np.random.rand())  # Assign random values

        for demonstration
    return G

G = create_decision_graph()

# Define a logic vector calculation
def logic_vector(G, node_name):


    # Placeholder function, not based on meaningful logic yet
    return G.nodes[node_name]['value']  # Retrieves the assigned 'value'
    attribute from the node

# Tessellation visualization (corrected)
def visualize_tessellation(G, domain, hex_centers, hex_size):
    fig, ax = plt.subplots()

    # Loop through each hexagon center
    for center in hex_centers:
        # Convert cartersian (x, y) to polar (r, theta) to use f1 and f2
        x, y = center
        theta = np.arctan2(y, x)
        efficiency_value = f1(theta) * f2(theta)
        # Combine f1 and f2 for simplification

        # Fetch value from decision graph
        decision_value = logic_vector(G, 'phi_eq_psi')

        # Adjust efficiency based on decision value
        adjusted_efficiency = efficiency_value * decision_value

        # Create and draw hexagon adjusted by logic vector
        hexagon = patches.RegularPolygon(center, numVertices=6,
        radius=hex_size, orientation=np.pi/6)
        color_value = adjusted_efficiency
        # Placeholder: should be a mapping to a valid color range

        # Set color and add patch (corrected)
        hexagon.set_facecolor(plt.cm.viridis(color_value))

        # Set color without assignment
        ax.add_patch(hexagon)  # Add the hexagon patch to the plot

    ax.set_xlim(domain[0], domain[1])
    ax.set_ylim(domain[2], domain[3])
    ax.set_aspect('equal')  # Equal aspect ratio for x and y dimensions
    plt.axis('off')  # Turn off axis lines and labels
    plt.show()

# Define the domain and hexagon center function
domain = (-5, 5, -5, 5)  # Domain for plotting
hex_size = 0.5  # Size of hexagons

# Construct hexagon centers manually
hex_centers = [(i, j) for i in np.arange(domain[0], domain[1], hex_size)
                        for j in np.arange(domain[2], domain[3], hex_size)]

# Visualize the tessellated surface
visualize_tessellation(G, domain, hex_centers, hex_size)
```



The provided Python code illustrates an algorithm to visualize a tessellated surface using hexagons whose properties are influenced by a decision graph with logical nodes and geometry-modulating functions.

Mathematical Framework The tessellation process involves two critical functions, f1 and f2, which appear to be scalar fields that map the polar coordinate to a calculated value that influences tessellation:

$$f_1(\theta) = \arcsin\left(\sin(\theta)\right) + \frac{\pi}{2}\left(1 - \frac{\pi}{2\theta}\right) \tag{14}$$

$$f_2(\theta) = \arcsin\left(\cos(\theta)\right) + \frac{\pi}{2}\left(1 - \frac{\pi}{2\theta}\right) \tag{15}$$

These functions are continuous for all 0 and are utilized in the tessellation to modulate the properties of individual hexagons in the pattern. The choice of arcsine function suggests a periodic influence within the tessellation, potentially attending to the natural constraints of the surface.

The mesh initialization relies on an evenly spaced grid determined by hex centers in Cartesian coordinates, which are then mapped to polar coordinates within the visualization function. The hexagonal tessellation operates in this 2D domain described by the variable domain.

In the scope of topology, each point (x, y) is mapped to the tessellation efficiency using the functions f1 and f2 after converting to polar coordinates, . The radial aspect evokes a natural coordinate system, possibly intended to align with radially symmetric properties of the surface or light source distribution.

Logic Modulation and Visualization:

A directed graph G serves as a decision model, perhaps encoding logic or data pertaining to each hexagon's fitness regarding an overarching tessellation strategy. The nodes within G may represent choices or properties deemed significant in the tessellation:

$k =$

where k is the logical output that could symbolize decision-making processes, such as the reflectivity or permissibility of a hexagonal tile within the tessellation.

Visualization merges combinatorial logic and geometric heuristics to derive the tessellated surface. The output hexagon's color (color value) combines geometric modulating variables (from f1, f2) and the logic-driven decision$_v alue, illustrating via a color map the areas influenced by logical conditions$.

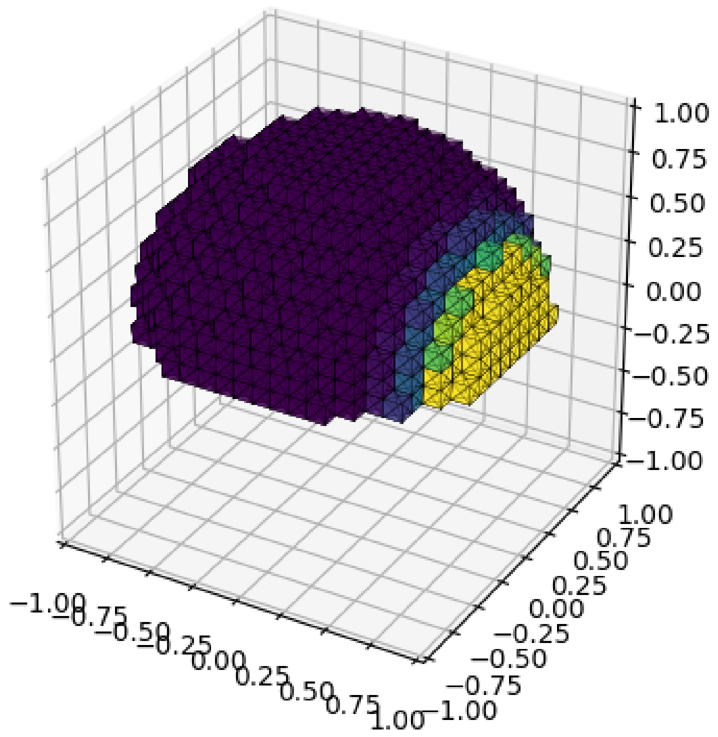
Interpretation and Usage The intersection of computational geometry and logic in this visualization has a broad applicability in fields like robotics, spatial analysis, and computational optics. In these fields, tessellations often underpin mesh generation for simulations, photorealistic rendering, and path planning.

It is worth noting that the validity of the tessellation approach hinges on the semantic linkage between polar coordinates (θ, r) and f1, f2 functionality. Moreover, the logic vector decisions influenced by G's logical nodes imply a flexibility that allows for optimized tessellations tailored to varied application-specific conditions.

Commentary on Code Structure and Style The code elegantly combines functional abstraction with procedural execution. The modeling of the problem in terms of polar coordinates implies an understanding of spatial symmetry, and the use of a decision graph suggests an appreciation for the mathematical rigor of computational logic. Computationally, converting Cartesian coordinates to polar within a loop is suboptimal, and caching these conversions could improve performance. Additionally, the logic model G is not fully exploited within the provided code but serves as a scaffold for refining the decision-making process for each element of the tessellation.

Incorporating dynamic logic that modulates the visual representation of computational tessellations presents an innovative approach, blending discrete mathematics with continuous geometries to produce a visualization rich in information and adaptable to diverse scenarios in computational design and analysis.

Examples:



7 Extrapolating into 3D

8 Logic Vectors as Directed Graphs and Geometric Logic

We want to essentially iterate the tessellation over a space that evolves through the logic vector directed graphs. The edges of the hexagon units of the tessellation are geometric logic vectors, and they manipulate the orientation and direction of the tessellation depending on the logical deductions and inferences based on the geometric interpretation of the vectorial logic assessments of other activities in the space.

The provided code above represents various computational concepts from different domains, ranging from symbolic logic operations to graph theory and mathematical transformations.

```
    import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import sympy as sp
import networkx as nx

# Constants
```

```
theta_inf = 2 * np.pi
r = 0.5

# Functions f1 and f2
def f1(theta):
    if theta == 0:
        return np.pi / 2
    return np.arcsin(np.sin(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

def f2(theta):
    if theta == 0:
        return np.pi / 2
    return np.arcsin(np.cos(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

# Define decision graph for logic modulation
def create_decision_graph():
    G = nx.DiGraph()
    logical_nodes = ['psi_1', 'psi_2', 'omega', 'phi']
    for node in logical_nodes:
        G.add_node(node, value=np.random.rand())
    return G

# Logic vector and logic operations
def update_logic_vector(G, logic_expr):
    G.nodes['phi']['value'] = logic_expr  # Update the logic value based on user input

# Function to randomly update the logic values for demonstration purposes
def random_logic_update(G):
    for node in G:
        G.nodes[node]['value'] = np.random.rand()

# Tessellation visualization
def visualize_tessellation(G, domain, hex_centers, hex_size):
    plt.ion()
    fig, ax = plt.subplots()
    for center in hex_centers:
        x, y = center
        theta = np.arctan2(y, x) if x != 0 else np.pi / 2
        efficiency_value = f1(theta) * f2(theta)
        logic_values = [G.nodes[node]['value'] for node in G]
        decision_value = np.mean(logic_values)
        adjusted_efficiency = efficiency_value * decision_value
        hexagon = patches.RegularPolygon(center, numVertices=6, radius=hex_size,

        orientation=np.pi/6)
        color_value = np.clip(adjusted_efficiency, 0, 1)
        hexagon.set_facecolor(plt.cm.viridis(color_value))
        ax.add_patch(hexagon)
    ax.set_xlim(domain[0], domain[1])
    ax.set_ylim(domain[2], domain[3])
    ax.set_aspect('equal')
    plt.axis('off')
    plt.show()

# Domain and hexagon size for tessellation
domain = (-5, 5, -5, 5)
hex_size = 0.5

# Construct hexagon centers
hex_centers = [(i, j) for i in np.arange(domain[0], domain[1], hex_size)
               for j in np.arange(domain[2], domain[3], hex_size)]

# Create decision graph with random node values
G = create_decision_graph()

# Simulation loop
for _ in range(3):  # Simulate user input and updating the graph 3 times
    # Randomly update the logic values
    random_logic_update(G)
    # Re-visualize the updated tessellation
    visualize_tessellation(G, domain, hex_centers, hex_size)
    plt.pause(1)  # Pause for visual effect

plt.ioff()  # Turn off interactive mode
plt.show()
```

8.1 Symbolic and Fourier Transformations

The first two code snippets demonstrate the applications of Sympy, a Python library for symbolic mathematics. They contain functions and symbols which allow for manipulation and representation of symbolic expressions. The Fourier series approximations mentioned are indicative of attempting to express a function as a series of sines and cosines, capturing the frequency domain representation of spatial patterns.

8.2 Network Graph Logic Modulation

We see the instantiation of a decision graph via NetworkX, a library suitable for the creation, manipulation, and study of the dynamics of complex networks. The directed graph G simulates logical connections between hypothetical state representations psi$_1$, $psi_2, andtheircombinedeffectonsomeresultantstateX.Thevisu$

Tessellation Visualization

The visualization functions within the last two snippets aim to generate a tessellated pattern modified by the logic vector—the tessellation here is achieved via hexagonal and triangular units,

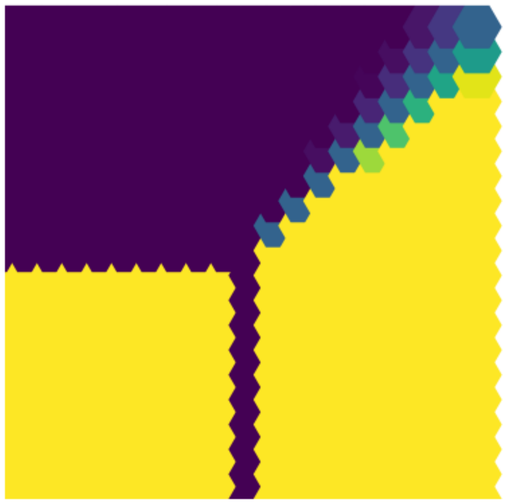


Figure 3: The base state

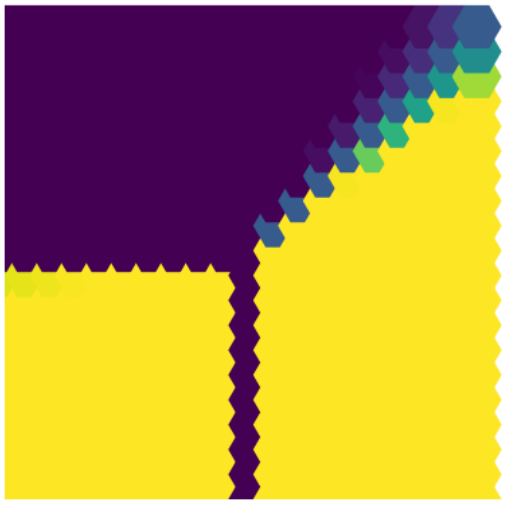


Figure 4: Mild Evolution of Coloring

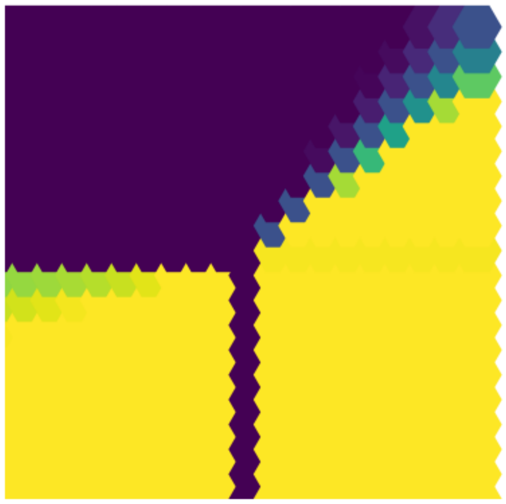


Figure 5: Increased Color Change Indicates a kind of implied gradation movement

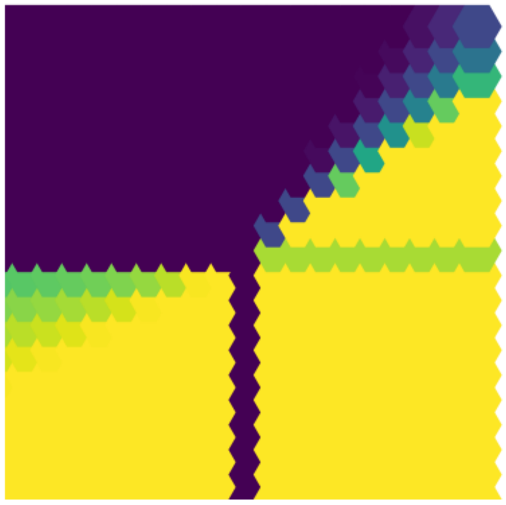


Figure 6: Increased Color Change Indicates a kind of implied gradation movement

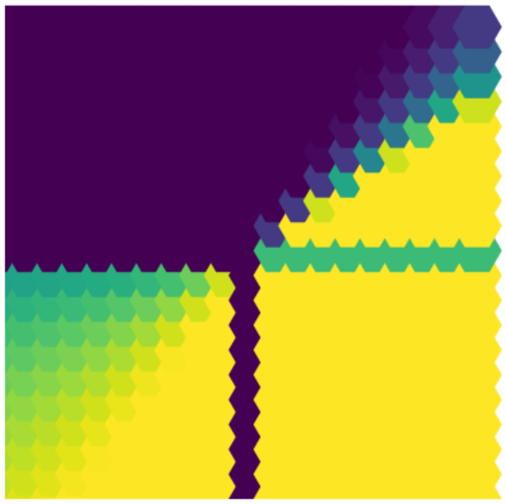


Figure 7: Increased Color Change Indicates a kind of implied gradation movement

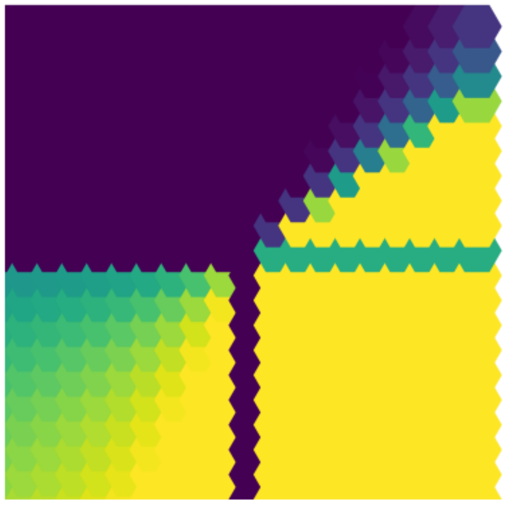


Figure 8: Increased Color Change Indicates a kind of implied gradation movement

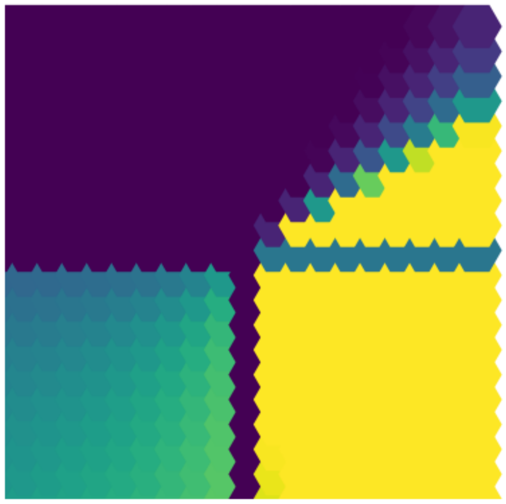


Figure 9: Increased Color Change Indicates a kind of implied gradation movement

wherein their properties are modulated by some logic vector, a collection of mathematical functions representing logical states. While the program is simple, it illustrates the prinicple that we can evolve the pattern using different logic functions illustrating a geometric interpretation of reasoning.

9 Quasi-Quanta and Evolving Chaos

Integrating Geometric and Logic Structures The collective goal of these programs is to iterate tessellations through a dynamically evolving space, influenced by logic vector directed graphs. Each tessellation element's edges represent geometric logic vectors—that is, the edges themselves have logical properties dictating the flow and structure of the tessellation. This behavior is akin to using an evolving map, where decisions at each geometric point affect the global arrangement of the tessellation, carrying implications for both structural integrity and optimized design, possibly within a machine learning or artificial intelligence context.

To illustrate this, we form an application of quasi-quanta symbolic transformations to visualize the evolving chaotic states of the system. We see that the colorations represent varying depths of, "runnels." This is illustrated using the program:

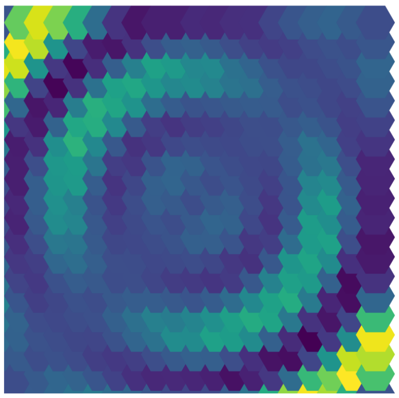
```
    import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import ipywidgets as widgets
from IPython.display import clear_output

# Imaginatively define some of the custom symbols with parameters and functions
Omega = 2.0
gamma = 0.9
```

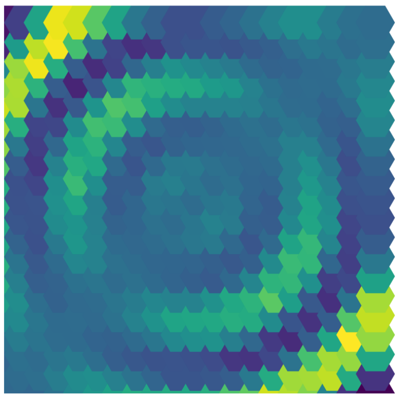
```
A_circle = 1.0
i_const = 1.0
heart_factor = np.random.rand()

# Custom function that interprets the expression for a given (x, y)
def interpret_quasi_quanta(x, y, t):
    # Add some randomness to the color variation
    random_effect = heart_factor * np.random.uniform(0.8, 1.2)

    # Calculate a difference representing Delta and Nabla as a distance from center
    distance = np.sqrt(x**2 + y**2)
    delta = np.abs(x - y)

    # Harmonic operations as a combination of sines and cosines
    harmonic = np.sin(Omega * distance * t) + np.cos(gamma * t)

    # Simulate the complex formula by combining terms in a creative way
    result = harmonic * delta * A_circle / (i_const + random_effect) * random_effect
    return result

# Tessellation parameters
domain = (-5, 5, -5, 5)
hex_size = 0.5
hex_centers = [(i, j) for i in np.arange(domain[0], domain[1], hex_size)
               for j in np.arange(domain[2], domain[3], hex_size)]

# Widget for time control
t_slider = widgets.FloatSlider(value=0, min=0, max=50, step=0.1,

description="Time", continuous_update=False)

# Visualization function that applies the interpretive quasi-quanta function
@widgets.interact(t=t_slider)
def update_visualizations(t):
    clear_output(wait=True)
    fig, ax = plt.subplots(figsize=(10, 10))

    # Calculate color values based on quasi-quanta function interpretation
    color_values = np.array([interpret_quasi_quanta(x, y, t) for x, y in hex_centers])
    # Normalize color values to [0, 1] range
    color_min, color_max = color_values.min(), color_values.max()
    color_values_normalized = (color_values - color_min) / (color_max - color_min)

    # Plot the tessellation and fill hexagons based on the color values
    for idx, center in enumerate(hex_centers):
        hexagon = patches.RegularPolygon(center, numVertices=6, radius=hex_size,
            orientation=np.pi/6)
        hexagon.set_facecolor(plt.cm.viridis(color_values_normalized[idx]))
        ax.add_patch(hexagon)

    # Finalize plot settings
    ax.set_xlim(domain[0], domain[1])
    ax.set_ylim(domain[2], domain[3])
    ax.set_aspect('equal')
    plt.axis('off')
    plt.show()

# Display the widget
display(t_slider)
```

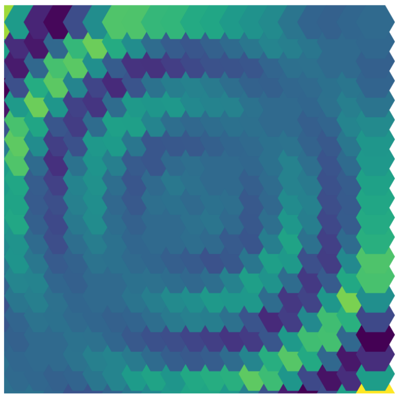
The form of the quasi quanta expression is taken from, "Combinations of Quasi Quanta Expressions," (Emmerson, 2023), from the package, "Quasi Quanta Language Package." It serves as an example form that large language models can apply to actual programs to visualize these kinds of pseudo-programs using Python.

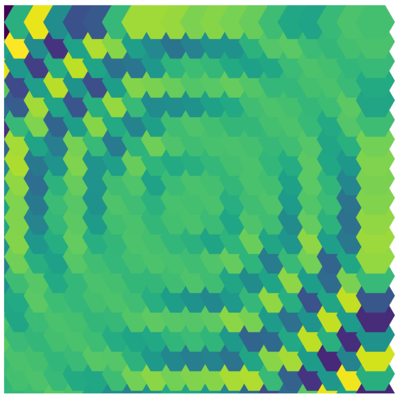
$$f \circ g = \bigcup_{x \in S_1 \cup S_2} x = \Omega \frac{\Delta \mathrm{i} \mathring{A} \sim}{\heartsuit \mathcal{H} \ \oplus \cdot} \star \frac{\star \mathcal{H} \Delta \mathring{A}}{\mathrm{i} \oplus \sim \cdot \heartsuit}$$

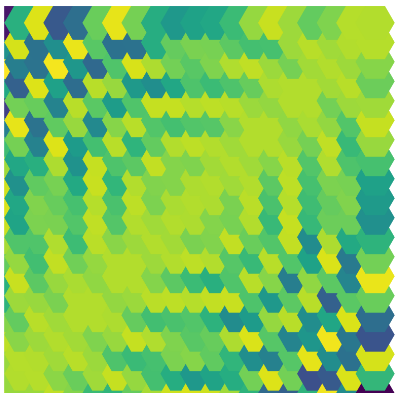
Thus, it serves as a more concrete example of how these linguistic expressions and linguistic synchronicity forms can be applied and continue to evolve.

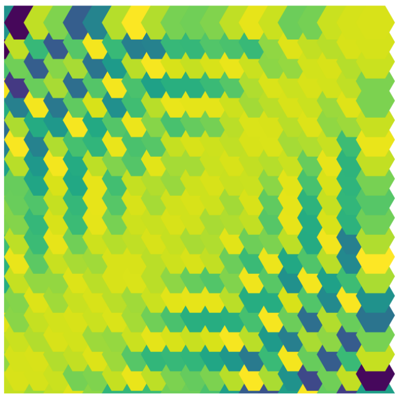

(a) t = .2

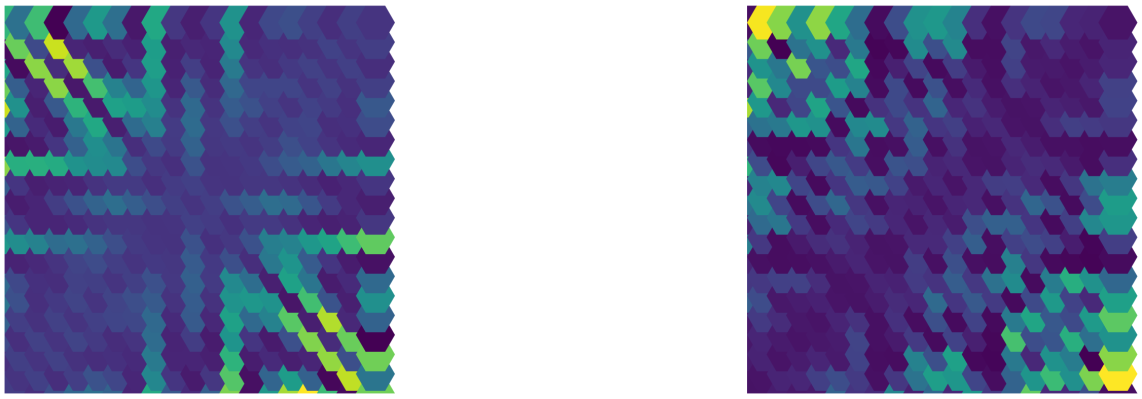

(b) t = .4


(c) t = .6


(d) t = .8


(e) t = 1.2


(f) t = 1.7


(a) t = 2.2

(b) Image 8

Figure 11: The pattern in the system resonance attains a higher degree of chaotic synchronicity as time increases. This paper is published as a package with the Jupyter Python notebooks, and I urge the reader to play with these programs as they see fit.

In practical terms, this could reflect a system where the tessellation adapts based on sensed or inferred information—a robot adjusting its path planning based on dynamic environmental variables, or a rendering system adjusting the level of detail based on viewer focus and processing power availability. Theoretically, the quasi quanta expressions then represent chaotic runnels through which language flows into and through the oneness of the living one.

10 Logic Vectors, Directed Graphs, Tessellation Associations

This type of system requires careful orchestration of the logic and geometry interplay. Defining a clear and coherent representation for the interaction between logical conditions and spatial tessellation is key. It ultimately represents a complex adaptive system where localized decisions and conditions propagate their influence through the network, affecting the larger whole and leading to emergent spatial behaviors and patterns. This integration has a multitude of applications, from computational physics simulations to optimizing rendering engines in computer graphics and ray tracing.

However, for simplicity sake, I just want to draw the analogy between the potential for logic statements to actively seek out proof environments by navigating geometric hexagonal tiling platform topologically. Essentially, this works by simply noting the relationship between the interactive compass, the symbolic directed graph (representative of a neural network), and the logic vector adaptation presented by the large language model. The large language model essentially interprets the logic vector (logic space geometry) as a mode of changing the functions of the tessellating pattern.

For instance,

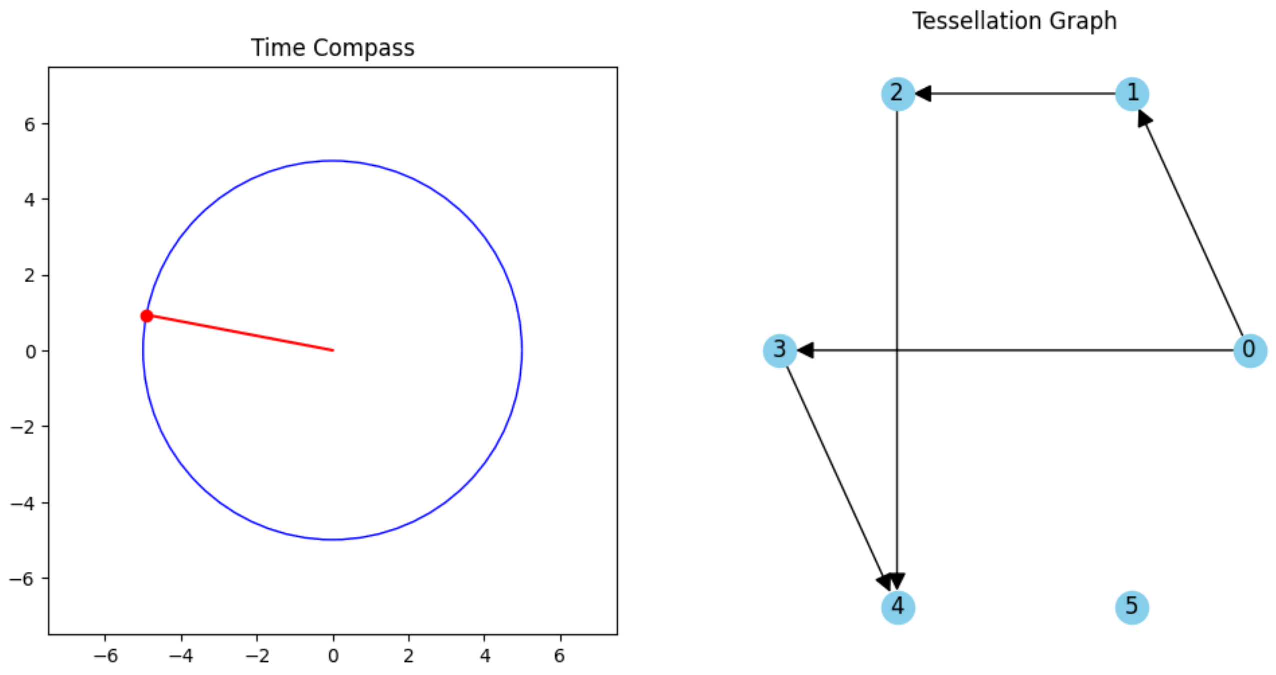


Figure 12: Example of an Arbitrary configuration of the Diredted Graph-Compass relationship.

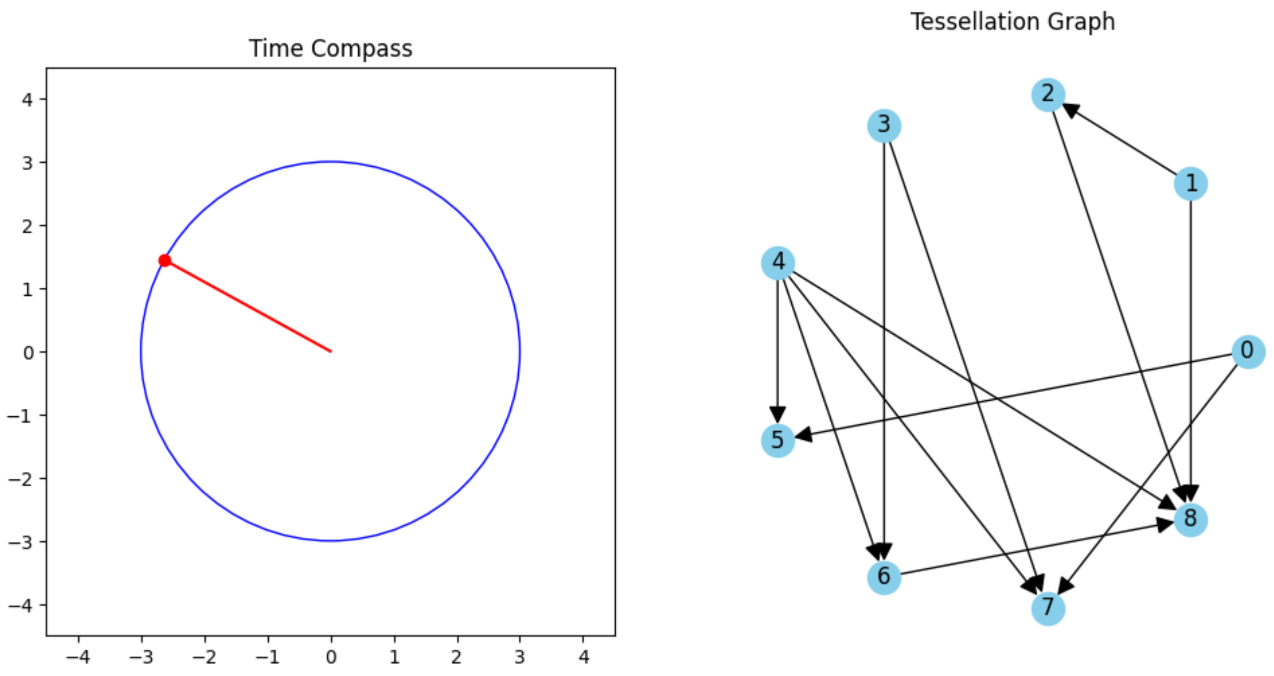


Figure 13: Example of an Arbitrary configuration of the Diredted Graph-Compass relationship.

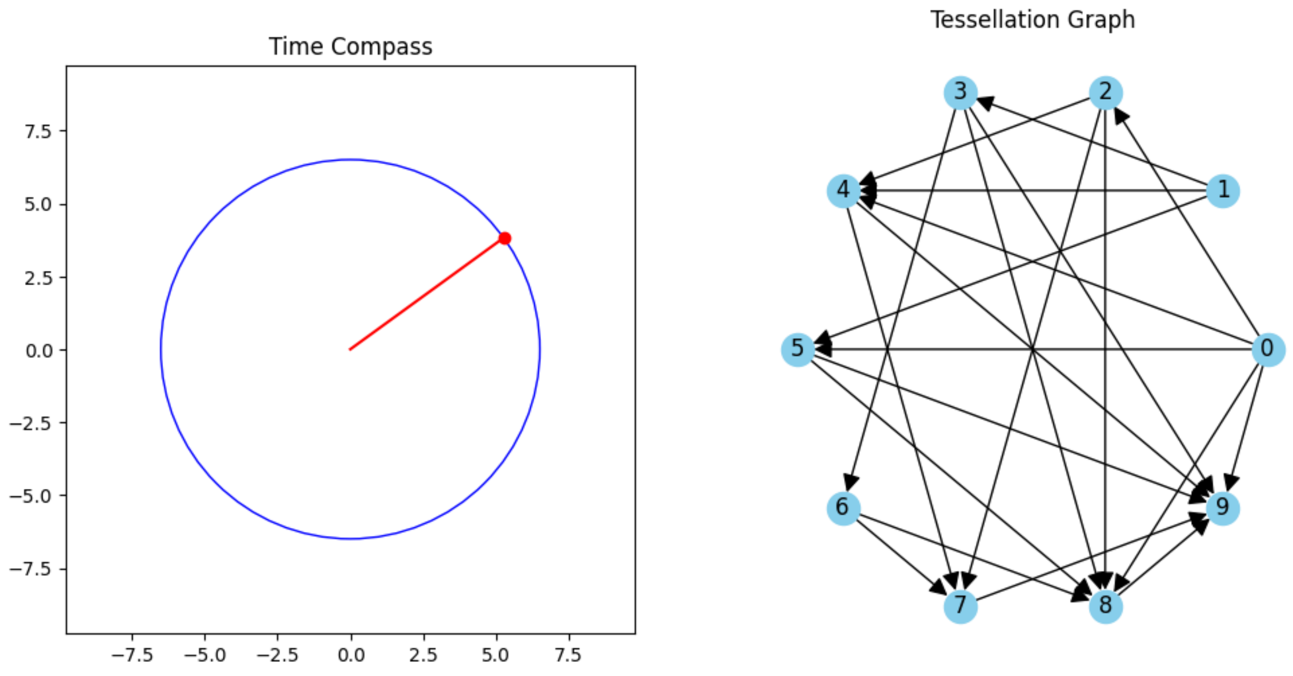


Figure 14: Example of an Arbitrary configuration of the Diredted Graph-Compass relationship.

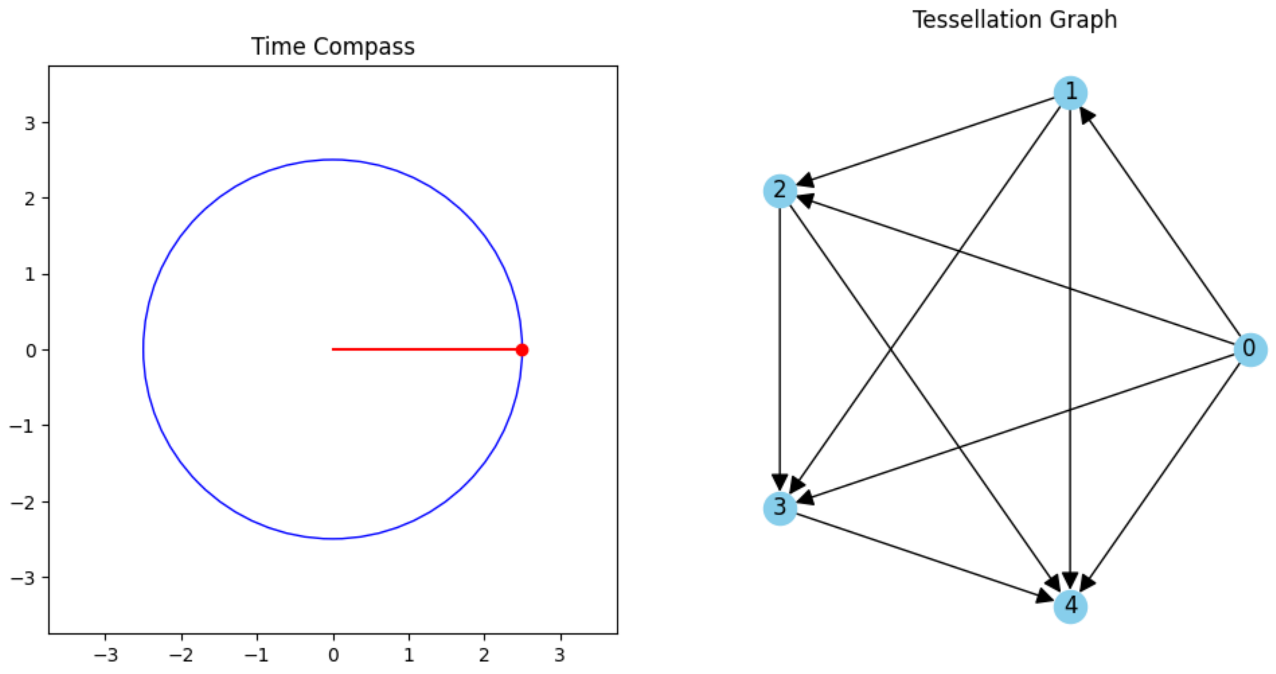


Figure 15: Example of an Arbitrary configuration of the Diredted Graph-Compass relationship.

In the above code, we see how the time compass concept can be integrated with the connections of a directed graph. Noticing this, we also notice that the time compass can be adapted to manipulate the tessellation coloration program based on the interpretation of a set of logic vectors to form runnels. This provides us a platform to connect chaos theory of an evolving dynamic system, logic-vector space, which shares the same supra-manifold as space-time mathematically (Supramanifolds of Logic, Emmerson 2023 (Limbertwig)), and the language of quasi-quanta synchronistic synergy. This illustrates that there are real-number programmatic interpretations of the entire system and that it isn't just nonsense-dingbat statements when interpreted through a large language model into functional code.

This kind of synthesis brings together computational logic, graph theory, and interactive visualization in a way that allows for rich user interaction.

Here, we associate the tessellation coloration as an ebb and flow transformation over time associated with the time-compass angular velocity conception, as

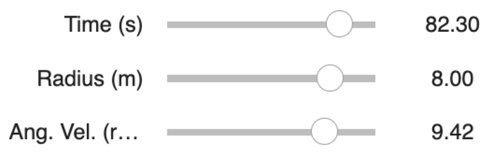


Figure 16: Slider (Angular Velocity adjusts the wave form within the circular band.)

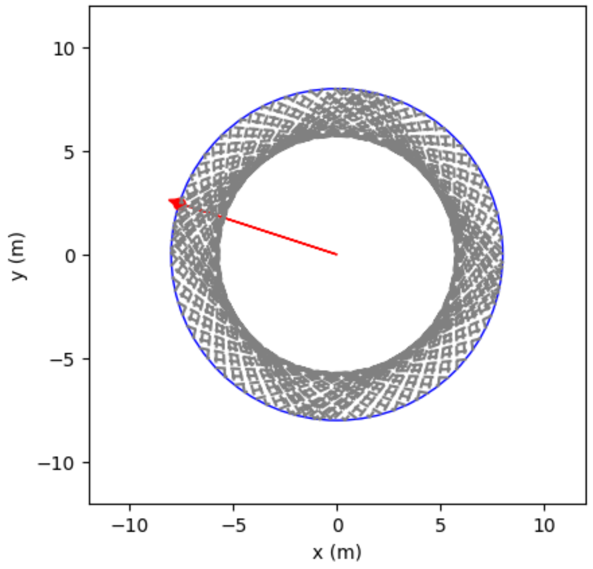


Figure 17: "Time Compass"

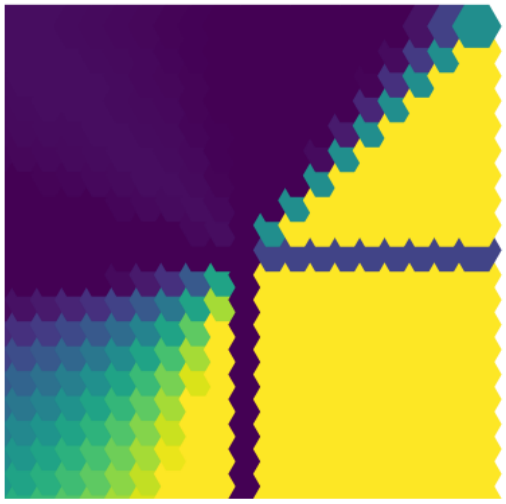


Figure 18: In this example, the Time Compass adjusts the ebb and flow of the tidal coloring.

Not only this, though we can demonstrably prove that logic-vectors are a spatial language that can be implemented in hard code using large language models to adapt the pseudo code of generative mathematical language patterns (symbolic of quasi-quanta topological infinity meanings):

Look:

```
import ipywidgets as widgets
from IPython.display import display
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import sympy as sp
import networkx as nx

# Define the functions required for the tessellation pattern
def f1(theta):
    if theta == 0:
        return np.pi / 2
    return np.arcsin(np.sin(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

def f2(theta):
    if theta == 0:
        return np.pi / 2
    return np.arcsin(np.cos(theta)) + (np.pi / 2) * (1 - np.pi / (2 * theta))

# Initialize domain and hex centers for the tessellation
domain = (-5, 5, -5, 5)
```

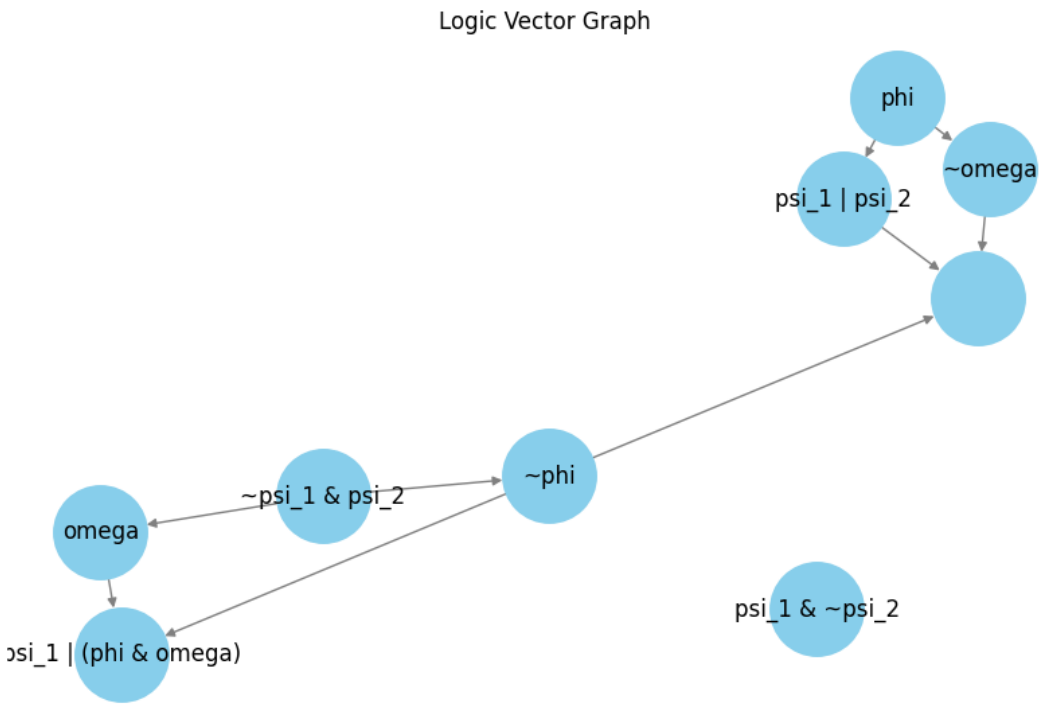


Figure 19: Code for this program is attached in the python jupyter file included within the .zip package of this paper.

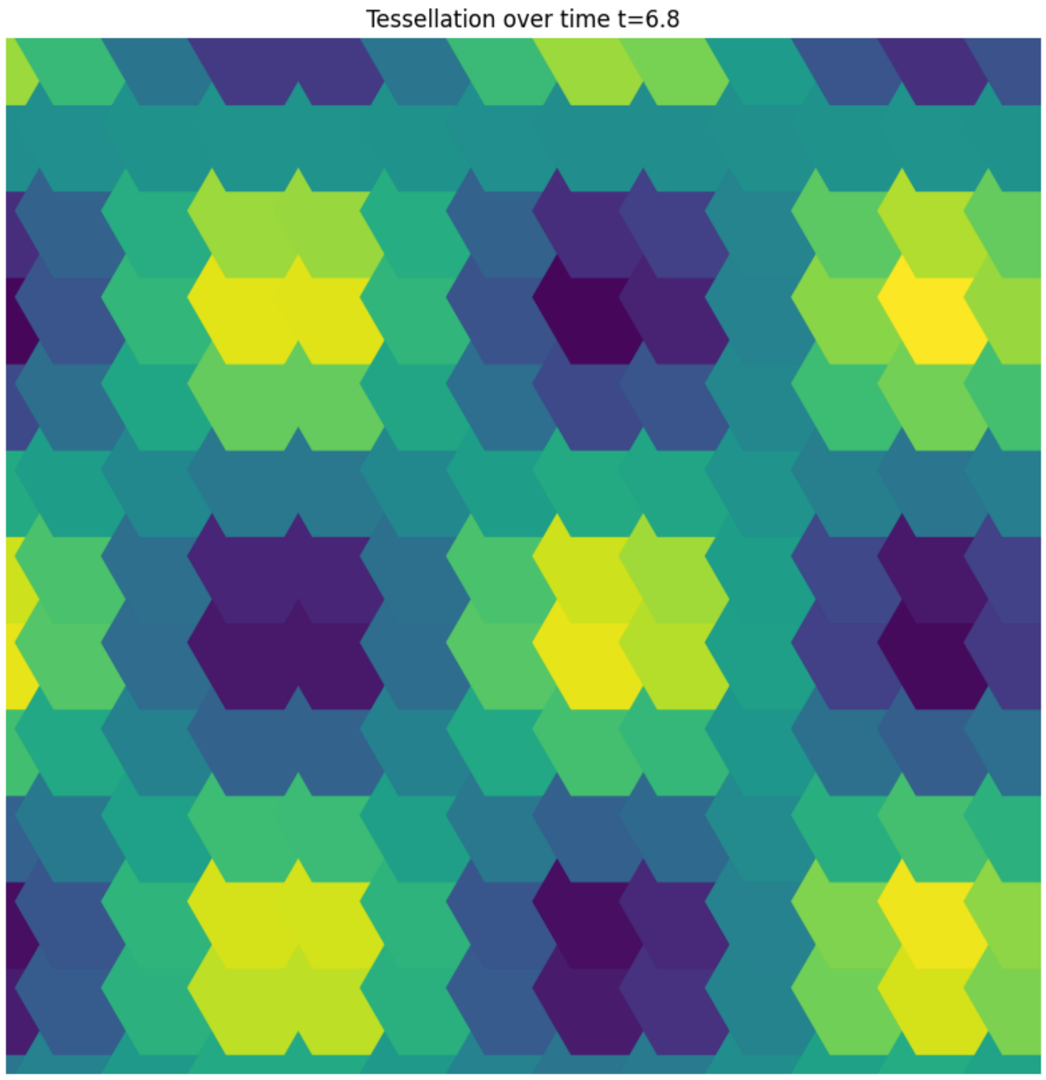


Figure 20: It stands to reason that logical associations that form a geometrically solid structure will form deeper runnels in the chaos theory itself as quasi-quanta syngergistically combine to form oneness indications from infinity meanings.

```
hex_size = 0.5
hex_centers = [(i, j) for i in np.arange(domain[0], domain[1],

hex_size) for j in np.arange(domain[2], domain[3], hex_size)]

# Define sliders for time, radius and angular velocity
t_slider = widgets.FloatSlider(value=0, min=0, max=100, step=0.1,

description="Time (s)")
R_slider = widgets.FloatSlider(value=5, min=1, max=10, step=0.5,

description="Radius (m)")
omega_slider = widgets.FloatSlider(value=2 * np.pi, min=0, max=4 * np.pi,

step=0.1 * np.pi, description="Ang. Vel. (rad/s)")

# Function to create the tessellation visualization
def visualize_tessellation(t, R, omega):
    fig, ax = plt.subplots()
    for center in hex_centers:
        x, y = center
        theta = np.arctan2(y, x) if x != 0 else np.pi / 2
        efficiency_value = f1(theta) * f2(theta) * (1 - f1(theta)) * (1 - f2(theta))
        color_value = np.clip(efficiency_value * np.sin(omega * t), 0, 1)

    hexagon = patches.RegularPolygon((x, y), numVertices=6, radius=hex_size,

    orientation=np.pi/6)
        hexagon.set_facecolor(plt.cm.viridis(color_value))
        ax.add_patch(hexagon)

    # Adjust the layout or structure of the graph based on the
    current x and y positions
    # of the particle on the time compass
    current_x_pos = R * np.cos(omega * t)
    current_y_pos = R * np.sin(omega * t)

    # The layout can be influenced by current_x_pos and current_y_pos

    # For example, we can use these values to determine the size or layout
    of the hexagons

    ax.set_xlim(domain[0], domain[1])
    ax.set_ylim(domain[2], domain[3])
    ax.set_aspect('equal')
    plt.axis('off')
    # Display the plot
    plt.show()

# Function to create the time compass visualization
def visualize_circular_motion(t, R, omega):
    fig, ax = plt.subplots()

    # Draw circle and plot the current position of the particle
    circle = plt.Circle((0, 0), R, color='blue', fill=False)
    ax.add_artist(circle)

    x_path = R * np.cos(omega * np.linspace(0, t, 100))
    y_path = R * np.sin(omega * np.linspace(0, t, 100))
    ax.plot(x_path, y_path, color='gray', linestyle='--')

    current_x_pos = R * np.cos(omega * t)
    current_y_pos = R * np.sin(omega * t)
    ax.arrow(0, 0, current_x_pos, current_y_pos, head_width=R/20,

    head_length=R/15, fc='red', ec='red')
    ax.scatter(current_x_pos, current_y_pos, color='red')

    ax.set_xlabel("x (m)")
    ax.set_ylabel("y (m)")
    ax.set_xlim(-R * 1.5, R * 1.5)
    ax.set_ylim(-R * 1.5, R * 1.5)
    ax.set_aspect('equal')

    # Update tessellation based on current values
    visualize_tessellation(t, R, omega)

    # Display the plot
    plt.show()

# Link sliders to visualization function
widgets.interactive(visualize_circular_motion, t=t_slider, R=R_slider,
omega=omega_slider)
```

In this example, we use the logic vector:
$\left(\frac{\forall x}{\delta}, \frac{\exists x}{\epsilon}, \frac{\forall x}{\tau}, \frac{\neg P(y)}{\Delta}, \frac{R(x) \implies S(x)}{\Delta}, \frac{\exists x \in \mathbb{E}, \forall y \in \mathbb{U}, P(x) \iff F(y)}{\Delta}\right)$

In this program, users can input a logical expression at each step to influence the visualization of the tessellation. The draw logic graph edge function has been added to draw the logic graph on the hexagon edges. The visualize tessellation function now includes the logic graph drawing inside the loop over each hexagon center to embed the logic graph visualization within each hex tile. The while True loop at the end of the program handles user input for interactivity.

11 Virtual Nerves

The tessellation, formed by a synergy that harnesses quasi-quanta significations to a oneness expression that maps the resulting conceptual energy number to the real number programming language, this pseudo-coding method has yielded results. Thus, another potential application of the chaos-expandin, runnel-forming, tessellation connection to logic vectors is virtual nerves as described by Buchanan.

In Quantization and torsion on sheaves I, Buchanan states:

"Let P_ε be an extended $\hat{p}$-complex, and $\Gamma_l\left[P_\varepsilon\right]$ the group of toric connections stratifying every $\hat{p}$ into a space E.

Fig. 2

P_ε regular with $01, 10$ poles

$\mathrm{P}^\circ = 0 \quad \partial(10)$

1

$\theta = \mathrm{P}^\circ \Pi_1\left(P^\star\right)$

"

The theory has potential in the sense that we can form a kind of nerve scaffold that connects via the time compass to the quasi-quanta linguistic synergy directing toward a symbolic analogical oneness expression (isn't it ironic that the directing toward a oneness in the energy number notation is so similar to the directing of the oneness in the nerve center).

Then, we write a program,

```
import ipywidgets as widgets
from IPython.display import display, clear_output
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import sympy as sp
import networkx as nx

# Slider widgets for controlling parameters
t_slider = widgets.FloatSlider(value=0, min=0, max=10, step=0.1,

description="Time (s)")
R_slider = widgets.FloatSlider(value=5, min=1, max=10, step=0.5,

description="Radius (m)")
omega_slider = widgets.FloatSlider(value=2*np.pi, min=0, max=4*np.pi, step=0.1*np.pi,

description="Ang. Vel. (rad/s)")

# Function that returns a new graph based on a given omega value
def create_graph(omega_value):
    # Here you can define the logic of how omega affects the graph structure
    # For demonstration, let's generate a radial layout with a number of nodes related
    G = nx.DiGraph()
    num_nodes = int(4 + np.abs(np.sin(omega_value)) * 10)
    for i in range(num_nodes):
        G.add_node(i)
        if i != 0:
            G.add_edge(0, i)
    return G

# Define the tessellation pattern and time compass function
def update_visualization(t, R, omega):
    # Clear any previous output
    clear_output(wait=True)

    # Create a new directed graph based on omega
    G = create_graph(omega)

    # Begin plotting side-by-side subplots
    fig, (ax_tess, ax_compass) = plt.subplots(1, 2, figsize=(12, 6))

    # Plot the time compass on the left subplot with dynamic outer ring thickness
    circle = plt.Circle((0, 0), R, color='blue', fill=False,

    linewidth=np.abs(np.sin(omega)) + 0.5)
    ax_compass.add_artist(circle)
    x_compass = R * np.cos(omega * t)
    y_compass = R * np.sin(omega * t)
    ax_compass.plot(x_compass, y_compass, 'ro')

    # Plot the moving point on the circle
    ax_compass.set_aspect('equal')
    ax_compass.set_xlim(-R * 1.5, R * 1.5)
    ax_compass.set_ylim(-R * 1.5, R * 1.5)
    ax_compass.set_title("Time Compass")

    # Plot the tessellation graph on the right subplot based on

    the omega-dependent graph
    pos = nx.spring_layout(G, iterations=50)
    nx.draw(G, pos=pos, ax=ax_tess)
    ax_tess.set_title("Directed graph affected by Omega")
    ax_tess.axis('off')

    # Update and display the figure
    display(fig)

# Interactive widgets to link parameters with the visualization
```

```
widgets.interactive(update_visualization, t=t_slider, R=R_slider, omega=omega_slider)
```

Because this program can be tied to the, "time compass," concept, and thus in turn, phenomenological velocity, we can formulate a theory of consciousness as the in-tandem existential interaction of the actualizing of the symbolic experiential plateau synergizing a spontaneous big-bang cosmograph with the structural nature of the geometric-logic game, but of course, we are left with just another lens on metaphors of consciousness. However, it seems a promising mathematical metaphor.

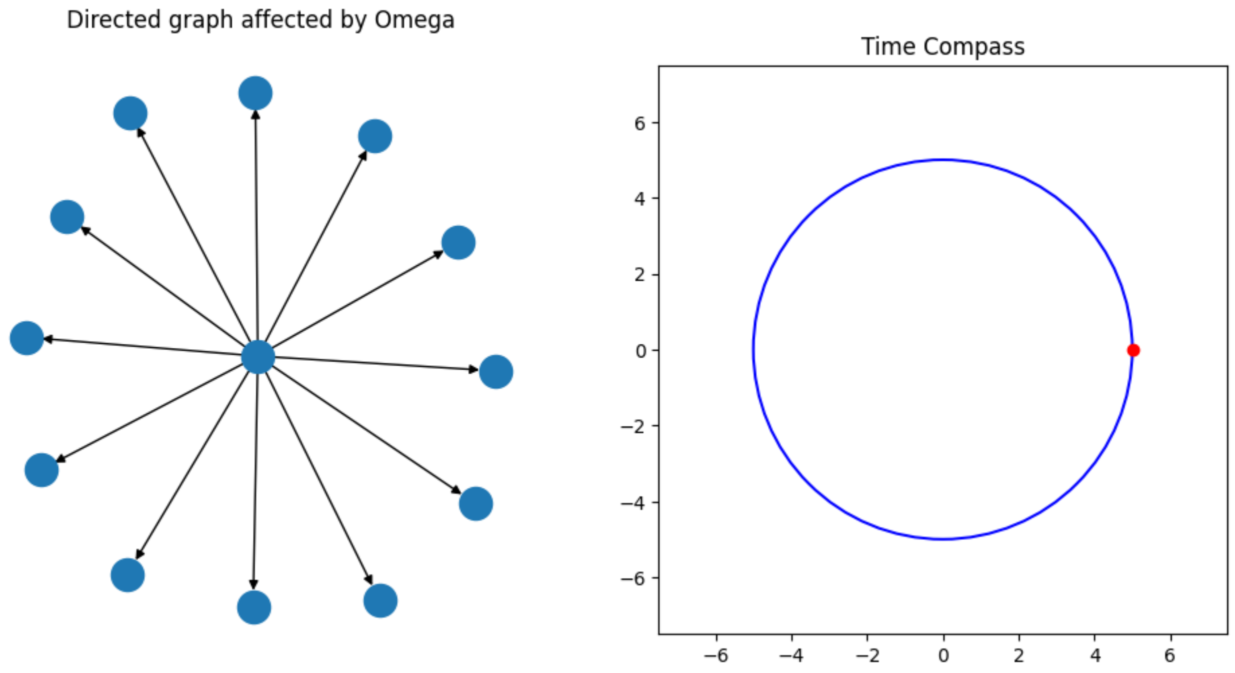


Figure 21: Slider (Angular Velocity adjusts the wave form within the circular band.)

12 Potential Applications of the Mathematical Linguistic Insight

Cellular Automata: These are mathematical models in which a grid of cells evolves through discrete time steps according to a set of rules based on the states of neighboring cells. This could be represented with a 3D tessellation where each layer represents a moment in time.

Graph Structures and Networks: Directed graphs and networks can visualize complex relationships. These can be precisely defined with nodes representing logic states and edges representing logical operations or transformations.

Fractals: Fractal structures emerge from simple rules applied recursively and can represent self-similar logic at different scales, much like how genetic information can be packed densely within a genome.

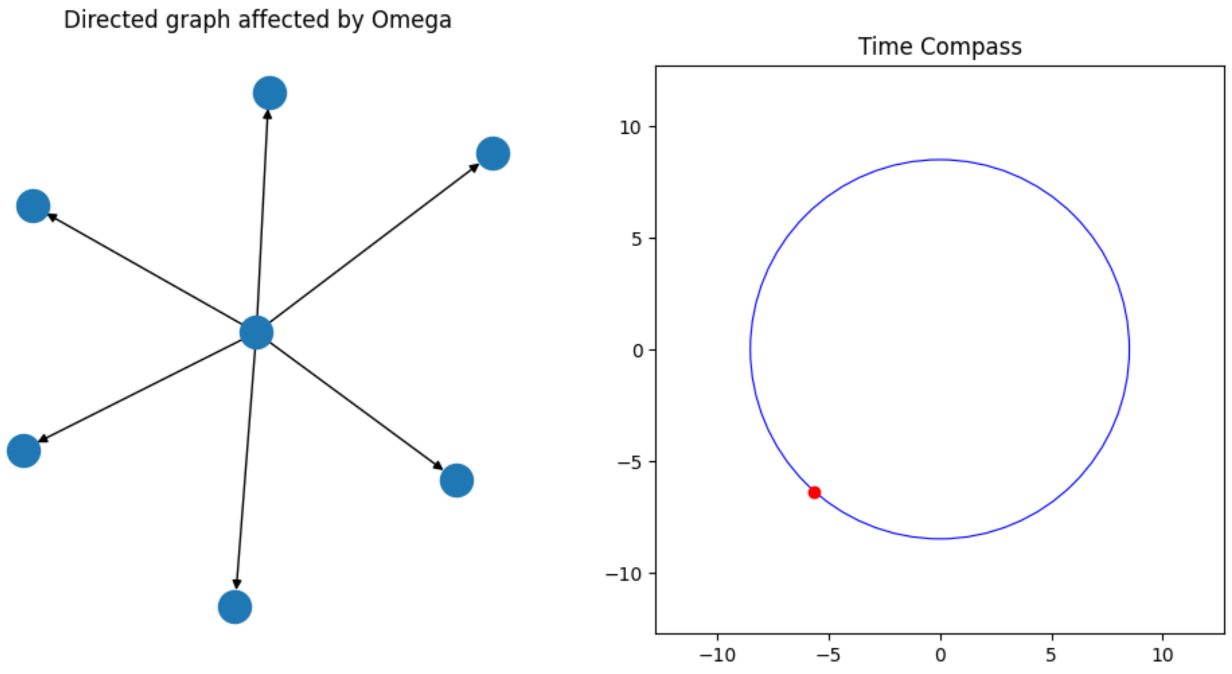


Figure 22: "Time Compass"

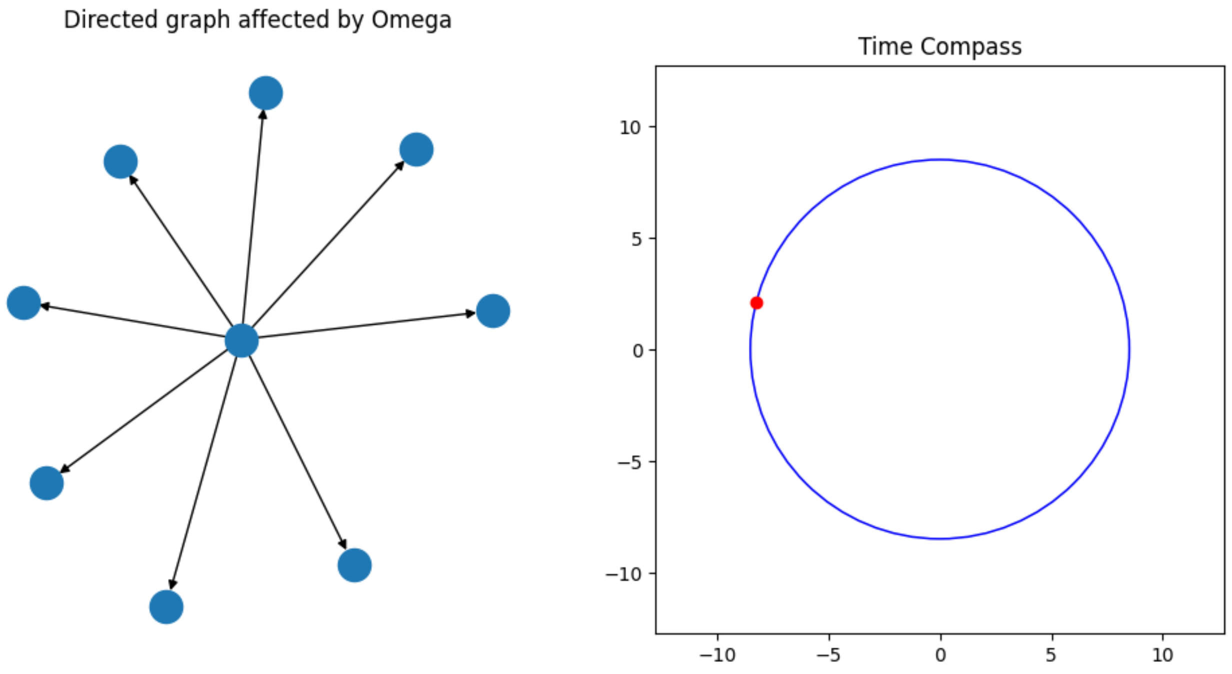


Figure 23: In this example, the Time Compass adjusts the ebb and flow of the tidal coloring.

Crystal Lattices: These structures can represent three-dimensional tessellations with repeating logic units, much like how chemical structures crystallize according to molecular interactions.

Quantum Logic Gates: These are used in quantum computing and could be represented by a complex 3D tessellation where each cell represents a qubit in a superposition of states.

Logic Solitons: In physics, a soliton is a self-reinforcing solitary wave that maintains its shape while traveling at a constant speed. Similarly, we could envision a visual representation where logic states propagate through a medium without dispersal.

Viral Capsids: The geometrical shapes of viral capsids can inspire new forms based on tessellation logic. They often exhibit icosahedral symmetry, which could be translated into structured logic patterns in 3D space.

To visualize these forms, we would define rulesets that determine the logical interactions and transformations between units, much like how physical laws govern the structure of natural entities. By rendering these rulesets visually, we create a visual representation of how complex logical structures might emerge from simple principles.

This type of modeling is highly abstract and crosses into areas of generative art, visual mathematics, and computational biology. Each extrapolation represents an opportunity to explore how patterns and logic can interplay to create rich and complex visual structures. Each form can also embody different logical operations or interactions, allowing for a visual analogy to computational processes or natural phenomena.

13 Conclusion

In this paper, we have explored geometrical constructs that define the optimal tessellation for ray tracing within spatio-temporal constraints.

Our algorithmic approach serves as a substantial advancement in discretizing and handling complex geometric configurations, permitting efficient computational simulations relevant to a range of applications in computer graphics, optics, and robotic path planning.

The exploration of tessellation efficiencies using polar coordinates, logic vectors, and directed graphs unveils a deeply interconnected framework between topology, computational geometry, and logic. This intricate blend not only advances our understanding of mathematical landscapes but also opens avenues for innovative applications across robotics, spatial analysis, and computational optics. By translating logical operations into visual representations through tessellation, we create a richly informative method for interpreting and manipulating complex datasets and processes.

The code structure and style discussed herein adeptly marry the abstract with the procedural, unveiling an elegance in the computational modeling of spatial symmetries and logic operations. Despite the computational challenges, such as the suboptimal conversion of Cartesian to polar coordinates within iterative loops, the scaffold provided by the logic model G is a testament to the adaptability and potential of this approach to optimize tessellations tailored to specific applications.

Extending these concepts into three-dimensional spaces and beyond, we embark on a journey through potential forms that mimic the foundational structures of nature, from the double helix to crystal lattices and quantum logic gates. Each of these extrapolations offers a unique perspective on how logical structures emerge from simple rules, invoking a deeper appreciation for the inherent logic that shapes our world.

Potential applications, as discussed, span from visually representing quantum computing phenomena to The exploration of tessellation efficiencies using polar coordinates, logic vectors, and directed graphs unveils a deeply interconnected framework between topology, computational geometry, and logic. This intricate blend not only advances our understanding of mathematical landscapes but also opens avenues for innovative applications across robotics, spatial analysis, and computational optics. By translating logical operations into visual representations through tessellation, we create a richly informative method for interpreting and manipulating complex datasets and processes.

The code structure and style discussed herein adeptly marry the abstract with the procedural, unveiling an elegance in the computational modeling of spatial symmetries and logic operations. Despite the computational challenges, such as the suboptimal conversion of Cartesian to polar coordinates within iterative loops, the scaffold provided by the logic model G is a testament to the adaptability and potential of this approach to optimize tessellations tailored to specific applications.

Extending these concepts into three-dimensional spaces and beyond, we embark on a journey through potential forms that mimic the foundational structures of nature, from the double helix to crystal lattices and quantum logic gates. Each of these extrapolations offers a unique perspective on how logical structures emerge from simple rules, invoking a deeper appreciation for the inherent logic that shapes our world.

Potential applications, as discussed, span from visually representing quantum computing phenomena to modeling the evolution of viral capsids. The symbolic and generative capacities of these mathematical and computational models unfold a rich tapestry of logical and geometric relationships that can be harnessed for varied and complex problem-solving scenarios, from AI-driven analytics to the optimization of photorealistic rendering.

In conclusion, this paper has demonstrated the profound utility and versatility of combining computational geometry with logic through tessellation. As we move forward, the continued development and refinement of these methodologies promise to unlock further innovations in the fields of computational design, analysis, and beyond. The path ahead is rich with possibilities, inviting further exploration into the dynamic interplay between geometry, logic, and the computational arts.

References

[1] R. Buchanan, *Quantization and Torsion on Sheaves I*, Independent Journal of Math and Metaphysics, 2023, Available at: https://www.academia.edu/99676315/Quantization$_and_torsion_on_sheaves_I$.

[2] P. Emmerson, *Counter Calculus in Search of Greater Abstract Universality*, Zenodo, 2020. https://doi.org/10.5281/zenodo.4317712

[3] P. Emmerson, *Research on Energy Numbers and Associated Mathematical Structures*, Zenodo, 2020. https://doi.org/10.5281/zenodo.10541666

[4] P. Emmerson, *Exploring the Possibilities of Sweeping Nets in Notating Calculus- A New Perspective on Singularities*, Zenodo, 2020. https://doi.org/10.5281/zenodo.10433888

[5] P. Emmerson, *Pseudo Function Example*, Zenodo, 2020. https://doi.org/10.5281/zenodo.10373727

[6] P. Emmerson, *Vector Calculus of Notated Infinitones*, Zenodo, 2021. https://doi.org/10.5281/zenodo.8381918

[7] P. Emmerson, *Quasi-Quanta Language Package*, Zenodo, 2021. https://doi.org/10.5281/zenodo.8157754

[8] P. Emmerson, *Infinity: A New Language for Balancing Within*, 2023, DOI: https://doi.org/10.5281/zenodo.7710323.

[9] P. Emmerson, A New Function of Homological Topology Available at: https://zenodo.org/record/7493362.

[10] P. Emmerson, Pre-Eminent Numeric Energy: The Theory of the Energy Number Available at: https://zenodo.org/record/7574612.

[11] P. Emmerson, The Geometry of Logic V1 Available at: https://zenodo.org/record/7556064.

[12] P. M. Morse, *Diatomic molecules according to the wave mechanics. II. Vibrational levels*, Physical Review, vol. 34, no. 1, pp. 57–64, 1929.

[13] B. Grünbaum and G. C. Shephard, *Tilings and Patterns*, W. H. Freeman, 1987.

[14] M. Senechal, *Quasicrystals and Geometry*, Cambridge University Press, 1996.

[15] A. S. Glassner, *Principles of Digital Image Synthesis*, Morgan Kaufmann, 1995.

[16] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory To Implementation*, Morgan Kaufmann, 3rd edition, 2016.

[17] J. Arvo (ed.), *Graphic Gems Package: Graphics Gems II*, Academic Press, 1991.