```
////////Houdini Vex Coding/////////////////////////

// Bram vd Berg || 4687949
// Jordy van Eijk || 4566297

///////////////////////////////////////////////////


////1.Data_Calculation


/////Distance_to_ exit///////////

//initialize variables
float D[];

// Wp vertical movement
float weg_ver = 5;

//location of exits
vector exit_1 = { -5,0,50 };
vector exit_2 = { -70,0,15 };
vector exit_3 = { -35,0,115 };
vector exit_4 = { 50,0,120 };

//location point to calculate
vector pcur = point(0, "P", @ptnum);

// exit 1
float dis_1_x = abs(exit_1[0] - pcur[0]);
float dis_1_y = abs(weg_ver * (exit_1[1] - pcur[1]));
float dis_1_z = abs(exit_1[2] - pcur[2]);
// totale total distance
float dis_total_1 = dis_1_x + dis_1_y + dis_1_z;
//add to list
append(D, dis_total_1);

// exit 2
float dis_2_x = abs(exit_2[0] - pcur[0]);
float dis_2_y = abs(weg_ver * (exit_2[1] - pcur[1]));
float dis_2_z = abs(exit_2[2] - pcur[2]);

float dis_total_2 = dis_2_x + dis_2_y + dis_2_z;
append(D, dis_total_2);

// exit 3
float dis_3_x = abs(exit_3[0] - pcur[0]);
float dis_3_y = abs(weg_ver * (exit_3[1] - pcur[1]));
float dis_3_z = abs(exit_3[2] - pcur[2]);

float dis_total_3 = dis_3_x + dis_3_y + dis_3_z;
append(D, dis_total_3);

// exit 4
float dis_4_x = abs(exit_4[0] - pcur[0]);
float dis_4_y = abs(weg_ver * (exit_4[1] - pcur[1]));
float dis_4_z = abs(exit_4[2] - pcur[2]);

float dis_total_4 = dis_4_x + dis_4_y + dis_4_z;
append(D, dis_total_4);

//sort results
int Dis_id[] = argsort(D);
float dis[] = sort(D);

// add atribute to point
f@Distance = dis[0];
f@Distance_id = Dis_id[0];


////////////////////////////////////////////////////////////////////////////////

///////////Noise_Calculation////////////////

int noise_points = npoints(1);
vector pos_vox = point(0, "P", @ptnum);
float noise_total = 0;

for (int n = 0; n < noise_points; n++)
{
    vector pos_noise = point(1, "P", n);
    float noise_dis = distance(pos_vox, pos_noise);
    float noise_p = (100 / (noise_dis * noise_dis));
    noise_total = noise_total + noise_p;
}
// add attribute
f@noise = noise_total;


//////////////////////////////////////////////

/////////Wp_Calculation////////////////

//get the number of functions
int nfunc = npoints(1);
// initialize variables
float wp[];
float wpcurr;

// for each function calculate Wp and add this to wpcurr
for (int j = 0; j < nfunc; j++)
{

    vector fcur = point(1, "factor", j);
    float ratio = point(0, "ratio", @ptnum);
    float Distance = point(0, "Distance", @ptnum);
    float noise = point(0, "noise", @ptnum);

    wpcurr = (pow(ratio, fcur[0]) + pow(Distance, fcur[2]) + pow(noise, fcur[1])) / 3;
    append(wp, wpcurr);
```

```
}

// add wpcurr to point

setpointattrib(0, "WProduct", @ptnum, wp, "set");

//////////////////////////////////////////////



//////2.Growth_model

//////remove_unconnected_points//////////////////
int neigh = neighbourcount(0, @ptnum);
if (neigh <= 2)
{
    removepoint(0, @ptnum);
}

/////////////////////////////////////////

/////Growth_agent(including: Stacking,Flattening & find new seed of neighbours ==0 but area isn't fulfilled)////////////


// 0: initialize variables
int nfunc = npoints(1);
float squarness = chf("squarness");
float flatenning = chf("flatenning");
float stacking = chf("stacking");

//check if everyone is done growing
int done = 0;

// 1: iterate over each function (agent)
for (int j = 0; j < nfunc; j++)
{
    // 1.1: check if the function(agent) have enough voxels
    //current area
    int cur_area = findattribvalcount(0, "point", "parent", j);
    //desired area
    int test = chi("voxel_size");
    int des_area = point(1, "area", j);
    int des_voxels = des_area / (test * test);
    //compare current with desired
    if (des_voxels > cur_area)
    {
        // 1.2: prepare children to find the boundary of function
        //get the number of children
        int num_child = findattribvalcount(0, "point", "parent", j);
        //initialize the boundary list
        int func_bounds_id[];
        float func_bounds_wp[];
        // 1.3: iterate over children
        for (int i = 0; i < num_child; i++)
        {
            // 1.3.1: find the neighbours of the child
            //retrieve the child id
            int child_id = findattribval(0, "point", "parent", j, i);
            //retrieve the child neighbours
            int child_neighs[] = neighbours(0, child_id);

            //get the position of child
            vector child_pos = point(0, "P", child_id);

            // 1.3.2: iterate over the neighbours of the child
            foreach(int neigh; child_neighs)
            {
                //1.3.2.1: check if they are occupied
                if (1 - inpointgroup(0, "occupied", neigh))
                {
                    //get the neighbour position
                    vector neigh_pos = point(0, "P", neigh);


                    //find it in the currently existing boundary list
                    int found = find(func_bounds_id, neigh);
                    //if you did not find it add it to the list
                    if (found < 0)
                    {
                        // add the id
                        append(func_bounds_id, neigh);
                        // add the wp
                        float wp[] = point(0, "WProduct", neigh);


                        //check if they are on the same level
                        if (j == 1 || j == 13)
                        {
                            if (abs(neigh_pos.y - child_pos.y) < 0.001) wp[j] *= flatenning;
                        }

                        //check if the voxel below is available
                        if (des_area > 250)
                        {
                            if (neigh_pos.y < child_pos.y) wp[j] *= stacking;
                        }

                        if (j == 9 || j == 10 || j == 11 || j == 12)
                        {
                            if ((neigh_pos.x - child_pos.x) < 0.1 || (neigh_pos.z - child_pos.z) < 0.1) wp[j] *= stacking;
                        }
                        //////////////////////////////////////////////////////////////////////////////////////////////
                        append(func_bounds_wp, wp[j]);

                    }
                    //if you found it increase the wp by the "squarness" ratio
                    else
                    {
                        //multiply by squarness factor
                        func_bounds_wp[found] *= squarness;
                    }
                }
            }
        }
        // 1.3: sort the list of boundary voxels in decreasing order
        int sorted_indicies[] = reverse(argsort(func_bounds_wp));
        func_bounds_id = reorder(func_bounds_id, sorted_indicies);

        // 1.4: set the first voxel in the (sorted) boundary list as the new child (occupy it by the function)
        setpointgroup(0, "occupied", func_bounds_id[0], 1, "set");
        setpointattrib(0, "parent", func_bounds_id[0], j, "set");
        ////werkt goed/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        //check if the function realy has enough voxels or if it is just out of neighbours
        //use the initial placement code
        if (len(func_bounds_wp) < 1)
        {
            float all_wp[];
            int all_id[];
            //iterate over voxels
            for (int V = 0; V < @numpt; V++)
            {
                if (inpointgroup(0, "occupied", V) < 1)
                {
                    float V_wp[] = point(0, "WProduct", V);
                    float V_wp_A = V_wp[j];
                    append(all_wp, V_wp_A);
                    append(all_id, V);
                }
            }
            //sorting
            int sorted_indicies_2[] = reverse(argsort(all_wp));
            all_id = reorder(all_id, sorted_indicies_2);
            //choosing
            int selection = all_id[0];
            //occupying
            setpointgroup(0, "occupied", selection, 1, "set");
            setpointattrib(0, "parent", selection, j, "set");
        }
    }
    //if the growth is done add one to the functions that are done
    else
    {
        done++;
    }
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
}
```

```
////werkt
  goed////////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////
//if everyone is done growing move voxels down
if (done == nfunc)
{
    for (int j = 0; j < nfunc; j++)
    {
        //find the number of children
        int num_child = findattribvalcount(0, "point", "parent", j);

        //save all the y values of the children and there ids
        float child_y[];
        int child_ids[];

        //initialize the boundary list
        int func_bounds_id[];
        float func_bounds_wp[];

        // 1.3: iterate over children
        for (int i = 0; i < num_child; i++)
        {
            // 1.3.1: find the neighbours of the child
            //retrieve the child id
            int child_id = findattribval(0, "point", "parent", j, i);
            append(child_ids, child_id);
            //retrieve the child neighbours
            int child_neighs[] = neighbours(0, child_id);

            //get the position of child
            vector child_pos = point(0, "P", child_id);
            append(child_y, child_pos.y);

            // 1.3.2: iterate over the neighbours of the child
            foreach(int neigh; child_neighs)
            {
                //1.3.2.1: check if they are occupied
                if (1 - inpointgroup(0, "occupied", neigh))
                {
                    //get the neighbour position
                    vector neigh_pos = point(0, "P", neigh);

                    //find it in the currently existing boundary list
                    int found = find(func_bounds_id, neigh);

                    //if you did not find it add it to the list
                    if (found < 0)
                    {
                        if (neigh_pos.y < child_pos.y)
                        {
                            // add the id
                            append(func_bounds_id, neigh);
                            // add the wp
                            float wp[] = point(0, "WProduct", neigh);
                            append(func_bounds_wp, wp[j]);
                        }
                    }
                }
            }
        }
        if (len(func_bounds_wp) > 0.9)
        {
            // 1.3: sort the list of boundary voxels in decreasing order
            int sorted_indicies[] = reverse(argsort(func_bounds_wp));
            func_bounds_id = reorder(func_bounds_id, sorted_indicies);

            int child_y_sort[] = reverse(argsort(child_y));
            child_ids = reorder(child_ids, child_y_sort);

            // 1.4: set the first voxel in the (sorted) boundary list as the new child (occupy it by the function)
            //set the new point
            setpointgroup(0, "occupied", func_bounds_id[0], 1, "set");
            setpointattrib(0, "parent", func_bounds_id[0], j, "set");
            //remove the old point
            setpointgroup(0, "occupied", child_ids[0], 0, "set");
            setpointattrib(0, "parent", child_ids[0], -1, "set");
        }
    }
}
/////////////////////////////////////

///////////color functions//////////

int nfunc = npoints(1);
for (int i = 0; i < nfunc; i++)
{
    if (@parent == i)
    {
        v@Cd.r = point(1, "Cdr", i);
        v@Cd.g = point(1, "Cdg", i);
        v@Cd.b = point(1, "Cdb", i);
    }
}

/////////////////////////////////////////////////



//3. Elevatorschafts

////////find max distance from clusterpoint to elevator////////

int numclusters = chi("numclusters");
float maxdistance = chf("maxdistance");
float tempdis = 0.00001;
int nump = npoints(0);
for (int i = 0; i < nump; i++)
{
    int centerp = point(0, "center", i);
    if (centerp == 1)
    {
        int centerid = point(0, "cluster", i);
        for (int j = 0; j < nump; j++)
        {
            int clusterid = point(0, "cluster", j);
            if (centerid == clusterid)
            {
                vector point = point(0, "P", j);
                vector center = point(0, "P", i);
                float clusterdis = distance(point, center);
                if (clusterdis > tempdis)
                {
                    tempdis = clusterdis;
                }

            }
        }
    }
}
setdetailattrib(0, "maxdis", tempdis);
if (tempdis > maxdistance)
{
    setdetailattrib(0, "enough_elevators", -2);
}

///////////////////////////////////////////////////////////

//extract centerpoints from list and add Yvalue(before Flattening) to clustered points////

int centerp = inpointgroup(0, "averagepoints", @ptnum);
if (centerp == 1)
{
    setpointattrib(0, "center", @ptnum,1);
}
i@heigth = int(@rest[1]);

////////////////////////////////////////////////////////

///calc_numfloors_per_cluster/////////

int nump = npoints(0);
for (int i = 0; i < nump; i++)
{
    int center = point(0, "center", i);
    if (center == 1)
    {
        int maxy = 1;
        int centerid = point(0, "cluster", i);
        for (int j = 0; j < nump; j++)
        {
            int clusterid = point(0, "cluster", j);
            if (centerid == clusterid)
            {
                int yval = point(0, "heigth", j);
                if (yval > maxy)
                {
                    maxy = yval;
                }
            }
        }
        setdetailattrib(0, "maxy", maxy);
        int floors = maxy / 3;
        setpointattrib(0, "floors", i, floors);
        if (floors < 2)
        {
            floors = floors + 1;
        }
        setpointattrib(0, "floors", i, floors);
    }
}

///////////////////////////////////////////////////////

///shaft_points////////////////

int floors = i@floors;
vector dir = set(0, 3, 0);
for (int f = 1; f < floors + 1; f++)
{
    vector npos = v@P + f * dir;
    int npt = addpoint(0, npos);
    int elcl = @ptnum;
        setpointattrib(0, "erik", npt, elcl);

}

////////////////////////////////////////////

//4.Corridors

////find closest voxels///////////

int A = chi("A");
int B = chi("B");
int num_a_pts[] = findattribval(0, "point", "parent", A);
int num_b_pts[] = findattribval(0, "point", "parent", B);

float dists[];
int starts[];
int ends[];
foreach(int a; num_a_pts)
{
    vector a_pos = point(0, "P", a);
    foreach(int b; num_b_pts)
    {
        vector b_pos = point(0, "P", b);
        float dist = distance(a_pos, b_pos);
        append(dists, dist);
        append(starts, a);
        append(ends, b);
    }
}
```

```

int indicies_sorted[] = argsort(dists);
int start_ordered[] = reorder(starts, indicies_sorted);
int end_ordered[] = reorder(ends, indicies_sorted);

int check_A = start_ordered[0];
int check_B = end_ordered[0];

setpointgroup(0, "ends", start_ordered[0], 1, "set");
setpointgroup(0, "ends", end_ordered[0], 1, "set");

///////////////////////////////////////////////////////////////

////find average elevator points/////////

int elevators = npoints(0);
float elevator_x;
float elevator_z;

for (int b = 0; b < elevators; b++)
{
    vector loc = point(0, "P", b);
    elevator_x = elevator_x + loc.x;
    elevator_z = elevator_z + loc.z;
}

elevator_x = elevator_x / elevators;
elevator_z = elevator_z / elevators;

f@elevator_x = elevator_x;
f@elevator_z = elevator_z;

///////////////////////////////////////////

///place elevators/////////

int elevators = npoints(1);

for (int b = 0; b < elevators; b++)
{
    vector loc = point(1, "P", b);

    int eleva = nearpoint(0, loc);
    vector loc_1 = point(0, "P", eleva);

    if (loc_1.y == 0)
    {
        setpointattrib(0, "parent", eleva, 15, "set");
        setpointgroup(0, "occupied", eleva, 1, "set");
    }
    else
    {
        setpointattrib(0, "parent", eleva, 15, "set");
        setpointgroup(0, "occupied", eleva, 1, "set");
    }
}
```

```

//////////////////////////////////////////////

//place groundfloor hallways///

//get the centre of all the elevators
float el_x = detail(1, "elevator_x", 0);
float el_z = detail(1, "elevator_z", 0);
vector ava_ele = set(el_x, 0, el_z);

//get the max and min of the x and z coordinates
float max_x = detail(0, "max_x", 0);
float min_x = detail(0, "min_x", 0);
float max_z = detail(0, "max_z", 0);
float min_z = detail(0, "min_z", 0);



//get the number of elevators
int num_elevators = findattribvalcount(0, "point", "parent", 15);

for (int b = 0; b < num_elevators; b++)
{
    //find the id and the location of the elevator
    int elevator_id = findattribval(0, "point", "parent", 15, b);
    vector loc_1 = point(0, "P", elevator_id);
    //calculate the distance from the centre to the elevator
    float dis_ava_ele = distance(ava_ele, loc_1);
    float max_dis = 0;

    //check in witch quarter of a circle the elevator is and what the distance to the edge of the building is
    int option;
    float dis_x;
    float dis_z;

    if (loc_1.x > el_x)
    {
        if (loc_1.z > el_z)
        {
            option = 1;
            dis_x = abs(max_x - loc_1.x);
            dis_z = abs(max_z - loc_1.z);
        }
        if (loc_1.z < el_z)
        {
            option = 2;
            dis_x = abs(max_x - loc_1.x);
            dis_z = abs(min_z - loc_1.z);
        }
    }

    if (loc_1.x < el_x)
    {

        if (loc_1.z > el_z)
        {
            option = 3;
            dis_x = abs(min_x - loc_1.x);
            dis_z = abs(max_z - loc_1.z);
        }

        if (loc_1.z < el_z)
        {
            option = 4;
            dis_x = abs(min_x - loc_1.x);
            dis_z = abs(min_z - loc_1.z);
        }
    }

    //itterate over all the points
    for (int a = 0; a < @numpt; a++)
    {
        vector loc_2 = point(0, "P", a);
        float dis_ava_point = distance(ava_ele, loc_2);
        int option_v2;
        if (loc_2.x > el_x)
        {
            if (loc_2.z > el_z) option_v2 = 1;
            if (loc_2.z < el_z) option_v2 = 2;
        }
        if (loc_2.x < el_x)
        {
            if (loc_2.z > el_z) option_v2 = 3;
            if (loc_2.z < el_z) option_v2 = 4;
        }
        if (option == option_v2)
        {
            if (dis_x > dis_z)
            {
                if (dis_ava_point > dis_ava_ele)
                {
                    if (abs(loc_2.x - loc_1.x) < 0.01)
                    {
                        if (loc_2.y == 0)
                        {
                            setpointattrib(0, "parent", a, 16, "set");
                        }
                    }
                }
            }

            if (dis_x < dis_z)
            {
                if (dis_ava_point > dis_ava_ele)
                {
                    if (abs(loc_2.z - loc_1.z) < 0.01)
                    {
                        if (loc_2.y == 0)
                        {
                            setpointattrib(0, "parent", a, 16, "set");
                        }
                    }
                }
            }
        }
    }
}

//////////////////////////////////////////////////

///XXXXXXXXXXXXX/////

int points = npoints(1);

for (int p = 0; p < points; p++)
{
    vector loc = point(1, "P", p);
    int new_point = nearpoint(0, loc);

    setpointattrib(0, "parent", new_point, 14, "set");
    setpointgroup(0, "occupied", new_point, 1, "set");
}

for (int q = 0; q < @numpt; q++)
{
    int parent = point(0, "parent", q);
    if (parent == 15 || parent == 16)
    {
        setpointattrib(0, "parent", q, 14, "set");
    }
}

/////////////////////////////////////////////////

//5.Roofgardens

///Calculate_Distance_to_the_closest_voxel_above_it///
int nump = npoints(0);
vector hitp;
float u;
float v;
for (int i = 0; i < nump; i++)
{

    vector origin = point(0, "P", i) + set(0, 1.55, 0);
    vector direction = set(0, 100, 0);
    int ind_1 = intersect(1, origin, direction, hitp, u, v);
    float distance = distance(origin, hitp);
    setpointattrib(0, "hitdistance", i, distance);


    if (distance > 4 || distance == 0)
    {
        setpointgroup(0, "roofvoxel", i, 1);
    }

}
```

```
/////////////////////////////////////////////////////

///determine_topfaces///////
int points = npoints(0);
for (int i = 5; i < points; i = i + 6)
{
    setprimgroup(0, "Topprim", i, 1);
}

//////////////////////////////////////////////////


//6.Facade

/////add_parent_id_to_prim////
int numprim = nprimitives(0);
for (int i = 0; i < numprim; i++)
{
    int primpoint = primpoint(0, i, 0);
    int primfunc = point(0, "parent", primpoint);
    setprimattrib(0, "parent", i, primfunc);
}


/////////////////////////////////////////////////

//set_prim_to_coresponding_facade_template/////
int primnum = nprimitives(0);

for (int i = 0; i < primnum; i++)
{
    int parent = primattrib(0, "parent", i, 0);
    if (parent == 9 || parent == 10 || parent == 11)
    {
        setprimgroup(0, "Housing", i, 1);
    }
    if (parent == 1)
    {
        setprimgroup(0, "Parking", i, 1);
    }
    if (parent == 2 || parent == 8)
    {
        setprimgroup(0, "Shops", i, 1);
    }
    if (parent == 3 || parent == 4 || parent == 12)
    {
        setprimgroup(0, "Glass", i, 1);
    }
    if (parent == 5 || parent == 6)
    {
        setprimgroup(0, "Comunity", i, 1);
    }
    if (parent == 7 || parent == 13)
    {
        setprimgroup(0, "Horeca", i, 1);
    }
```

```cpp
    if (parent == 0)
    {
        setprimgroup(0, "Workspace", i, 1);
    }
    if (parent == 17)
    {
        setprimgroup(0, "Emptytop", i, 1);
    }
    if (parent == 66)
    {
        setprimgroup(0, "Bottom", i, 1);
    }
}
```