1 Background

For my project, I implemented the space colonization algorithm for modeling trees as outlined in this paper:


The algorithm starts off with a set of points which will act as the leaves of the trees and a root. The goal is to grow the branches of the root towards the leaves. The result is hopefully a structure that resembles a tree.

Figure 1: Tree rendered with 200 Leaves
2 Algorithm

For this project, I implemented the algorithm for a 2D tree. In general the algorithm is as follows:

(a) Root extends until within detection range (maxDist) of at least 1 Leaf

(b) Every leaf within detection range finds the closest branch

(c) Direction vectors are calculated from the detected leaves to their respective closest branch

(d) From each branch determined to be closest to a leaf, find the average direction vector of every direction from that branch to their leaves

(e) Create a new branch in direction from step d

(f) Check if the new branch enters the kill distance (minDist)

(g) Delete the visited leaves

(h) Loop back to b, until no new branches are found (either because there are no leaves at all, or if there are no leaves within any branch’s detection range
3 Implementation

3.1 Data Types

3.1.1 Leaf

A leaf is represented by a position in a 2D world. It also has a Bool attribute to show that it has been killed.

```haskell
    type Point = (Float, Float)
data Leaf = Leaf Point Bool | None
```

3.1.2 Branch

A Branch also has a position, but also includes a parent branch and a direction vector. The length of each branch is the same for every branch.

```haskell
    data Branch = Empty | Branch {
        position :: Point,
        parent :: Branch,
        direction :: Point
    }
```

3.1.3 Tree

A tree is the overarching data structure that holds the leaves and branches in lists. max_dist is the maximum detection radius of each leaf and min_dist is the kill radius of each leaf. Both are provided by the user.

```haskell
    type Leaves = [Leaf]
type Branches = [Branch]
data Tree = DONE Tree | Tree {
    leaves :: Leaves,
    root :: Branch,
    branches :: Branches,
    max_dist :: Float,
    min_dist :: Float,
    detected :: Bool
    }
```
3.1.4 Algorithm

The main computation of the implementation of this algorithm can be summarized with the following pseudo-code:

```plaintext
closestBranches = []
For each alive leaf:
closest = findClosestBranch(leaf, branches)
    new_direction = normalize(calculateDirection(leaf, closest))
    closestBranches.push((closest, new_direction))

groupedBranches = groupByBranch(closestBranches)

newBranches = []
For each group in groupedBranches:
    branch = group.shared_branch()
    sum_direction = sumDir(group) //Point
    average_dir = sum_direction / group.length()
    new_position = add(average_dir, branch.position())
    newBranch = new Branch(position=new_position,
                            direction=average_dir, parent=branch)
    newBranches.push(newBranch)

addBranchesToTree(tree, newBranches)
```

The first loop iterates over the leaves to find the closest branch and calculate a direction vector. The second loops groups and averages the directions to the paired leaves. The full Haskell code can be found in Tree.hs in the code listing at the end of the report.
4 Parallelization

The work in the first loop from the pseudocode is very easy to separate into independent parallel work. This is because each computation to find the closest branch to each leaf do not depend on the other leaves. After testing different variations/strategies to parallelize this first loop, I found that using `parMap rpar` provided the best performance and speed up with increasing cores.

```
parMap rpar (\x -> closestBranch x (branches tree)
  (min_dist tree) (max_dist tree)) unreached
```

Using just this strategy, the algorithm observed just about 3x speedup with 8 cores:

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.353</td>
</tr>
<tr>
<td>2</td>
<td>1.533</td>
</tr>
<tr>
<td>3</td>
<td>0.833</td>
</tr>
<tr>
<td>4</td>
<td>0.583</td>
</tr>
<tr>
<td>5</td>
<td>0.443</td>
</tr>
<tr>
<td>6</td>
<td>0.414</td>
</tr>
<tr>
<td>7</td>
<td>0.444</td>
</tr>
<tr>
<td>8</td>
<td>0.394</td>
</tr>
</tbody>
</table>

The parallelization provided pretty consistent speed increases with increasing cores. Using 3 cores as an example, the percentage of sparks converted
was ~90% as shown log output and threadscope analysis:

SPARKS: 127335 (112666 converted, 0 overflowed, 0 dud, 10604 GC’d, 4065 fizzled)

The work on 3 cores is distributed pretty evenly throughout with very little time spent in garbage collection or breaks for sequential computation. However, as the number of cores increase the program has trouble at the start of the program:

n=3

n=4
However, because the time spent in garbage collection and the short amount of time it spends in this state, the time saved in the parallelization vastly overshadows the time lost in those steps.

5 Code Listing

5.1 Main.h

```haskell
module Main where
import Graphics.Gloss
import Tree
import Render
import System.Environment(getArgs)
import TestPoints
import System.Exit

maxDistance :: Float
maxDistance = 60.0

minDistance :: Float
minDistance = 5.0;

simulationLoop :: Tree → IO ()
simulationLoop (DONE _) = putStrLn "Done"
simulationLoop tree = simulationLoop (nextBranch True True tree)

window :: Display
window = InWindow "Tree" (500, 500) (0, 0)

backgroundColor :: Color
backgroundColor = makeColor 255 255 255 255

startTree :: Tree
startTree = initialTree testPoints 500 maxDistance minDistance

main :: IO ()
main = do args <- getArgs
```

5
case args of
    [maxDist,minDist,speed] -> simulate window backgroundColor (
        read speed :: Int) (initialTree testPoints 500 (read maxDist :: Float) (read minDist :: Float)) treeAsPicture nextBranch
    [maxDist,minDist]  -> simulationLoop (initialTree testPoints 500 (read maxDist :: Float) (read minDist :: Float))
    _                -> putStrLn "Usage ./tree-exec maxDist minDist <
                                      simulationDisplaySpeed>" >> exitSuccess

5.2 Tree.hs

module Tree where
import Data.List
import Control.DeepSeq
import Data.List.Split
import Control.Parallel.Strategies

data Branch = Empty
             | Branch {
                      position :: Point,
                      parent :: Branch,
                      direction :: Point -- Vector Representation of
                                      direction
                  }

data Tree = DONE Tree
            | Tree {
                   leaves :: Leaves,
                   root :: Branch,
                   branches :: Branches,
                   max_dist :: Float,
                   min_dist :: Float,
                   window_size :: Float,
                   detected :: Bool
               }

Point Arithmetic Helpers
29 − }
30 add :: Point → Point → Point
31 add (x1, y1) (x2, y2) =
32 let
33    x = x1 + x2
34    y = y1 + y2
35 in (x, y)
36
37 sub :: Point → Point → Point
38 sub (x1, y1) (x2, y2) =
39 let
40    x = x1 - x2
41    y = y1 - y2
42 in (x, y)
43
44 vdiv :: Point → Float → Point
45 vdiv (x1, y1) a =
46 let
47    x = x1 / a
48    y = y1 / a
49 in (x, y)
50
51 vmult :: Point → Float → Point
52 vmult (x1, y1) a =
53 let
54    x = x1 / a
55    y = y1 / a
56 in (x, y)
57
58 distance :: Point → Point → Float
59 distance (x1,y1) (x2,y2) = let x' = x1 - x2
60      y' = y1 - y2
61     in sqrt (x'*x' + y'*y')
62
63 normalize :: Floating b => (b, b) → (b, b)
64 normalize (x,y) = let magnitude = sqrt ((x*x) + (y*y))
65        in
66              (x/magnitude, y/magnitude)
67
Tree, Branch, Leaf helpers

-- Convert Array of points to Leaves
pointsToLeaves :: [(Float, Float)] -> [Leaf]
pointsToLeaves arr = (parMap rseq (\(x,y) -> Leaf (x,y) False) arr)

-- Check if Branch is
notEmpty :: Branch -> Bool
notEmpty b = case b of
  Empty -> False
  otherwise -> True

-- Initialize a tree
initialTree :: [(Float, Float)] -> Float -> Float -> Float -> Tree
initialTree arr size max min = Tree {
  leaves = pointsToLeaves arr,
  root = root_init,
  branches = [root_init],
  max_dist = max,
  min_dist = min,
  window_size = size,
  detected = False
}

  where root_init = Branch {position=(0, -size /2), parent = Empty, direction = (0,1)}

addBranch :: Tree -> Branch -> Tree
addBranch tree branch = tree {branches= branch : (branches tree)}

addBranches :: Tree -> [Branch] -> Tree
addBranches tree b = tree {branches = b ++ (branches tree)}

detectLeaves :: Branch -> [Leaf] -> Float -> Bool
detectLeaves branch lvs maxDist = any (==True) (parMap rseq f lvs)

  where f None = False
\[ f (\text{Leaf } (x,y)) = \text{distance } (x,y) \]
\[ \text{(position branch)} < \text{maxDist} \]

\[
\begin{align*}
\text{closestBranch :: Leaf } & \rightarrow [\text{Branch}] \rightarrow \text{Float } \rightarrow \text{Float } \rightarrow (\text{Leaf, Branch}) \\
\text{closestBranch None } & = (\text{None, Empty}) \\
\text{closestBranch } (\text{Leaf } (x,y)) & \text{ br minDist maxDist } = \text{let closest } = \text{minimumBy} \\
& \begin{aligned}
& f \text{ br } \\
& \text{dis } = \text{distance } (\text{position closest } (x,y)) \\
& \text{newDir } = \text{sub } (x,y) (\text{position closest}) \\
& \text{normalized } = \text{normalize newDir} \\
& \text{in} \\
& \text{if } (\text{dis } >= \text{maxDist}) \\
& \text{then} \\
& (\text{(Leaf } (x,y) \text{ False), Empty}) \\
& \text{else} \\
& \text{if } (\text{dis } <= \text{minDist}) \\
& \text{then} \\
& (\text{(Leaf } (x,y) \text{ True), closest } \{\text{parent=} \text{closest, direction } = \text{normalized}\}) \\
& \text{else} \\
& (\text{(Leaf } (x,y) \text{ False), closest } \{\text{parent } = \text{closest, direction } = \text{normalized}\}) \\
\end{aligned}
\end{align*}
\]

\[
\begin{align*}
\text{averageDir :: } [\text{Branch}] & \rightarrow \text{Branch} \\
\text{averageDir } \text{brches } = \text{let ref } = (\text{head brches}) \\
& \begin{aligned}
& \text{-- sum } = \text{foldr1 add } (\text{parMap rseq } (\text{direction}) \\
& \text{branches}) \\
\end{aligned}
\end{align*}
\]
sumDir = (foldl' (\acc b -> add acc (direction b)) (direction (parent ref)) brches)
new_dir = normalize (vdiv sumDir (fromIntegral ((length brches))))
new_pos = add (position ref) new_dir

Branch { position=new_pos, parent = (parent ref), direction = new_dir }

calculateNewBranches :: [Branch] -> [Branch]
calculateNewBranches closests = let grouped = groupBy branchPos closests
                        in map averageDir grouped
                        where branchPos a b = (position a == position b)

step :: Tree -> Tree
step tree = let top = head (branches tree)
in (detectLeaves top (leaves tree) (max_dist tree)) of
    False -> addBranch tree (Branch {position=(add (position top) (direction top)), parent = top, direction = (direction top)})
    True -> tree {detected = True}

grow :: Tree -> Tree
grow tree = let unreached = filter (\(Leaf (_,_) reached) -> not reached) (leaves tree)
           (newLeaves, closests) = unzip (parMap rpar (\x ->
            closestBranch x (branches tree) (min_dist tree) (max_dist tree)) unreached)
           filteredClosests = filter notEmpty closests
           newBranches = calculateNewBranches filteredClosests
           in case newBranches of
              [] -> DONE tree
              _ -> addBranches (tree {leaves = newLeaves}) newBranches

eastBranch :: p1 -> p2 -> Tree -> Tree
nextBranch \_ \_ (DONE tree) = DONE tree

nextBranch \_ \_ tree = case (detected tree) of
                      False -> step tree
                      True  -> grow tree

5.3 Render.hs

module Render where
import Graphics.Gloss
import Tree

drawPoint :: Leaf -> Picture
drawPoint (Leaf (x,y) reached) = case reached of
                      False -> Color red (Translate x y (ThickCircle 2 2))
                      True  -> Blank

drawBranch :: Branch -> Picture
drawBranch b = case (parent b) of
                      Empty  -> Blank
                      otherwise -> let point = position b
                         parent_point = position (parent b)
in
                         line [point, parent_point]

treeAsPicture :: Tree -> Picture
treeAsPicture (DONE tree) = let branchPictures = map drawBranch (branches tree)
                           leafPictures = map drawPoint (leaves tree)
in
                           pictures ( leafPictures ++ branchPictures )
treeAsPicture tree = let branchPictures = map drawBranch (branches tree)
                      leafPictures = map drawPoint (leaves tree)
in
                      pictures (branchPictures ++ leafPictures)