```
1  ////////Houdini Vex Coding//////////////////////////
2
3  // Bram vd Berg || 4687949
4  // Jordy van Eijk || 4566297
5
6  ////////////////////////////////////////////////////
7
8
9  ////1.Data_Calculation
10
11
12 /////Distance_to_ exit///////////
13
14 //initialize variables
15 float D[];
16
17 // Wp vertical movement
18 float weg_ver = 5;
19
20 //location of exits
21 vector exit_1 = { -5,0,50 };
22 vector exit_2 = { -70,0,15 };
23 vector exit_3 = { -35,0,115 };
24 vector exit_4 = { 50,0,120 };
25
26 //location point to calculate
27 vector pcur = point(0, "P", @ptnum);
28
29 // exit 1
30 float dis_1_x = abs(exit_1[0] - pcur[0]);
31 float dis_1_y = abs(weg_ver * (exit_1[1] - pcur[1]));
32 float dis_1_z = abs(exit_1[2] - pcur[2]);
33 // totale total distance
34 float dis_total_1 = dis_1_x + dis_1_y + dis_1_z;
35 //add to list
36 append(D, dis_total_1);
37
38 // exit 2
39 float dis_2_x = abs(exit_2[0] - pcur[0]);
40 float dis_2_y = abs(weg_ver * (exit_2[1] - pcur[1]));
41 float dis_2_z = abs(exit_2[2] - pcur[2]);
42
43 float dis_total_2 = dis_2_x + dis_2_y + dis_2_z;
44 append(D, dis_total_2);
45
46 // exit 3
47 float dis_3_x = abs(exit_3[0] - pcur[0]);
48 float dis_3_y = abs(weg_ver * (exit_3[1] - pcur[1]));
49 float dis_3_z = abs(exit_3[2] - pcur[2]);
50
51 float dis_total_3 = dis_3_x + dis_3_y + dis_3_z;
52 append(D, dis_total_3);
53
54 // exit 4
55 float dis_4_x = abs(exit_4[0] - pcur[0]);
56 float dis_4_y = abs(weg_ver * (exit_4[1] - pcur[1]));
```

```
57  float dis_4_z = abs(exit_4[2] - pcur[2]);
58
59  float dis_total_4 = dis_4_x + dis_4_y + dis_4_z;
60  append(D, dis_total_4);
61
62  //sort results
63  int Dis_id[] = argsort(D);
64  float dis[] = sort(D);
65
66  // add atribute to point
67  f@Distance = dis[0];
68  f@Distance_id = Dis_id[0];
69
70
71  ////////////////////////////////////////////////////////////////////////////////
      //
72
73  ///////////Noise_Calculation///////////////
74
75  int noise_points = npoints(1);
76  vector pos_vox = point(0, "P", @ptnum);
77  float noise_total = 0;
78
79  for (int n = 0; n < noise_points; n++)
80  {
81      vector pos_noise = point(1, "P", n);
82      float noise_dis = distance(pos_vox, pos_noise);
83      float noise_p = (100 / (noise_dis * noise_dis));
84      noise_total = noise_total + noise_p;
85  }
86  // add attribute
87  f@noise = noise_total;
88
89
90  ////////////////////////////////////////////
91
92  ////////Wp_Calculation///////////////
93
94  //get the number of functions
95  int nfunc = npoints(1);
96  // initialize variables
97  float wp[];
98  float wpcurr;
99
100 // for each function calculate Wp and add this to wpcurr
101 for (int j = 0; j < nfunc; j++)
102 {
103
104     vector fcur = point(1, "factor", j);
105     float ratio = point(0, "ratio", @ptnum);
106     float Distance = point(0, "Distance", @ptnum);
107     float noise = point(0, "noise", @ptnum);
108
109     wpcurr = (pow(ratio, fcur[0]) + pow(Distance, fcur[2]) + pow(noise, fcur
          [1])) / 3;
110     append(wp, wpcurr);
```

```
111 }
112
113 // add wpcurr to point
114
115 setpointattrib(0, "WProduct", @ptnum, wp, "set");
116
117 ////////////////////////////////////////////
118
119
120
121 //////2.Growth_model
122
123 //////remove_unconnected_points//////////////////
124 int neigh = neighbourcount(0, @ptnum);
125 if (neigh <= 2)
126 {
127     removepoint(0, @ptnum);
128 }
129
130 ////////////////////////////////////////
131
132 /////Growth_agent(including: Stacking,Flattening & find new seed of neighbours
       ==0 but area isn't fulfilled)//////////////
133
134
135 // 0: initialize variables
136 int nfunc = npoints(1);
137 float squarness = chf("squarness");
138 float flatenning = chf("flatenning");
139 float stacking = chf("stacking");
140
141 //check if everyone is done growing
142 int done = 0;
143
144 // 1: iterate over each function (agent)
145 for (int j = 0; j < nfunc; j++)
146 {
147     // 1.1: check if the function(agent) have enough voxels
148     //current area
149     int cur_area = findattribvalcount(0, "point", "parent", j);
150     //desired area
151     int test = chi("voxel_size");
152     int des_area = point(1, "area", j);
153     int des_voxels = des_area / (test * test);
154     //compare current with desired
155     if (des_voxels > cur_area)
156     {
157         // 1.2: prepare children to find the boundary of function
158         //get the number of children
159         int num_child = findattribvalcount(0, "point", "parent", j);
160         //initialize the boundary list
161         int func_bounds_id[];
162         float func_bounds_wp[];
163         // 1.3: iterate over children
164         for (int i = 0; i < num_child; i++)
165         {
```

```
166             // 1.3.1: find the neighbours of the child
167             //retrieve the child id
168             int child_id = findattribval(0, "point", "parent", j, i);
169             //retrieve the child neighbours
170             int child_neighs[] = neighbours(0, child_id);
171 
172             //get the position of child
173             vector child_pos = point(0, "P", child_id);
174 
175             // 1.3.2: iterate over the neighbours of the child
176             foreach(int neigh; child_neighs)
177             {
178                 //1.3.2.1: check if they are occupied
179                 if (1 - inpointgroup(0, "occupied", neigh))
180                 {
181                     //get the neighbour position
182                     vector neigh_pos = point(0, "P", neigh);
183 
184 
185                     //find it in the currently existing boundary list
186                     int found = find(func_bounds_id, neigh);
187                     //if you did not find it add it to the list
188                     if (found < 0)
189                     {
190                         // add the id
191                         append(func_bounds_id, neigh);
192                         // add the wp
193                         float wp[] = point(0, "WProduct", neigh);
194 
195 
196                         //check if they are on the same level
197                         if (j == 1 || j == 13)
198                         {
199                             if (abs(neigh_pos.y - child_pos.y) < 0.001) wp[j] ↵
                    *= flatenning;
200                         }
201 
202                         //check if the voxel below is available
203                         if (des_area > 250)
204                         {
205                             if (neigh_pos.y < child_pos.y) wp[j] *= stacking;
206                         }
207 
208                         if (j == 9 || j == 10 || j == 11 || j == 12)
209                         {
210                             if ((neigh_pos.x - child_pos.x) < 0.1 ||          ↵
                    (neigh_pos.z - child_pos.z) < 0.1) wp[j] *= stacking;
211                         }
212                                                                               ↵
    ////////////////////////////////////////////////////////////////////////////↵
    ////////////////
213                         append(func_bounds_wp, wp[j]);
214 
215                     }
216                     //if you found it increase the wp by the "squarness" ratio
217                     else
```

```
218                     {
219                         //multiply by squarness factor
220                         func_bounds_wp[found] *= squarness;
221                     }
222                 }
223             }
224         }
225         // 1.3: sort the list of boundary voxels in decreasing order
226         int sorted_indicies[] = reverse(argsort(func_bounds_wp));
227         func_bounds_id = reorder(func_bounds_id, sorted_indicies);
228 
229         // 1.4: set the first voxel in the (sorted) boundary list as the new child (occupy it by the function)
230         setpointgroup(0, "occupied", func_bounds_id[0], 1, "set");
231         setpointattrib(0, "parent", func_bounds_id[0], j, "set");
232         ////werkt goed////////////////////////////////////////////////////////////////////////
            ////////////////////////////////////////////////////////////////////
233         //check if the function realy has enough voxels or if it is just out of neighbours
234         //use the initial placement code
235         if (len(func_bounds_wp) < 1)
236         {
237             float all_wp[];
238             int all_id[];
239             //iterate over voxels
240             for (int V = 0; V < @numpt; V++)
241             {
242                 if (inpointgroup(0, "occupied", V) < 1)
243                 {
244                     float V_wp[] = point(0, "WProduct", V);
245                     float V_wp_A = V_wp[j];
246                     append(all_wp, V_wp_A);
247                     append(all_id, V);
248                 }
249             }
250             //sorting
251             int sorted_indicies_2[] = reverse(argsort(all_wp));
252             all_id = reorder(all_id, sorted_indicies_2);
253             //choosing
254             int selection = all_id[0];
255             //occupying
256             setpointgroup(0, "occupied", selection, 1, "set");
257             setpointattrib(0, "parent", selection, j, "set");
258         }
259     }
260     //if the growth is done add one to the functions that are done
261     else
262     {
263         done++;
264     }
265     ////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////
266 }
267 
268 
```

```
269 ////werkt goed////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////
270 //if everyone is done growing move voxels down
271 if (done == nfunc)
272 {
273     for (int j = 0; j < nfunc; j++)
274     {
275         //find the number of children
276         int num_child = findattribvalcount(0, "point", "parent", j);
277 
278         //save all the y values of the children and there ids
279         float child_y[];
280         int child_ids[];
281 
282         //initialize the boundary list
283         int func_bounds_id[];
284         float func_bounds_wp[];
285 
286         // 1.3: iterate over children
287         for (int i = 0; i < num_child; i++)
288         {
289             // 1.3.1: find the neighbours of the child
290             //retrieve the child id
291             int child_id = findattribval(0, "point", "parent", j, i);
292             append(child_ids, child_id);
293             //retrieve the child neighbours
294             int child_neighs[] = neighbours(0, child_id);
295 
296             //get the position of child
297             vector child_pos = point(0, "P", child_id);
298             append(child_y, child_pos.y);
299 
300             // 1.3.2: iterate over the neighbours of the child
301             foreach(int neigh; child_neighs)
302             {
303                 //1.3.2.1: check if they are occupied
304                 if (1 - inpointgroup(0, "occupied", neigh))
305                 {
306                     //get the neighbour position
307                     vector neigh_pos = point(0, "P", neigh);
308 
309                     //find it in the currently existing boundary list
310                     int found = find(func_bounds_id, neigh);
311 
312                     //if you did not find it add it to the list
313                     if (found < 0)
314                     {
315                         if (neigh_pos.y < child_pos.y)
316                         {
317                             // add the id
318                             append(func_bounds_id, neigh);
319                             // add the wp
320                             float wp[] = point(0, "WProduct", neigh);
321                             append(func_bounds_wp, wp[j]);
322                         }
```

```
323                     }
324                 }
325             }
326         }
327         if (len(func_bounds_wp) > 0.9)
328         {
329             // 1.3: sort the list of boundary voxels in decreasing order
330             int sorted_indicies[] = reverse(argsort(func_bounds_wp));
331             func_bounds_id = reorder(func_bounds_id, sorted_indicies);
332 
333             int child_y_sort[] = reverse(argsort(child_y));
334             child_ids = reorder(child_ids, child_y_sort);
335 
336             // 1.4: set the first voxel in the (sorted) boundary list as the
                  new child (occupy it by the function)
337             //set the new point
338             setpointgroup(0, "occupied", func_bounds_id[0], 1, "set");
339             setpointattrib(0, "parent", func_bounds_id[0], j, "set");
340             //remove the old point
341             setpointgroup(0, "occupied", child_ids[0], 0, "set");
342             setpointattrib(0, "parent", child_ids[0], -1, "set");
343         }
344     }
345 }
346 ////////////////////////////////////
347 
348 ///////////color functions//////////
349 
350 int nfunc = npoints(1);
351 for (int i = 0; i < nfunc; i++)
352 {
353     if (@parent == i)
354     {
355         v@Cd.r = point(1, "Cdr", i);
356         v@Cd.g = point(1, "Cdg", i);
357         v@Cd.b = point(1, "Cdb", i);
358     }
359 }
360 
361 ////////////////////////////////////////////////
362 
363 
364 
365 //3. Elevatorschafts
366 
367 ////////find max distance from clusterpoint to elevator/////////
368 
369 int numclusters = chi("numclusters");
370 float maxdistance = chf("maxdistance");
371 float tempdis = 0.00001;
372 int nump = npoints(0);
373 for (int i = 0; i < nump; i++)
374 {
375     int centerp = point(0, "center", i);
376     if (centerp == 1)
377     {
```

```
378         int centerid = point(0, "cluster", i);
379         for (int j = 0; j < nump; j++)
380         {
381             int clusterid = point(0, "cluster", j);
382             if (centerid == clusterid)
383             {
384                 vector point = point(0, "P", j);
385                 vector center = point(0, "P", i);
386                 float clusterdis = distance(point, center);
387                 if (clusterdis > tempdis)
388                 {
389                     tempdis = clusterdis;
390                 }
391 
392             }
393         }
394     }
395 }
396 setdetailattrib(0, "maxdis", tempdis);
397 if (tempdis > maxdistance)
398 {
399     setdetailattrib(0, "enough_elevators", -2);
400 }
401 
402 ////////////////////////////////////////////////////////
403 
404 //extract centerpoints from list and add Yvalue(before Flattening) to
      clustered points////
405 
406 int centerp = inpointgroup(0, "averagepoints", @ptnum);
407 if (centerp == 1)
408 {
409     setpointattrib(0, "center", @ptnum,1);
410 }
411 i@heigth = int(@rest[1]);
412 
413 ////////////////////////////////////////////////////////
414 
415 ///calc_numfloors_per_cluster//////////
416 
417 int nump = npoints(0);
418 for (int i = 0; i < nump; i++)
419 {
420     int center = point(0, "center", i);
421     if (center == 1)
422     {
423         int maxy = 1;
424         int centerid = point(0, "cluster", i);
425         for (int j = 0; j < nump; j++)
426         {
427             int clusterid = point(0, "cluster", j);
428             if (centerid == clusterid)
429             {
430                 int yval = point(0, "heigth", j);
431                 if (yval > maxy)
432                 {
```

```
433                     maxy = yval;
434                 }
435             }
436         }
437         setdetailattrib(0, "maxy", maxy);
438         int floors = maxy / 3;
439         setpointattrib(0, "floors", i, floors);
440         if (floors < 2)
441         {
442             floors = floors + 1;
443         }
444         setpointattrib(0, "floors", i, floors);
445     }
446 }
447 
448 ////////////////////////////////////////////////////
449 
450 ///shaft_points/////////////////
451 
452 int floors = i@floors;
453 vector dir = set(0, 3, 0);
454 for (int f = 1; f < floors + 1; f++)
455 {
456     vector npos = v@P + f * dir;
457     int npt = addpoint(0, npos);
458     int elcl = @ptnum;
459         setpointattrib(0, "erik", npt, elcl);
460 
461 }
462 
463 ////////////////////////////////////////////
464 
465 //4.Corridors
466 
467 ////find closest voxels////////////
468 
469 int A = chi("A");
470 int B = chi("B");
471 int num_a_pts[] = findattribval(0, "point", "parent", A);
472 int num_b_pts[] = findattribval(0, "point", "parent", B);
473 
474 float dists[];
475 int starts[];
476 int ends[];
477 foreach(int a; num_a_pts)
478 {
479     vector a_pos = point(0, "P", a);
480     foreach(int b; num_b_pts)
481     {
482         vector b_pos = point(0, "P", b);
483         float dist = distance(a_pos, b_pos);
484         append(dists, dist);
485         append(starts, a);
486         append(ends, b);
487     }
488 }
```

```
489
490 int indicies_sorted[] = argsort(dists);
491 int start_ordered[] = reorder(starts, indicies_sorted);
492 int end_ordered[] = reorder(ends, indicies_sorted);
493
494 int check_A = start_ordered[0];
495 int check_B = end_ordered[0];
496
497 setpointgroup(0, "ends", start_ordered[0], 1, "set");
498 setpointgroup(0, "ends", end_ordered[0], 1, "set");
499
500 ////////////////////////////////////////////////////////////
501
502 ////find average elevator points/////////
503
504 int elevators = npoints(0);
505 float elevator_x;
506 float elevator_z;
507
508 for (int b = 0; b < elevators; b++)
509 {
510     vector loc = point(0, "P", b);
511     elevator_x = elevator_x + loc.x;
512     elevator_z = elevator_z + loc.z;
513 }
514
515 elevator_x = elevator_x / elevators;
516 elevator_z = elevator_z / elevators;
517
518 f@elevator_x = elevator_x;
519 f@elevator_z = elevator_z;
520
521 ////////////////////////////////////////
522
523 ///place elevators////////
524
525 int elevators = npoints(1);
526
527 for (int b = 0; b < elevators; b++)
528 {
529     vector loc = point(1, "P", b);
530
531     int eleva = nearpoint(0, loc);
532     vector loc_1 = point(0, "P", eleva);
533
534     if (loc_1.y == 0)
535     {
536         setpointattrib(0, "parent", eleva, 15, "set");
537         setpointgroup(0, "occupied", eleva, 1, "set");
538     }
539     else
540     {
541         setpointattrib(0, "parent", eleva, 15, "set");
542         setpointgroup(0, "occupied", eleva, 1, "set");
543     }
544 }
```

```
545
546 ////////////////////////////////////////////
547
548 //place groundfloor hallways///
549
550 //get the centre of all the elevators
551 float el_x = detail(1, "elevator_x", 0);
552 float el_z = detail(1, "elevator_z", 0);
553 vector ava_ele = set(el_x, 0, el_z);
554
555 //get the max and min of the x and z coordinates
556 float max_x = detail(0, "max_x", 0);
557 float min_x = detail(0, "min_x", 0);
558 float max_z = detail(0, "max_z", 0);
559 float min_z = detail(0, "min_z", 0);
560
561
562
563 //get the number of elevators
564 int num_elevators = findattribvalcount(0, "point", "parent", 15);
565
566 for (int b = 0; b < num_elevators; b++)
567 {
568     //find the id and the location of the elevator
569     int elevator_id = findattribval(0, "point", "parent", 15, b);
570     vector loc_1 = point(0, "P", elevator_id);
571     //calculate the distance from the centre to the elevator
572     float dis_ava_ele = distance(ava_ele, loc_1);
573     float max_dis = 0;
574
575     //check in witch quarter of a circle the elevator is and what the distance ↵
          to the edge of the building is
576     int option;
577     float dis_x;
578     float dis_z;
579
580     if (loc_1.x > el_x)
581     {
582         if (loc_1.z > el_z)
583         {
584             option = 1;
585             dis_x = abs(max_x - loc_1.x);
586             dis_z = abs(max_z - loc_1.z);
587         }
588         if (loc_1.z < el_z)
589         {
590             option = 2;
591             dis_x = abs(max_x - loc_1.x);
592             dis_z = abs(min_z - loc_1.z);
593         }
594     }
595
596     if (loc_1.x < el_x)
597     {
598
599         if (loc_1.z > el_z)
```

```
600         {
601             option = 3;
602             dis_x = abs(min_x - loc_1.x);
603             dis_z = abs(max_z - loc_1.z);
604         }
605 
606         if (loc_1.z < el_z)
607         {
608             option = 4;
609             dis_x = abs(min_x - loc_1.x);
610             dis_z = abs(min_z - loc_1.z);
611         }
612     }
613 
614     //itterate over all the points
615     for (int a = 0; a < @numpt; a++)
616     {
617         vector loc_2 = point(0, "P", a);
618         float dis_ava_point = distance(ava_ele, loc_2);
619         int option_v2;
620         if (loc_2.x > el_x)
621         {
622             if (loc_2.z > el_z) option_v2 = 1;
623             if (loc_2.z < el_z) option_v2 = 2;
624         }
625         if (loc_2.x < el_x)
626         {
627             if (loc_2.z > el_z) option_v2 = 3;
628             if (loc_2.z < el_z) option_v2 = 4;
629         }
630         if (option == option_v2)
631         {
632             if (dis_x > dis_z)
633             {
634                 if (dis_ava_point > dis_ava_ele)
635                 {
636                     if (abs(loc_2.x - loc_1.x) < 0.01)
637                     {
638                         if (loc_2.y == 0)
639                         {
640                             setpointattrib(0, "parent", a, 16, "set");
641                         }
642                     }
643                 }
644             }
645 
646             if (dis_x < dis_z)
647             {
648                 if (dis_ava_point > dis_ava_ele)
649                 {
650                     if (abs(loc_2.z - loc_1.z) < 0.01)
651                     {
652                         if (loc_2.y == 0)
653                         {
654                             setpointattrib(0, "parent", a, 16, "set");
655                         }
```

```
656                     }
657                 }
658             }
659         }
660     }
661 }
662
663 ////////////////////////////////////////////////
664
665 ///XXXXXXXXXXXX/////
666
667 int points = npoints(1);
668
669 for (int p = 0; p < points; p++)
670 {
671     vector loc = point(1, "P", p);
672     int new_point = nearpoint(0, loc);
673
674     setpointattrib(0, "parent", new_point, 14, "set");
675     setpointgroup(0, "occupied", new_point, 1, "set");
676 }
677
678 for (int q = 0; q < @numpt; q++)
679 {
680     int parent = point(0, "parent", q);
681     if (parent == 15 || parent == 16)
682     {
683         setpointattrib(0, "parent", q, 14, "set");
684     }
685 }
686
687 ////////////////////////////////////////////////
688
689 //5.Roofgardens
690
691 ///Calculate_Distance_to_the_closest_voxel_above_it///
692 int nump = npoints(0);
693 vector hitp;
694 float u;
695 float v;
696 for (int i = 0; i < nump; i++)
697 {
698
699     vector origin = point(0, "P", i) + set(0, 1.55, 0);
700     vector direction = set(0, 100, 0);
701     int ind_1 = intersect(1, origin, direction, hitp, u, v);
702     float distance = distance(origin, hitp);
703     setpointattrib(0, "hitdistance", i, distance);
704
705
706     if (distance > 4 || distance == 0)
707     {
708         setpointgroup(0, "roofvoxel", i, 1);
709     }
710
711 }
```

```
712 ////////////////////////////////////////////////////
713
714 ///determine_topfaces///////
715 int points = npoints(0);
716 for (int i = 5; i < points; i = i + 6)
717 {
718     setprimgroup(0, "Topprim", i, 1);
719 }
720
721 ////////////////////////////////////////////////////
722
723
724 //6.Facade
725
726 /////add_parent_id_to_prim////
727 int numprim = nprimitives(0);
728 for (int i = 0; i < numprim; i++)
729 {
730     int primpoint = primpoint(0, i, 0);
731     int primfunc = point(0, "parent", primpoint);
732     setprimattrib(0, "parent", i, primfunc);
733 }
734
735
736 ////////////////////////////////////////////////////
737
738 //set_prim_to_coresponding_facade_template/////
739 int primnum = nprimitives(0);
740
741 for (int i = 0; i < primnum; i++)
742 {
743     int parent = primattrib(0, "parent", i, 0);
744     if (parent == 9 || parent == 10 || parent == 11)
745     {
746         setprimgroup(0, "Housing", i, 1);
747     }
748     if (parent == 1)
749     {
750         setprimgroup(0, "Parking", i, 1);
751     }
752     if (parent == 2 || parent == 8)
753     {
754         setprimgroup(0, "Shops", i, 1);
755     }
756     if (parent == 3 || parent == 4 || parent == 12)
757     {
758         setprimgroup(0, "Glass", i, 1);
759     }
760     if (parent == 5 || parent == 6)
761     {
762         setprimgroup(0, "Comunity", i, 1);
763     }
764     if (parent == 7 || parent == 13)
765     {
766         setprimgroup(0, "Horeca", i, 1);
767     }
```

```
768     if (parent == 0)
769     {
770         setprimgroup(0, "Workspace", i, 1);
771     }
772     if (parent == 17)
773     {
774         setprimgroup(0, "Emptytop", i, 1);
775     }
776     if (parent == 66)
777     {
778         setprimgroup(0, "Bottom", i, 1);
779     }
780 }
781
782
```