Collision-Based Model of Plant Phyllotaxis

Mitchel Myers

**Professor: Dr. Jeyaprakash Chelladurai**

*COMP430: Interactive Computer Graphics*

*Project Report*

*Department of Applied Computer Science*

*Lock Haven University of Pennsylvania*

*Lock Haven, PA, USA*

Fall 2017

# 1. Introduction

## 1.1 Background

It can often be observed that the arrangement of plant organs attains a spiral like pattern. This arrangement is known as a phyllotaxis. The columns of the spiral are known as *parastichies*. Typically, the number of *parastichies* running in opposite directions are two adjacent numbers of the Fibonacci sequence [1]. The divergence angle of plant organs is very close to the Fibonacci angle, also known as the golden angle [1]. The golden angle is equal to approximately or exactly radians. Some geometric models of plant organ placement have been developed using this information. However, these models have limitations such as assuming equally sized organs or placement of organs on a flat disk. In paper in [1] proposes a collision-based model for the generation of phyllotaxis. This model provides a more generalized and flexible geometric arrangement of plant organs.

## 1.2 Collision-Based Model

Phyllotactic patterns are often observed in mature plants. However, the origins of these patterns occur early in the plants’ development. The plant’s organs begin as primordia which are pushed out from the base at a divergence angle of approximately 137.5°. As the plant grows, the primordia develop into mature organs and form the final shape of the plant. In the collision-based model, we begin with an empty receptacle of a predefined shape. The goal of the collision-based model is to distribute primordia on to the surface of the receptacle in a manner consistent with the natural phyllotaxis arrangement. While primordia in nature are pushed into their final position during plant development, this model places primordia on a fully formed receptacle by avoiding collisions with neighboring primordia.

## 1.3 Contribution

The purpose of this project is to implement the collision-based model of phyllotaxis generation as proposed by [1]. This model can be used to generate phyllotaxis for arbitrarily shaped planned. This simplifies the process of procedurally generating plants in scenes. The model is written in Python. It is intended for use with Blender through the bpy library, but can be used with other Python programs. It can be adapted to other languages as well.

## 1.4 Organization

The rest of this report is organized as follows: Section 2 gives a formal description of the algorithm. Section 3 describes the implementation. Section 4 concludes the report and discusses future work.

# 2 Formal Description

The algorithm uses a parameter to represent the position of primordia along the receptacle. The value of ranges between and The base of the receptacle occurs at and the tip occurs at Additionally, the parameter represents the divergence angle of a given primordium with respect to the initial primordium. A 2-D parameterized curve , parameterized as a function of t, models the shape of the receptacle. The receptacle is a surface of revolution generated by sweeping this curve around the receptacle axis. Therefore, the position of a given primordium can be given fully as pair of values, and , where is the index of the primordium according to the order of placement. represents the point on the receptacle at position and angle . Additionally, each primordium has a radius . The values generated by the collision-based model can then be described in this system. The initial primordium has the values and . The computation of each subsequent primordium is based on preceding primordia, such that . Subsequent are be calculated by finding the minimum in the range to such that the resulting primordium does not collide with any preceding primordia. It can be said two primordia are not colliding if the distance between their centers is greater than the sum of their radii. This can be represented formally as , where and are indices of primordia [1].

# 3 Implementation

In this section the implementation of the algorithm is described. Each code block is followed by its description. Descriptions following function headers describe the function and its parameters in general terms. Other descriptions describe the operation being performed

## 3.1 Collision-Based Model Generation

The algorithm is implemented in the generateSprialModel defined as follows:

def generateSpiralModel (curve, tMin, tMax, radii,

maxTry = 1000):

The function takes 4 to 5 parameters. The curve parameter is a function of which describes the curve. The tMin and tMax parameters correspond to the and in the formal description. The radii parameter defines the size of the primordia as a function of the primordia’s index. The optional maxTry defines the max primordia to generate. This prevents the algorithm from running without end.

angle = 0

t = tMin

index = 0

radius = radii(index)

primordia = [(angle, t, radius)]

This section defines the initial values. The initial angle is , the initial -value is . The index variable tracks the index of the current primordium. It is then used to as the parameter to the radii function to generate the first primordium’s radius. The first primordium is generated as the first element in the list of primordia. Each primordium is represented as a 3-tuple containing the angle offset of the primordium, the -value describing its position along the curve, and its radius.

while index < maxTry:

index += 1

angle += goldenAngle

radius = radii(index)

t = binarySearch(tMin, tMax,

lambda t: checkCollisions(

curve, t, angle, radius, primordia))

if t is None: break

primordia.append((angle, t, radius))

The while loop iterates up to maxTry times, limiting the number of primordia generated. Within each loop, the index is incremented. Then, the angle is incremented by goldenAngle which is computed as described in Section 1.1. Additionally, the radius of the next primordia is calculated and stored in radius. Next, the loop calls the binary search algorithm described below to find the -value of the current primordium. The result of the binarySearch is the least value of such that the current primordium does not collide with any existing primordia. The binarySearch function returns None when it fails (a -value could not be found that does not collide with an existing primordium). This condition is used to break the loop. Otherwise, the primordium is added to primordia and the loop continues.

return [(curveToPoint(curve, angleX, tX), radiusX)

for (angleX, tX, radiusX) in primordia]

Finally, the list of primordia is mapped from 3-tuples to 2-tuples. The first element is a 3D position calculated from the primodium’s angle and -value. The curveToPoint function performs this transform. The second element is the same radius from the 3-tuple. The resulting list is returned as the result of the algorithm.

## 3.2 Binary Search Algorithm Implementation

The binary search algorithm is implemented as follows:

def binarySearch (xMin, xMax, predicate,

maxError = 1e-8):

The binarySearch function attempts to find a value the least value of x in the range of xMin to xMax such that predicate returns true. predicate is a function of x. The maxError parameter describes the maximum error allowed in the approximation of x.

x = (xMin + xMax) / 2.0

error = abs((xMax - xMin) / 2.0)

foundAny = False

The x variable is initialized to the midpoint between xMin to xMax. The error variable tracks the current error in the approximation. The foundAny variable tracks whether any value of x has been found that satisfies predicate.

while error > maxError:

if predicate(x): # move closer to xMin

xMax = x

foundAny = True

else:

xMin = x

x = (xMin + xMax) / 2.0

error /= 2.0

This is the core of the algorithm. While the current error is greater than the max error the loop runs. When the predicate succeeds, the search range is halved towards xMin by setting xMax to x. Additionally, foundAny is set to true to indicate predicate has succeeded for at least one value of x. When predicate fails, the search range is halved towards tMax by setting tMin to x. Finally, x is recalculated as the midpoint of the search range, and the current error is halved.

return x if foundAny else None

Finally, the current x is returned if any value of x has been found than satisfies predicate. Otherwise, the algorithm fails by returning None.

## 3.3 Additional / Utility functions

This section discusses the purpose of other functions used during the algorithm without going into detail of their implementation. As such, the functions are identified by their signatures, including function name and parameter names. This is done rather than listing code blocks.

The checkCollisions(curve, t, angle, radius, primordia) function checks whether a new primordium collides with any existing primordium. The first four arguments are used to generate the position and size of the new primordium. The primordia argument is the current list of existing primordia which is used to check for collisions. The function is passed as the predicate to the binary search algorithm in the main loop of generateSpiralModel.

The collides(primordia0, primordia1) function checks whether primordia0 collides with primordia1. Each primordium is given as a 3-tuple, as described in Section 3.1. This function is used by the checkCollisions function.

The distance(point0, point1) function calculates the distance in 3D space between point0 and point1. This function is used by the collides function.

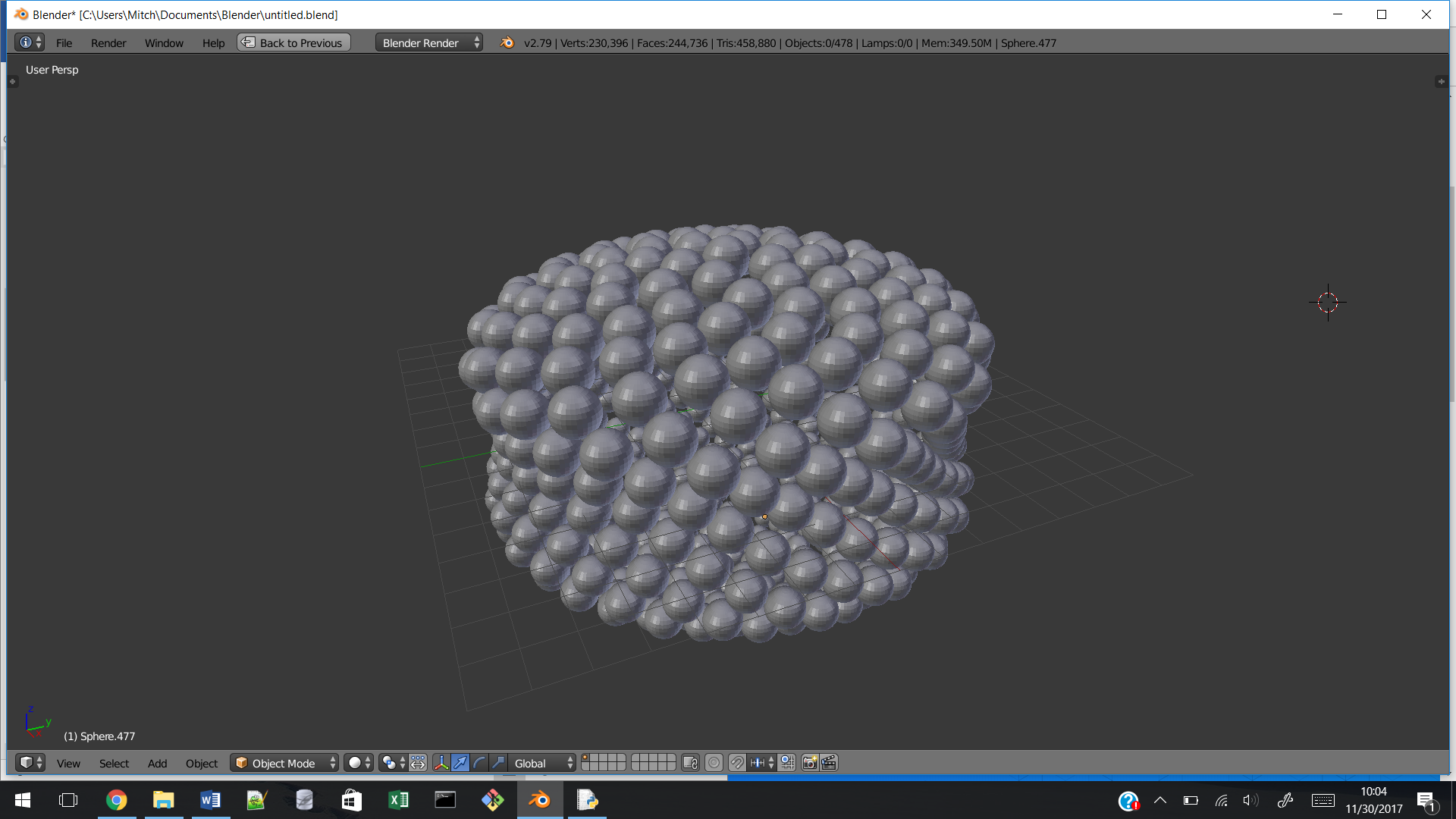
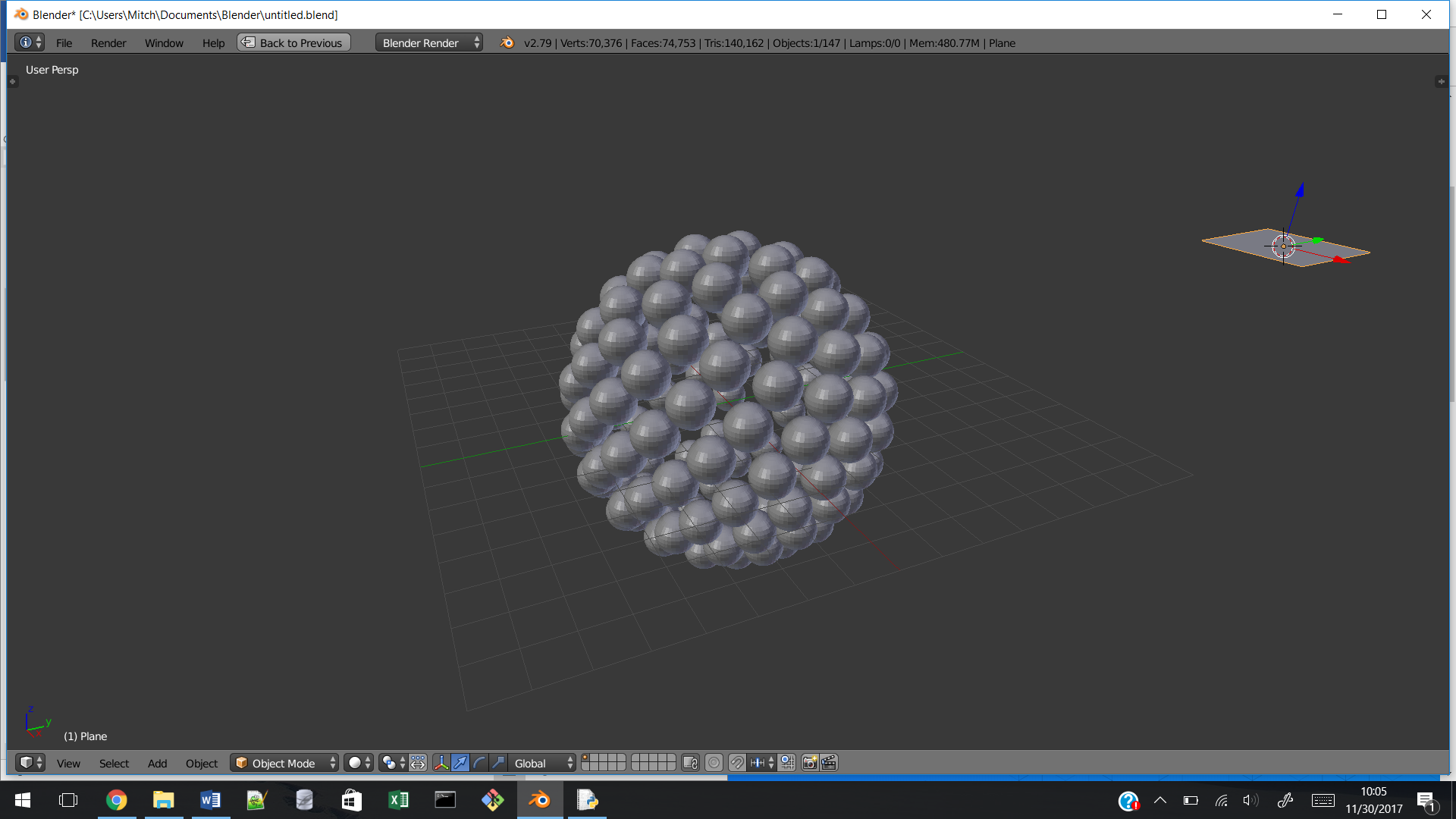
The curveToPoint(curve, angle, t) function generates a point in 3D space given a curve function, an angular offset, and a -value. This function is used anywhere the algorithm needs a position in 3D space rather than a (angle, -value) pair for a primodium.

Other functions defined in the project are used either to execute the algorithm (e.g. main and doFullGenerate) or as utility functions specifically for the integration with Blender (e.g. blend and blendColor).

# 4 Conclusions

## 4.1 Results

The collision-based model appears to approximate the phyllotactic plant models to a satisfactory extent. Below are images of the some of the models generated with various curves. The results of the model were used to generate models in Blender where spheres represent the primordia generated.

Figures 1 and 2: Models generated by the algorithm for various curves.

# 4.2 Future Work

This section discusses potential extensions to and modifications for this project.

Currently, the output of the algorithm is a list of pairs containing a 3D point in space and a radius. This is enough information to generate a spherical primordium. However, in the future, one may desire to replace primordia with modelled organs. Additional information, such as the outward angle of the primordia should also be generated by the algorithm.

In the current algorithm, when adding a new primordium to the model, the program checks whether the new primordium collides with any existing primordium. Additionally, the check is performed multiple times for each primordium during the call to the binary search algorithm. As the number of generated primordia increases, the number of checks increases linearly as well. Overall, the algorithm is approximately . Future work may make use a more complex data structures to reduce the time complexity of the algorithm.

Currently, the current implementation takes a curve function which only produces a single value for a given . The return value is used as the radius of the surface of revolution at the given height . In the future, it may prove more useful for both this radius and height be separate functions of For example, it may then be easier to use Bézier curves to model the shape of the plant.

References

[1] Fowler, D. R., Prusinkiewicz, P., & Battjes, J. (1992). A Collision-based Model of Spiral Phyllotaxis. *Computer Graphics (SIGGRAPH 92 Proceedings)*, 26(2): 361-368.