Serverless Programs - Python

- Voxel Generator Programs
- Report Programs
- View Programs
- Python Program Reference
- Program Examples

Voxel Generator Programs

A spatial entity will go through several stages in its lifetime. It is possible to provide custom logic for some of these stages. This version of the Voxel Farm platform allows running custom Python programs to produce volumetric spatial output. This type of program is known in the system as a Voxel Generator program.

Typically, a Voxel Generator program will go through the following stages:

1. Ask the user for input values, in case the generated object is parametric, and the user can provide values to these parameters.
2. Set the spatial entity bounds. This step is optional but should be always included because it allows the system to evaluate the spatial entity only within its known bounds, which is more efficient.
3. For each voxel in the spatial entity, emit a field density o other voxel attribute.

Initialization

Before anything else, the Python program must include the Voxel Farm Python library:

```
import voxelfarm
```

Voxel Generator programs must make sure to initialize the Voxel Farm Python library by calling “voxelfarm.init()”:

```
voxelfarm.init()
```

The program can then ask the user information about the entity using the “voxelfarm.input()” function:

```
h = voxelfarm.input(“altitude”, “Altitude”)
```

At this point, if the program knows of the spatial bounds of the entity, the bounds can be set by calling “voxelfarm.set_entity_bounds_x”, “voxelfarm.set_entity_bounds_y” and “voxelfarm.set_entity_bounds_z”.

```
vf.set_entity_bounds_z(h - 1000, h + 1000)
```

The program can either output voxel data, or field data. Field data will be contoured on the fly by the system and it is best suited for smooth surfaces. If the program needs to produce sharp edges and points, it is better to output voxel data.

Field Data Output

To output field data, the Python program must iterate over each voxel in the field. The range of voxels in a field is found in the “voxelfarm.field” variable:

```
for v in voxelfarm.field

    # will execute for each voxel “v” in the field
```

To get the coordinates in project space of the voxel, use the “voxelfarm.get_field_origin” function:

```
for v in voxelfarm.field

    p = voxelfarm.get_field_origin(v) # returns tuple for xyz
```

Using these coordinates, the user program can make useful computations. The program will have access to any Python library that is included in the Content Server. These computations eventually will produce a value for the field in the voxel’s origin. Voxel Farm will produce a surface wherever the field becomes zero. Places where the field is positive, will be considered “outside” the surface. Places where field is negative, will be considered solid and inside the surface.

To set the field value for the voxel, call the “voxelfarm.set_field” function:

```
for v in voxelfarm.field

    p = voxelfarm.get_field_origin(v) # returns tuple for xyz

    field = MyFieldFunction(p) # a call to custom function that outputs the field
```

```
    voxelfarm.set_field(v, field)
```

The following program creates an infinite plane with bumps on top of it:

```
import voxelfarm as vf

import math

vf.init()

h = vf.input("h", "Altitude")

vf.set_entity_bounds_z(h - 1000, h + 1000)

for v in vf.field:

    p = vf.get_field_origin(v)

    dh = 300 * (math.sin(0.001 * p[0]) + math.sin(0.001 * p[1]) + math.sin(0.001 * p[2]))

    dh = dh + 100 * (math.sin(0.01 * p[0]) + math.sin(0.01 * p[1]) +  + math.sin(0.01 * p[2]))

    vf.set_field(v, p[2] - h - dh)
```

Voxel Data Output

Consider using Voxel Data output in cases where you need the resulting spatial entity to contain sharp features.

Voxel Farm's voxel format allows for both sharp and smooth features. Once a spatial entity is built to output voxel data, its program must at least output a material definition for each voxel. Additional optional channels are a 3D surface vector and a color.

To output voxel data, the Python program must iterate over each voxel in the request. The range of voxels in a request is found in the "voxelfarm.voxels" variable:

```
for v in voxelfarm.voxels

    # will execute for each voxel “v” in the request
```

To get the coordinates in project space of the voxel, use the “voxelfarm.get_voxel_origin” function:

```
for v in voxelfarm.field

    p = voxelfarm.get_voxel_origin(v) # returns tuple for xyz
```

Using these coordinates, the user program can make useful computations. The program will have access to any Python library that is included in the Content Server. These computations eventually will produce a value that will determine if the voxel is set as solid material, and which values should be used for color and the surface position inside the voxel.

To set the material for a voxel, call the “voxelfarm.set_material” function:

```
for v in voxelfarm.field

    p = voxelfarm.get_field_origin(v) # returns tuple for xyz

    field = MyFieldFunction(p) # a call to custom function that outputs a field

    if (field < 0.0)

        voxelfarm.set_material(v, 1)
```

The following program creates an infinite plane with bumps on top of it:

```
import voxelfarm as vf

import math

vf.init()
```

```
h = vf.input("h", "Altitude")

vf.set_entity_bounds_z(h - 1000, h + 1000)

for v in vf.voxels:

    p = vf.get_voxel_origin(v)

    dh = 300 * (math.sin(0.001 * p[0]) + math.sin(0.001 * p[1]) + math.sin(0.001 * p[2]))

    dh = dh + 100 * (math.sin(0.01 * p[0]) + math.sin(0.01 * p[1]) +  + math.sin(0.01 * p[2]))

    field = p[2] - h - dh

    if (field < 0.0):

        vf.set_material(v, 1)
```

Report Programs

Reports are special entities that contain tabular results, like a CSV table. These results are obtained by running a custom program over the spatial datasets available in the project. It is up to the report program to select which datasets should be used, and how the data they contained should be mixed and interpreted. Once the program completes execution, its results are stored in the report entity.

A report program will typically go over the following stages:

1. Request user input. Report programs usually ask the user which datasets and attributes should be used and any other information required for the program to run.
2. Load the data. Once input is gathered, the program decides which data to load, and how it should be combined. The data can be a mashup of different datasets.
3. Look at the data. The report program can iterate over the data. If it decides it has found a relevant fact it can emit it as an RDF triplet.

The same report program is executed by the system in many threads at the same time, but over a different region of space in each case. The system will add the results of each individual run, integrating over all emitted RDF triplets. The triplets are then used to compose a table, which is stored in the report entity as a CSV file attachment.

Report Example

The following report program computes the volume of an object:

```
import voxelfarm as vf

entity = vf.input_entity("entity", "Entity", vf.type.voxel_terrain | vf.type.voxel_generator)

voxels = vf.load_voxels(entity, vf.attribute.volume, None)
```

```
volume = 0.0

for v in vf.voxels:

    volume += vf.get_volume(voxels, v)



sum = vf.start_sum("Item", "Object Volume")

vf.sum(sum, volume)
```

The program first asks the user for an entity by calling “input_entity()”. The program will compute the volume of this entity. The program requires this entity to be a Voxel Terrain or a Voxel Generator.

Next, the program calls “load_voxels”. This tells the system the program intends to use the voxel data associated with the input entity. The last two parameters to the function specify which attributes should appear in the voxel data. The first parameter is to select attributes from the default attribute set, like volume in this case. The second parameter is the set of custom attributes that will be loaded.

Then, for each voxel, the program calls the “get_volume()” function. This returns the volume for the voxel, a value between zero and the voxel size at the third power. The program proceeds to add up this value.

At the end, the program declares a new RDF triplet by calling “start_sum()”, the two parameters to this function contain the resource and property used for the triplet. The property value is incremented by the call to the “sum()” function. This is how every voxel’s volume is accounted for.

After running, this program generates a CSV file for the report that looks like this:

Item,Object Volume
Item,523605.137331239

Loading voxel attributes

The “load_voxels()” function instructs the system which voxel data must be loaded for the entity.

voxel_data = voxelfarm.load_voxels(entity, default_attributes, custom_attributes)

The last two parameters to the function specify which attributes we want to load from the voxel data.

The first parameter is to select attribute from the default attribute set, like volume in this case. This parameter is composed by performing a bitwise OR using the constants designated for each default attribute. The following table lists the possible values:

voxelfarm.attribute.none	Constant used for requesting no attributes
voxelfarm.attribute.volume	Requests the volume for each voxel

The second parameter is the set of custom attributes that will be loaded.

Volumetric accuracy

The report system in Voxel Farm is voxel-based. Voxels imply a discretization of space. A voxel may contain one or more samples of some object.

Voxel size must be equal or less than half the smallest distance from any two features in the object in order to be able to fully reconstruct the original object. Any pair of features less than the voxel size apart could erode, introducing an error comparable to the voxel size. This problem is generally referred to as aliasing.

Voxel Farm will use a precise sub-voxel geometric model to compute the volume value for the voxel. For instance, consider a sphere (in blue) that marginally intersects one voxel (in red):

The volume computed for this voxel will account only for the portion of the sphere that is really inside the voxel.

For volume computations, any surface inside the voxel is approximated to four quads. This is a close approximation, but it does introduce some error. This error is negative for convex surfaces and positive for concave ones.

At voxel size 0.5m, the earlier report program computes the volume of a 50m sphere to be: 523,605.137331239 cubic meters. The analytical result for the volume of a 50m sphere is: 523,598.7755983 cubic meters. The error introduced by the discretization of the sphere is 0.0012%

View Programs

A View entity defines a way to visualize several datasets together. In order to decide which datasets should be used, which information should be filtered, and how the results should be visualized, the user can supply a View Program, which will run a custom logic to construct a view layer.

A View program will typically go over the following stages:

1. Ask user input. Here the program prompts the user information to select which datasets and attributes should be used, visual options for rendering, and any other piece of information the view program requires in order to run.
2. Based on user input, create and configure a mashup of datasets from the available datasets in the project.
3. Set up rendering of the elements produced by the dataset mashup.

The following program creates a simple view layer to visualize a terrain mesh:

```
import voxelfarm as vf

terrain = vf.input_entity("terrain", "Terrain", vf.type.voxel_terrain)

mesh = vf.load_mesh(terrain, 0, None)

material = vf.new_material()

material.gradient = "dirt"

vf.render(mesh, material)
```

The program calls “input_entity()” to prompt the user for a terrain model. Next it asks the system to load the mesh data for the terrain using “load_mesh()”.

The following two lines set up a material that will be used to render the terrain mesh. The material is created by calling the “new_material()” function. The “gradient” property of the material is set to “dirt”, instructing the rendering system to ready this gradient texture and pass it as the gradient to be used by the GPU shader.

The last line calls the “render()” function, which instructs the mesh must be rendered with the material that was created in the previous lines.

Working with materials

A view program is expected to make at least one call to the “render()” function. This function takes two arguments, the last argument is the material that will be used to render the element. If no material is provided (“None” passed as material), the system will use the default material.

In this version, materials contain only two properties:

color	Defines an RGBA value as a four-member tuple. The RGB values will be used to tint the visual output for the component, while the alpha value will be used to make the surface transparent.
gradient	Can be one of the three available gradient modes: 1. “color” – Selects a vibrant color gradient 2. “dirt” – Selects a dirt-looking gradient 3. "difference" - Selects a red-to-green gradient

Python Program Reference

The following table lists the functions and properties available in the Python programmable interface:

input(id, label)	Requests a numeric input from the user. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for this value.
input_date(id, label)	Requests a date input from the user. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for this date.
input_string(id, label)	Requests a string input from the user. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for this string.
input_query(id, label, entity)	Requests a query input from the user. Queries are special strings that may involve conditional expressions that use the attributes in a dataset. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for this query. The “entity” parameter specifies which entity will be queried, this helps the system present the user a list of available attributes to query, which can be extracted from the entity’s metadata.
input_attributes(id, label, type, entity)	Requests the user to select a set of attributes. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for the attribute list. The “type” parameter indicates the type of input that will be used. Two values are allowed currently: • voxelfarm.type.value (1) – Used to select a single attribute • voxelfarm.type.set (2) – Used to select multiple attributes

input_entity(id, label, type)	Requests the user to pick an existing entity from the project. The value of the “id” parameter uniquely identifies the input, the programmer must make sure this value is only used for this input. The “label” parameter specifies the text that will appear in the UI to prompt the user for the entity selection. The “type” parameter contains a bit-field that signals which entity types are eligible for the input. The value for this parameter can be composed by performing a bitwise OR with the constants defined for each entity type: • voxelfarm.type.view (1 << 1) • voxelfarm.type.project (1 << 2) • voxelfarm.type.voxel_terrain (1 << 3) • voxelfarm.type.voxel_operations (1 << 4) • voxelfarm.type.block_model (1 << 5) • voxelfarm.type.point_cloud (1 << 6) • voxelfarm.type.voxel_generator (1 << 7) • voxelfarm.type.point_cloud_raw (1 << 8) • voxelfarm.type.heightmap_raw (1 << 9) • voxelfarm.type.block_model_raw (1 << 10) • voxelfarm.type.mesh_raw (1 << 11) • voxelfarm.type.mesh (1 << 12) • voxelfarm.type.ortho (1 << 13) • voxelfarm.type.program (1 << 14) • voxelfarm.type.folder (1 << 15)
set_entity_bounds_x(min, max)	Specifies the entity is contained between “min” and “max” coordinates along the X axis. This function is called only by Voxel Generator programs.
set_entity_bounds_y(min, max)	Specifies the entity is contained between “min” and “max” coordinates along the Y axis. This function is called only by Voxel Generator programs.
set_entity_bounds_z(min, max)	Specifies the entity is contained between “min” and “max” coordinates along the Z axis. This function is called only by Voxel Generator programs.
voxels	A range containing the voxels for the program to read or produce.
field	A range containing the field entries that should be produced. This is used only by Voxel Generator programs.
get_voxel_origin(voxel)	Returns a tuple that contains the (X,Y,Z) coordinates of the voxel’s origin in Project space.
get_field_origin(field_voxel)	Returns a tuple that contains the (X,Y,Z) coordinates of the field origin in Project space.
set_material(voxel, id)	Applies sub-material id to the voxel. This is used only by Voxel Generator programs. Only sub-material 1 is allowed in the current version.
set_vector(x, y, z)	Sets surface vector for the voxel. This is used only by Voxel Generator programs.

set_field(field_voxel, field_value)	Sets field value for the field voxel. This is used only by Voxel Generator programs.
set_query(entity, query)	Sets a query value to the specified entity.
set_attribute_gradient(entity, attribute, min_value, max_value)	Sets cutoff values for an attribute in the specified entity.
start_composite()	Creates a new virtual entity that will be the result of stacking multiple voxel entities as layers.
add_layer(composite, entity)	Adds the specified entity to an existing voxel composite (returned by start_composite).
get_entity_type(entity)	Returns the type of the specified entity. Possible return values are: • voxelfarm.type.view (1 << 1) • voxelfarm.type.project (1 << 2) • voxelfarm.type.voxel_terrain (1 << 3) • voxelfarm.type.voxel_operations (1 << 4) • voxelfarm.type.block_model (1 << 5) • voxelfarm.type.point_cloud (1 << 6) • voxelfarm.type.voxel_generator (1 << 7) • voxelfarm.type.point_cloud_raw (1 << 8) • voxelfarm.type.heightmap_raw (1 << 9) • voxelfarm.type.block_model_raw (1 << 10) • voxelfarm.type.mesh_raw (1 << 11) • voxelfarm.type.mesh (1 << 12) • voxelfarm.type.ortho (1 << 13) • voxelfarm.type.program (1 << 14) • voxelfarm.type.folder (1 << 15)
get_entity_name(entity)	Returns the readable name for the specified entity.
get_entity_date(entity)	Returns the capture date of the specified entity.
folder_contains(folder, entity)	Returns a Boolean indicating whether the entity is inside the specified folder. This function looks inside subfolders as well, so the entity could be nested multiple levels deep from the specified folder.
get_attributes(attributes)	Returns a range to iterate on the specified attribute set.
get_attribute_name(attributes, index)	Returns the attribute identifier for the attribute located at the specified index, from the specified attribute set.
start_sum(item, property)	Begins integration of an RDF triplet. The “item” and “property” parameters define the subject and property items of the triplet. See the “sum()” function to see how to provide a value to the triplet. Returns an integer value that identifies the new sum.
sum(sumId, value)	Adds the specified value to an existing sum. The “sumId” parameter is the identifier returned by the call to “start_sum()”.

get_volume(voxeldata, voxel)	Returns the volume of the specified voxel in the specified voxel data. The voxel data must be loaded with the “load_voxels()” function. The returned value will range from zero to the (voxel size)3
get_attribute_value(voxeldata, attribute, voxel)	Returns the value of the specified attribute for the specified voxel.
load_voxels(entity, attributes, custom_attributes)	Instructs the system to load the voxel data for the specified entity. The “attributes” parameter contains a bit mask that defines which default attributes should be loaded. This can be computed as a bitwise OR of the constants for the desired default attributes. Currently the following default attributes are supported: • voxelfarm.attribute.volume – Returns the volume of each voxel The “custom_attributes” parameter contains an array of strings where each entry corresponds to a custom attribute that may be found in the entity. If no custom attributes are required, it is possible to pass “None” in this parameter.
load_mesh(entity, attributes, custom_attributes)	Instructs the system to load the mesh data for the specified entity. In this version, the “attributes” parameter should be set to zero and the “custom_attributes” to None.
load_points(entity, attributes, custom_attributes)	Instructs the system to load the mesh data for the specified entity. In this version, the “attributes” parameter should be set to zero and the “custom_attributes” to None.
new_material()	Creates a new material instance.
render(component, material)	Instructs the system to render the specified component using the specified material. Only components of type mesh (returned by “load_mesh”) and of type points (returned by “load_points”) are supported by this version.

Program Examples

Voxel Generator for a Torus object

This program generates a volumetric Torus object:

```
import voxelfarm as vf
import math



def length2d(p):
    return math.sqrt(p[0] * p[0] + p[1] * p[1])

def torus(p, r1, r2):
    d = length2d((p[0], p[2]))
    q = (d - r1, p[1])
    return length2d(q) - r2

vf.init() # necessary for Voxel Generator programs

xo = vf.input("x", "Torus Center X")
yo = vf.input("y", "Torus Center Y")
zo = vf.input("z", "Torus Center Z")
r1 = vf.input("r1", "Torus Radius")
r2 = vf.input("r2", "Torus Inner Radius")

vf.set_entity_bounds_x(xo - r1 - r2, xo + r1 + r2)
vf.set_entity_bounds_z(zo - r1 - r2, zo + r1 + r2)
vf.set_entity_bounds_y(yo - r2, yo + r2)

for v in vf.field:
    voxel_p = vf.get_field_origin(v)
    p = (voxel_p[0] - xo, voxel_p[1] - yo, voxel_p[2] - zo)
    f = torus(p, r1, r2)
    vf.set_field(v, f)
```

The program produces the following output:

Depletion Report

The following program computes a depletion report using two date ranges and a block model:

```
import voxelfarm as vf

block_model = vf.input_entity("bm", "Block Model", vf.type.block_model)
attributes = vf.input_attributes("attrs", "Attribute", vf.type.value, block_model)

terrainAD1 = vf.input_date("terrainA_date_start", "Terrain A Start Date")
terrainAD2 = vf.input_date("terrainA_date_end", "Terrain A End Date")
folderA = vf.input_entity("folderA", "Terrain A Parent Folder", vf.type.folder)
terrainBD1 = vf.input_date("terrainB_date_start", "Terrain B Start Date")
terrainBD2 = vf.input_date("terrainB_date_end", "Terrain B End Date")
folderB = vf.input_entity("folderB", "Terrain B Parent Folder", vf.type.folder)

surfaceA = vf.start_composite()
surfaceB = vf.start_composite()

for i in vf.entities:
    if vf.get_entity_type(i) & vf.type.voxel_terrain != 0:
        date = vf.get_entity_date(i)
        if date >= terrainAD1 and date <= terrainAD2 and vf.folder_contains(folderA, i):
            vf.add_layer(surfaceA, i)
        if date >= terrainBD1 and date <= terrainBD2 and vf.folder_contains(folderB, i):
            vf.add_layer(surfaceB, i)

terrain_voxelsA = vf.load_voxels(surfaceA, vf.attribute.volume, None)
terrain_voxelsB = vf.load_voxels(surfaceB, vf.attribute.volume, None)
voxel_data = vf.load_voxels(block_model, vf.attribute.volume, attributes)

attribute_list = vf.get_attributes(attributes)
```

```
for attr in attribute_list:
    vf.start_sum("Depleted", vf.get_attribute_name(attributes, attr))

for v in vf.voxels:
    volume = vf.get_volume(voxel_data, v)
    if volume > 0.0:
        for attr in attribute_list:

            # get block model attribute value
            value = vf.get_attribute_value(voxel_data, attr, v)

            # get terrain volume
            volumeTerrainA = vf.get_volume(terrain_voxelsA, v)
            volumeTerrainB = vf.get_volume(terrain_voxelsB, v)

            # clip affected volume to block model volume
            volumeA = min(volume, volumeTerrainA)
            volumeB = min(volume, volumeTerrainB)

            # sum attribute weight
            if volumeA > volumeB:

                # this was cut
                volume = min(volume, volumeA - volumeB)
                vf.sum(attr, volume * value)
```

The program will compute the tonnage for a user-supplied list of attributes.

The two date ranges are used to compute two surfaces. All terrain datasets within the first date range will be composited together in a single surface. The program does the same with the second date range, obtaining a second surface.

The program will assume the second composite is the most recent one.

Block Model with Terrain View

This program overlays a semi-transparent terrain over a block model:

```
import voxelfarm as vf

terrain = vf.input_entity("terrain", "Terrain", vf.type.voxel_terrain)
```

```
bm = vf.input_entity("bm", "Block Model", vf.type.block_model)

mesh1 = vf.load_mesh(terrain, vf.attribute.none, None)
mesh2 = vf.load_mesh(bm, vf.attribute.none, None)

mat1 = vf.new_material()
mat1.color = [1, 1, 1, 0.9]
mat1.gradient = "dirt"

mat2 = vf.new_material()
mat2.color = [1, 1, 1, 1]
mat2.gradient = "color"

vf.render(mesh1, mat1)
vf.render(mesh2, mat2)
```