Solar envelope

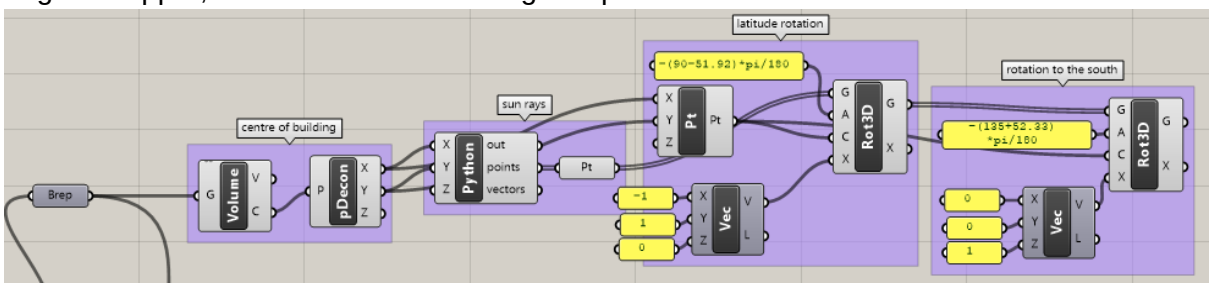
In order to reduce the amount of shadow cast on the surrounding buildings by our own building, a solar envelope was required to be computed. Certain voxels needed to be removed so that the windows of surrounding buildings can receive a sufficient amount of solar energy.

Simplified solar model

To calculate the solar rays during the day in this area, a solar model was needed.
Firstly, we looked at the ready-made solarpath from ladybug. This solar path was very detailed and graphically very heavy. The only two options were to choose 1 specified hour during a day during the year. This didn't give a good representation of the sun rays cast upon the neighbourhood during different seasons. The other option was to select two days as a start and end point, and the ladybug would calculate the solar rays of all the hours in between these days. When calculating for half a year (which would give the best representation), grasshopper couldn't handle all these calculations and shut down.

During a lecture, we got an explanation how to make your own simplified, but representative sunpath and the decision was made to try this out ourselves.

In grasshopper, we inserted the following components:



First the coordinates of the centre of our building were calculated to centre the part of the sphere that will represent our solarpath.

Secondly the python script was made to code the sun vectors and their origins. As inputs, the code gets the coordinates of the centre of our building.

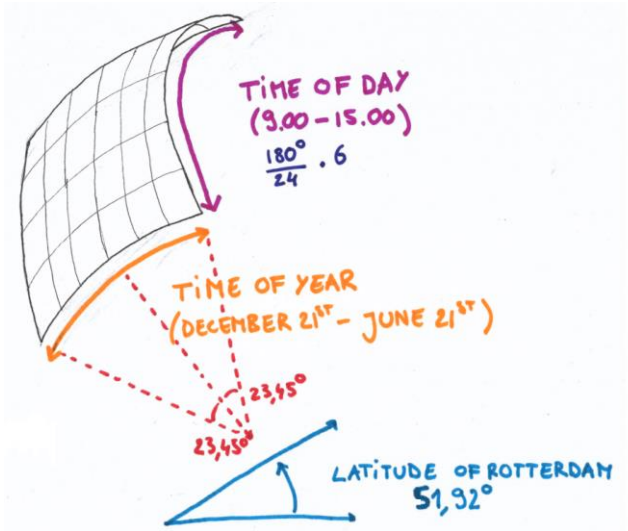
```
__author__ = "felis"
__version__ = "2018.11.30"

import rhinoscriptsyntax as rs
import Rhino.Geometry as rg
import math as math

pointslist = []
vectorlist = []
R=200000
hours = 6
months = 6
for theta in rs.frange(0, (2*math.pi / 24) * (hours), math.pi/12):
    for phi in rs.frange(-23.45*math.pi/180, 23.45*math.pi/180, (46.9*math.pi/180)/months):
        x = X + R * math.cos(phi) * math.cos(theta)
        y = Y + R * math.cos(phi) * math.sin(theta)
        z = Z + R * math.sin(phi)
        point = rg.Point3d(x, y, z)
        pointslist.append(point)
        vector = rg.Vector3d(X-x, Y-y, Z-z)
        vectorlist.append(vector)
        
points = pointslist
vectors = vectorlist
```

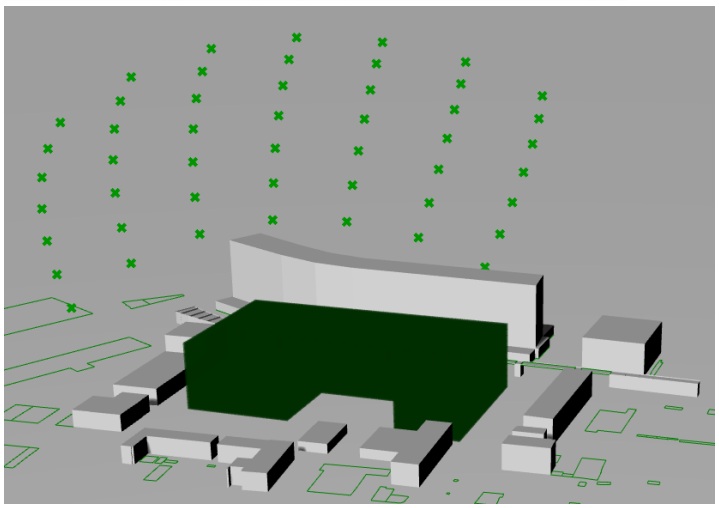
The decision was made to calculate the sun vectors for 6 hours a day, so from 09.00 o'clock till 15.00 o'clock. This would give a wide enough span for the neighbours to receive sunlight. The idea behind the script was to code a sphere that is horizontally devided by 24. This sphere, divided by 24, represent the rotation of the earth and therefore the path of the sun during 1 day. In the first for-loop, only 6 hours however are calculated, giving only the part of the sphere that we want to use for the solar path.
Because the axis of the sun tilts 23,45 degrees towards (in the summer) or away from the sun (in the winter), the height of the sun will vary between these angles during the the year. To get the full difference of the angle of the sun rays on earth, the sun path will be calculated from the shortest day (21st of December) to the longest day (21st of June). The sun rays will be calculated for each month, so a total of 6 times. In the second for-loop you can see that points on the sphere will be calculated in between these angles in for a total of 6 points. Inside the for-loops, the standard equations are given for the calculation of a sphere. With Rhino.Geometry, the desired points are then generated.
To generate the sun vectors on the neighbourhood, the difference is taken between the coordinates of the sun path and the coordinates of the centre of the building.

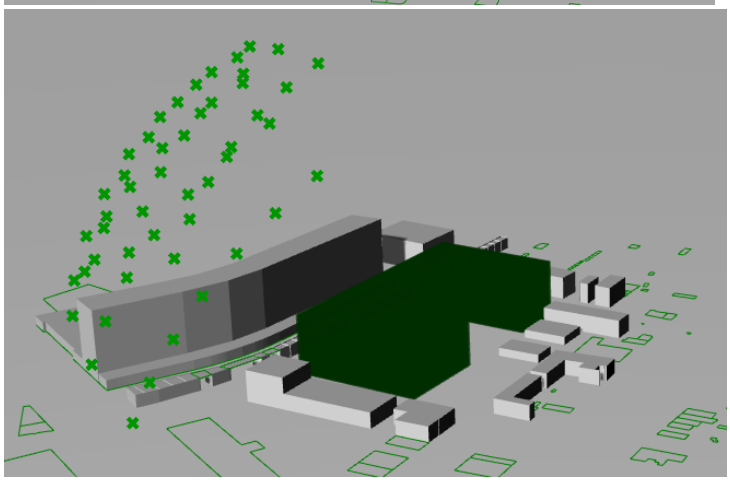
After the Python script, 2 rotations are made to the sun path.
Firstly, the current sun path represents the sun when standing on the north or south pole. To give the correct representation of the sun in Rotterdam, the sunpath should take into account the latitude of the city. For Rotterdam, this equals to 51,92 degrees. Because this angle is taken from the equator, the angle should be subtracted from the latitude of the north pole (90 degrees).

The second rotation is made to make sure that the middle of the sun path (which represents 12 o'clock) is situated on the southside of the neighbourhood.

In the following sketch, the angles of the sun path are explained.







Voxel & ray intersections

To see which voxels caused the most problems, the solar rays from the solar model were cast from each point of interest. These rays would eventually intersect with the voxels, indicating an obstruction. This model can be compared with “guns” being fired from the windows of surrounding building. The guns themselves would be placed on the points of

interest, firing at the voxels. If a voxel was shot a lot of times, it had to be removed to reduce the amount of shadow on the point of interest.

This means that from each PoI (point of interest) and for each solar vector, a ray had to be cast, meaning two *for* loops were required. The script, in this case, required three inputs:

- points of interest
- solar vectors
- voxels

The solar vectors, however, had to be turned around since they had to go toward the sun.

```
for each PoI:
    for each SolarVector:
        ReverseSolarVector = -SolarVector
        SolarRay = ray(PoI, ReverseSolarVector)
        Intersect(Voxels, SolarRay)
```

This script made use of the Intersection.MeshRay Method. The output of this function only gave the parameter at which point along the ray it intersected with a mesh. What we required was only if the ray intersects with a voxel or not. This was done by making sure the parameter was larger than 0 (a parameter lower than 0 means no intersections).

A for loop was made to loop through all the voxels and everytime the Intersection.MeshRay Method gave a parameter higher than 0 (=Voxel is hit), the index of this Voxel was stored in a list (HitVoxels). By looping through all the points of interests and sunvectors, the HitVoxels list becomes a long list of voxel indices that are hit.

Another double for-loop was made to loop through the HitVoxels list and count how many times each index was stored in this list (and therefore hit). These loops created another list (HitCounts) that stored these quantities.

```
for each PoI in PoI:
        for each SolarVector in SolarVector:
                ReverseSolarVector = -SolarVector
                SolarRay = ray(PoI, ReverseSolarVector)
                for each i,Voxel in enumerate(Voxels):
                        Intersect(Voxels, SolarRay)
                        if parameter  > 0:
                                HitIndice = i
                                HitVoxels.append(HitIndice)

for j in range(0,len(Voxels)):
        sum = 0
        for each HitVoxel in HitVoxels:
                if HitVoxel == j:
                        sum = sum + 1
        HitCounts.append(sum)
```

This resulted in the following code:

```
import rhinoscriptsyntax as rs
import Rhino.Geometry as rg
import Rhino.Geometry.Intersect as ri

HitVoxels = []
HitCounts = []

for i in range(0, PoI.Count):
    poi = PoI[i]
    for j in range(0, SV.Count):
        sv = SV[j]
        reverseSV = -sv
        ray = rg.Ray3d(poi, reverseSV)
        for k,voxel in enumerate(Voxels):
            P = ri.Intersection.MeshRay(voxel,ray)
            if P >= 0:
                HitIndice = k
                HitVoxels.append(HitIndice)

for l in range(len(Voxels)):
    sum = 0
    for m in range(len(HitVoxels)):
        hitvoxel = HitVoxels[m]
        if hitvoxel == l:
            sum += 1
    HitCounts.append(sum)
```

The output of the HitCounts list shows exactly how many times each voxel was hit by a ray. Colours were given to the voxels by using the maximum and minimum amount of hits as a boundary for a colour gradient.
To control which voxels had to be removed, another loop and input were added:

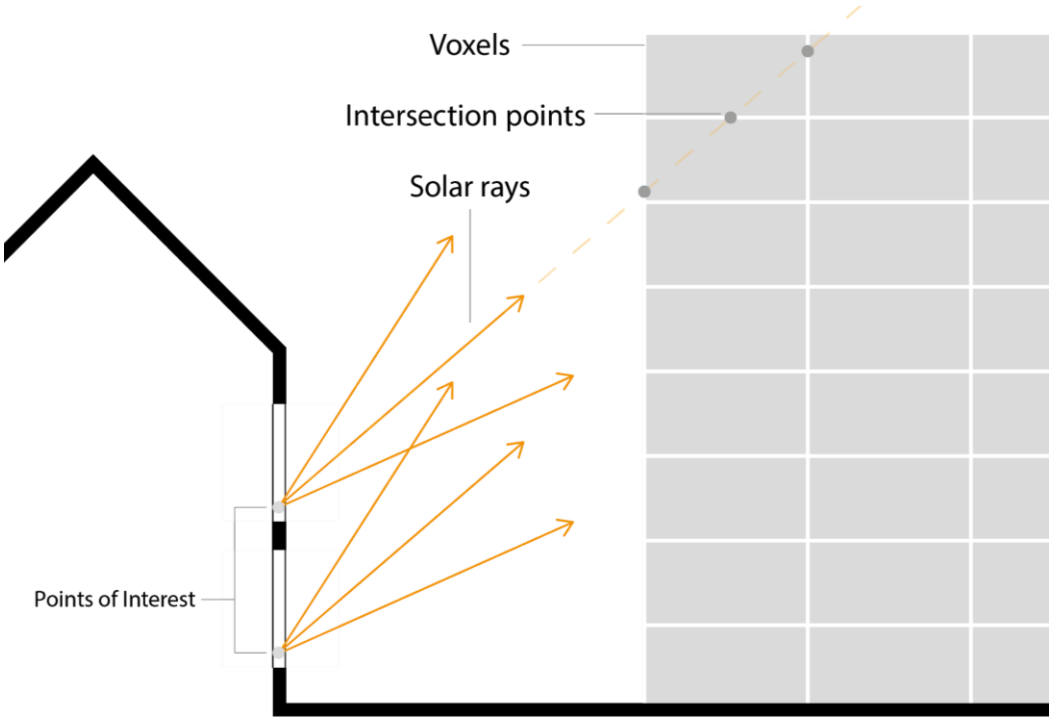
```
for n in range(0,len(HitCounts)):
```

```
if HitCounts[n] < maxhits:
        SolarEnvelope.append(Voxels[n])
else:
        RemovedVoxels.append(Voxels[n])
```

This loop only appends the voxels to the solar envelope if they are below the threshold, which is defined by a slider in grasshopper called maxhits.
A problem we encountered during the making of the solar envelope was the amount of meshes. The height of the building combined with the small dimensions of the meshes (1m x 1m x 3m) resulted in a very slow or malfunctioning process. The meshes of the solar envelope, in this case, could not even be displayed because Rhino would crash after a couple minutes.
This is why we increased the size of the voxels drastically: 5m x 5m x 3m instead of the previous 1 square meter. On one hand, this meant that the program could actually run. But on the other hand, this meant that the size of the voxels was too large to work with. This meant that we either had to:

- Create a NURBS solid that approaches the shape of the solar envelope, or:
- Convert the large voxels to NURBS solids and use these solids to create voxels of 1m x 1m x 3m.

In the end, this was no longer necessary because a larger voxel size did not have a negative effect on the solar envelope.



Input:
Points of Interest (list of points)

Solar Vectors (list of vectors)
Voxels (list of meshes)
maxhits (integer)

Pseudocode:

```
for each PoI in PoI:
        for each SolarVector in SolarVector:
                ReverseSolarVector = -SolarVector
                SolarRay = ray(PoI, ReverseSolarVector)
for each i,Voxel in enumerate(Voxels):
                        Intersect(Voxels, SolarRay)
                        if parameter  > 0:
                        HitIndice = i
                                HitVoxels.append(HitIndice)

for j in range(0,len(Voxels)):
        sum = 0
        for each HitVoxel in HitVoxels:
                if HitVoxel == j:
                        sum = sum + 1
        HitCounts.append(sum)

for n in range(0,len(HitCounts)):
        if HitCounts[n] < maxhits:
                SolarEnvelope.append(Voxels[n])
        else:
                RemovedVoxels.append(Voxels[n])
```

Output:

HitCounts: list of integers indicating how many times each voxel was hit
SolarEnvelope: list of meshes belonging to the solar envelope
RemovedVoxels: list of meshes removed from the original list of meshes