Center for
Geospatial Analytics

3D visualization of geospatial data using Blender

This is material for the summer geospatial studio held at Center for Geospatial Analytics, in Raleigh, NC, 2 August, 2017.

Prepared by : Payam Tabrizian
Presented with: Anna Petrasova, Vaclav Petras, and Helena Mitasova
Tested and reviewed by: Garrett Millar

Contents

Abstract

What if your geospatial data and simulations like flooding, fire-spread and view shed are converted to realistic, interactive and immersive 3D worlds, without the need to deal with overly complicated or proprietary 3D modelling software ? In this hands-on workshop we will explore how to automate importing and processing of various types of geospatial data (e.g., rasters, vectors) using Blender, an open-source 3D modelling and game engine software. We will start with a brief and focused introduction into Blender graphical user interface (GUI), Python API, as well as the GIS addon. Once we import our GIS data into Blender, we will go over the techniques (both with GUI and command line) to increase the realism of our 3D world through applying textures, shading, and lighting.

I. What is Blender and why using Blender?

Blender is an open-source 3D modelling, rendering and game engine software. You can create photorealistic scenes and life-like animations with it. The feature that makes Blender highly suitable for geospatial visualization is its capability to import various georeferenced data thanks to BlenderGIS addon. Almost every operation done in the blender interface, can be scripted in the Python scripting environment, allowing you to automate or batch process your 3D modelling workflow. Moreover, using Blender4web or sketchfab addons, you can would be able to publish

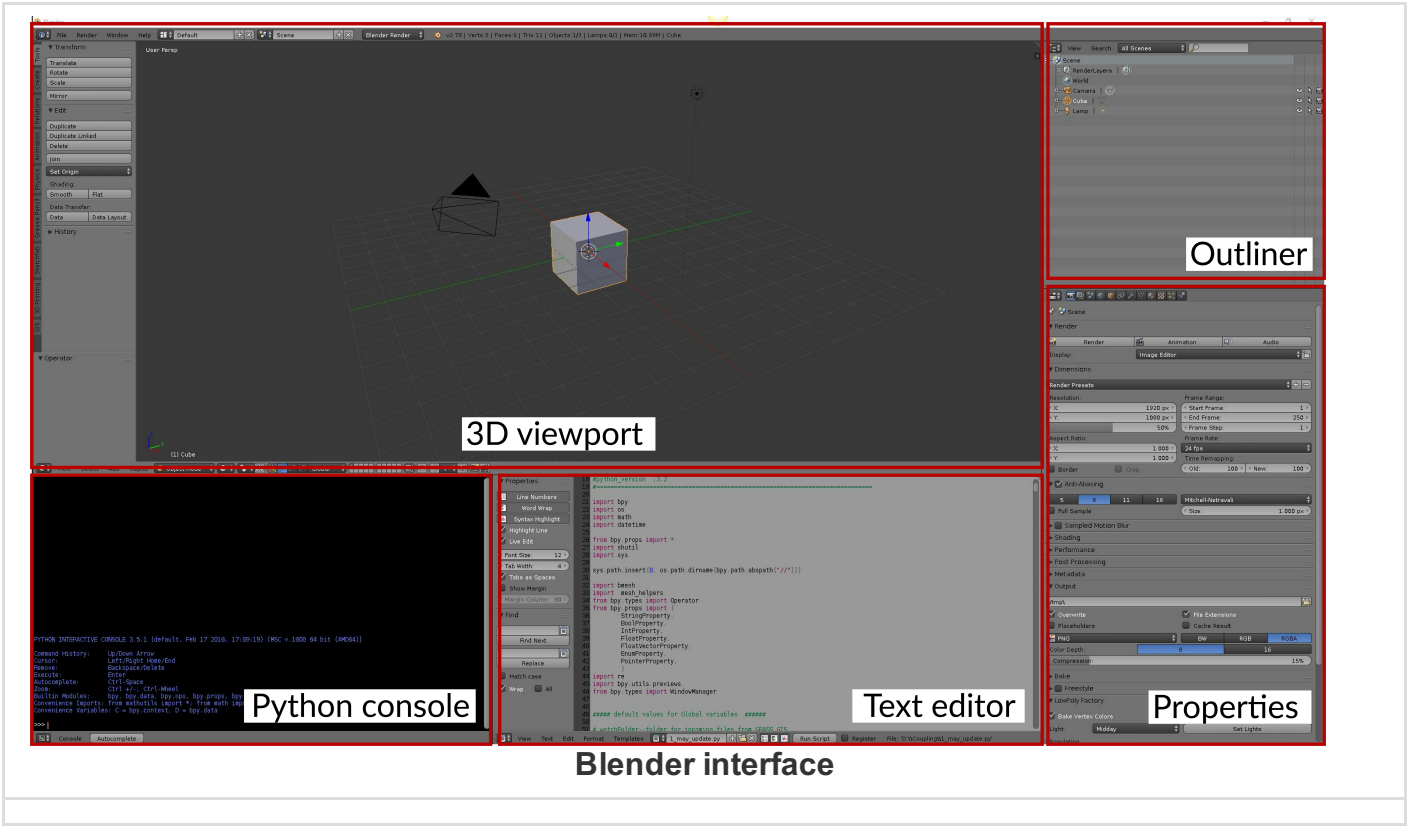
your geospatial models online, so that everyone can interactively explore or download your work.

Learn more about Sketchfab
A sample geospatial model in Sketchfab

II. Basic components of the Blender interface

Blender has numerous components and features that, thanks to it open-source capabilities, are growing every day. Covering all aspects of the software itself require several lessons. The purpose of this section is to provide a brief introduction to Blender's graphical user interface and some of its features that are essential for working with geospatial data, and will be used throughout this tutorial. We will specifically introduce the following components: **Areas, Editors, Tabs, Headers, Panels**

- Browse the workshop data folder, locate and open *interface_introduction.blend*



Blender interface

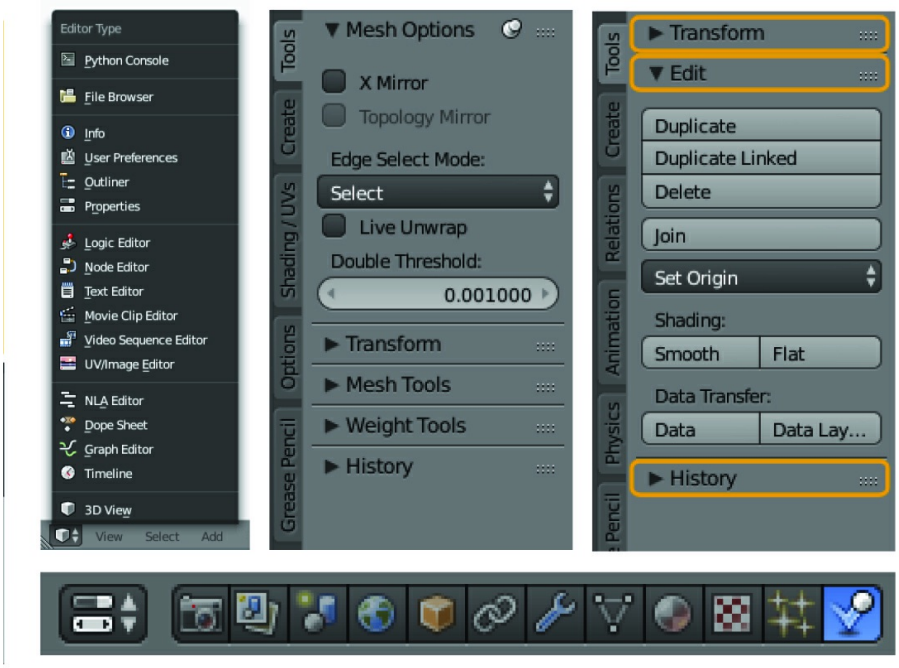
Blender's application window can be flexibly arranged and divided up into a number of **Areas**. An area contains the workspace for a particular type of editor, like a *3D View Editor*, or an *Outliner*. In figure above you can see the application window is divided into five areas, each assigned to an editor.

Editors are responsible for displaying and modifying different aspects of data. Imagine editors as full-fledge software specialized for a specific tasks, like changing data properties, image editing, video editing, animation design, game design, and etc. You can assign an area to a specific editor using **Editor Type selector** , the first button at the left side of a header (figure below, left). Every area in Blender may contain any type of editor and it is also possible to open the same type multiple times.

Tabs are overlapping sections in the user-interface. The Tabs header can be vertical (Tool Shelf) or horizontal (Properties Editor, User Preferences).

Another common feature is the **Header**, that contain menus and commonly used tools specific to each editor. It has a small horizontal strip shape with a lighter gray background, which sits either at the top or bottom of the area.

Finally, the smallest organizational unit in the user interface is a **Panel**. You can usually collapse panels to hide their contents. They are used in the Properties Editor, but also for example in the Tool Shelf and the Properties region. In the image below on the right you can see three panels one of them is expanded and the rest are collapsed.



Above: Editor type selector (left), A Toolbar w ith tabs (middle), Toolbar Panels (right) **Below:** A Header w ith tabs

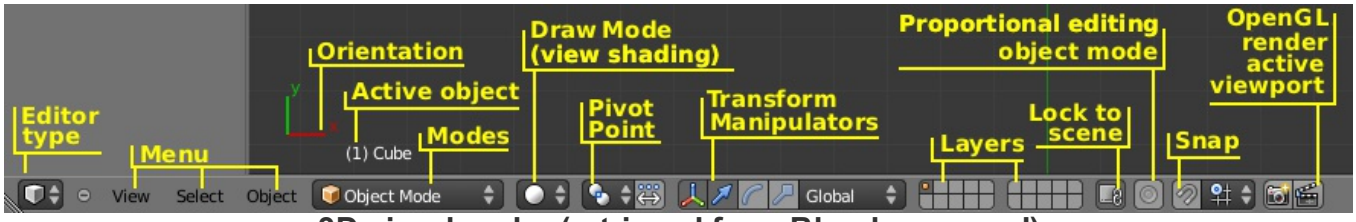
III. Editors

Now that you have some general ideas about the interface, in follow ing w e w ill review some of the commonly used editors.

3D view

The **3D View** is the visual interface w ith the 3D data and scene w ith numerous functionalities for modeling, animation, texture painting, etc. Unlike the 2D environment of GIS softw are, w here you can only navigate in x and y directions, 3D view port allow s full control over our view ing angle, depth, size, etc. You can press and hold dow n mouse scroll (or middle click) button to change the view ing angle (or orbiting around), shift and drag to pan, and roll to zoom back and forth.

Now note the panel on the left side of the region w hich is called **Tool shelf** and it has a variety of the tools for 3D editing. New ly installed addons also appear in this toolbar. Now notice the bottom **Header**. It includes menus for adding, editing objects as w ell as view ing and shading options.



3D view header (retrieved from Blender manual)

Header's **View menu** allow you to select a specific view point such as top, left or different perspectives. Also notice that each of these commands have a keyboard shortcut associated w ith them. For example you can push `numpad 3` (if you have a full keyboard) to sw itch to top view.

Add menu provides a list of different types of 2D and 3D object that can be added to a scene

Interaction mode allow s you to toggle betw een the **Object mode** and **Edit mode**. Edit mode allow s you to access more low -level structures of your object, like faces, and vertices. In the examples that w e complete in this tutorial, w e w ill use some of these options to refine the surface model. It is important to get familiar w ith the 3 core elements, **Faces**, **Edges** and **Vertex**. You can select these elements by clicking on their corresponding icons.

On the right side of the interaction mode, is the view port **Shading mode** w hich you can use to choose the visualization and view port rendering method. Default is the **Solid mode** that show s objects w ith solid faces, but w ithout textures and shading. The **Material mode** show s the object w ith textures and is suitable for having an idea how 3D objects may look like w ith materials. The **Rendering mode**, enables real-time

rendering, which computes the near-to-final product on-the-fly as you interact with the object.

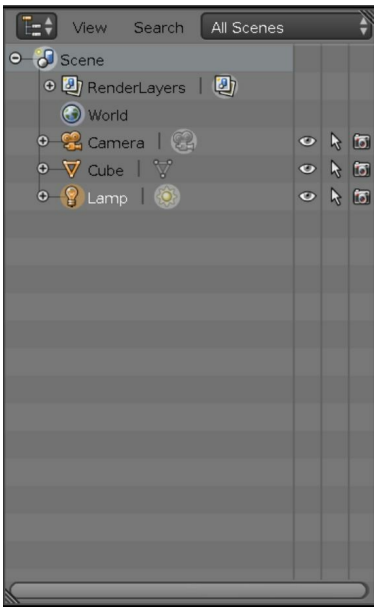
Basic object selection and interaction

Objects are basically everything that you see in the 3D view. They include 3D objects, lights and cameras. You can select any object in the scene using the mouse right-click. Selected objects are highlighted in orange so you can easily distinguish them. Use the 3 axes (i.e., handles) to move the object in your preferred direction. To select multiple objects, press and hold `control` key and right click on objects to add to your selection. You can rotate objects by pressing `R` keyboard button, or scale objects using `S` key. Note that when you are transforming an object, a numeric output on the left bottom of the 3D view port will give you more precise feedback on how much you moved, rotated or scaled an object. You can delete the object by selecting it, pressing `delete` key and selecting ok.

Learn more about 3D view

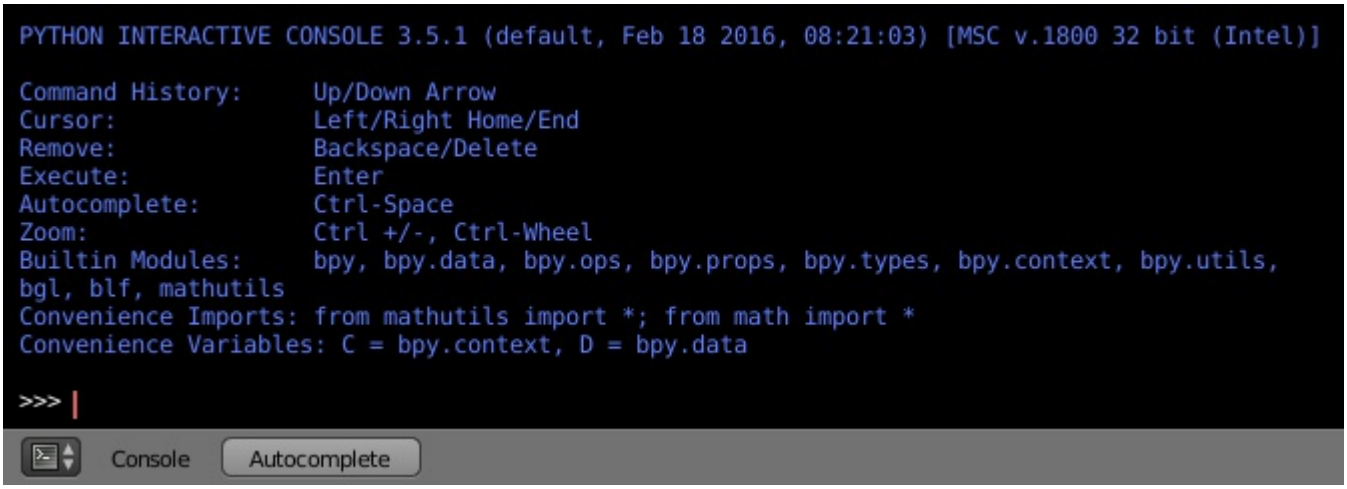
Outliner

As its name suggests, outliner lists and organizes the scene objects. From there you can set the hierarchy, visibility of the 3D objects or lock them if you need. You can also select and activate objects by clicking on their name in the list. Figure below shows **Outliner editor** that list three objects (Camera, Cube and Lamp). The Lamp object is selected.



Python console

The Python console is a very useful editor for testing and executing short commands, which can then integrated in larger workflows. The Blender modelling and gaming modules are already loaded in python console so you can you can test your code snippets without extra effort of calling the modules.



Python console (retrieved from Blender manual)| |:---:|

`Example 1.` Simple object operation using python console.

- Call *Cube* object and print its location
  - Copy and paste the individual command lines in the console and press enter

```
cubeObj = bpy.data.objects[“cube”]
print (cubeObj.location)
```

- Move cube object to location *x = 0 , y = 2 , z = 3*

```
cubeObj.location = (0, 2, 3)
```

- move cube object 5 units in positive X direction

```
cubeObj.location [0] += 5
```

- Select and delete cube object

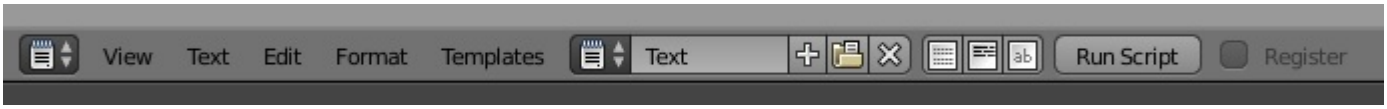
```
cubeObj.select = True
bpy.ops.object.delete()
```

- Adding a simple object (Blender's default Monkey object) w ith radius 2 ,and location (0,0,0)

```
bpy.ops.mesh.primitive_cube_add(radius=2, location = (0,0,0))
```

Text Editor

Text editor allow s you to edit your python script and run it inside Blender. By pressing the **+** icon you can start a new file and click on **Run Script** to execute your code. You need to call modelling and gaming modules in text editor.



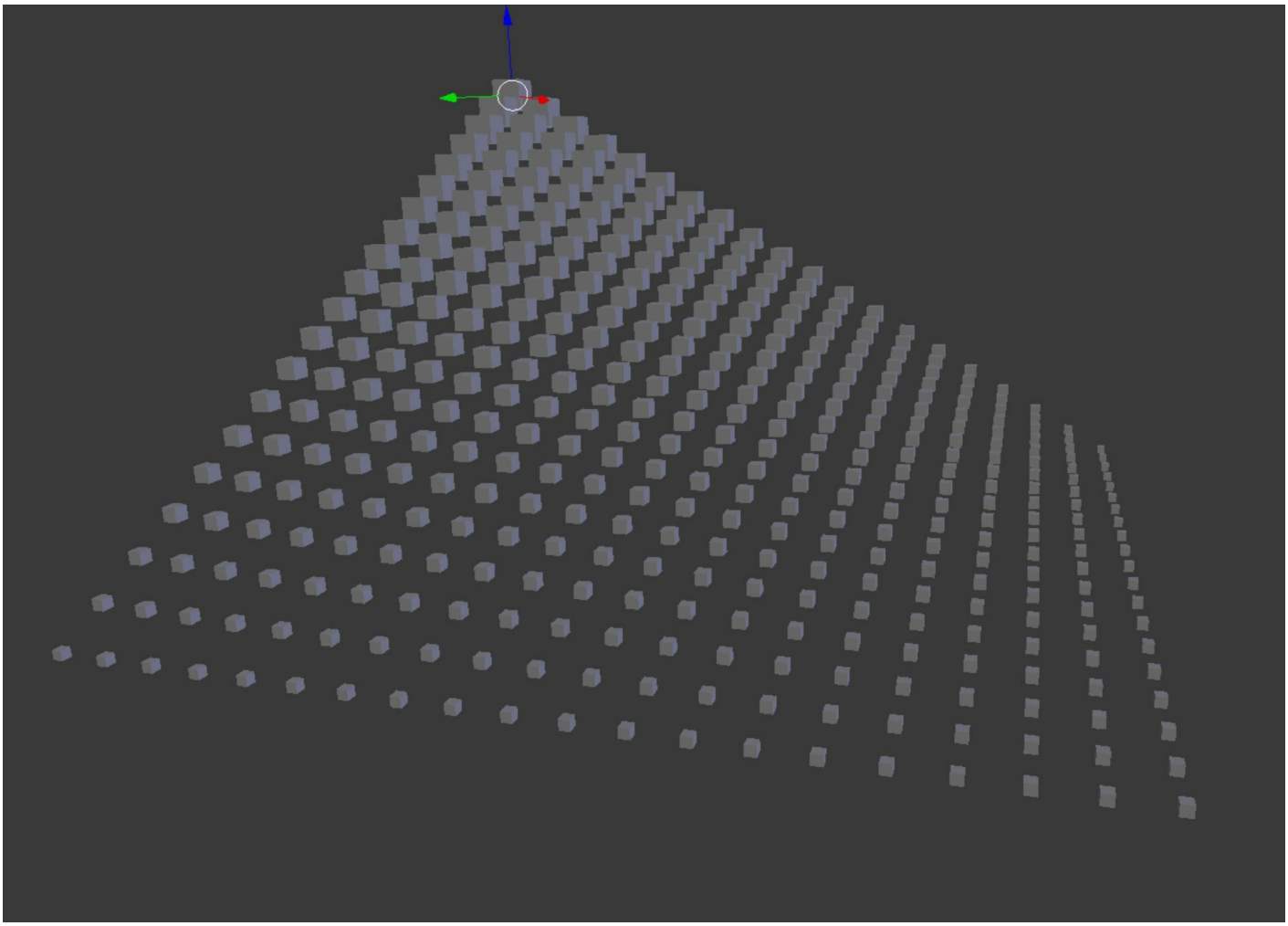
Text Editor| |:---:|

`Example 2.` Batch processing simple object operations using text editor

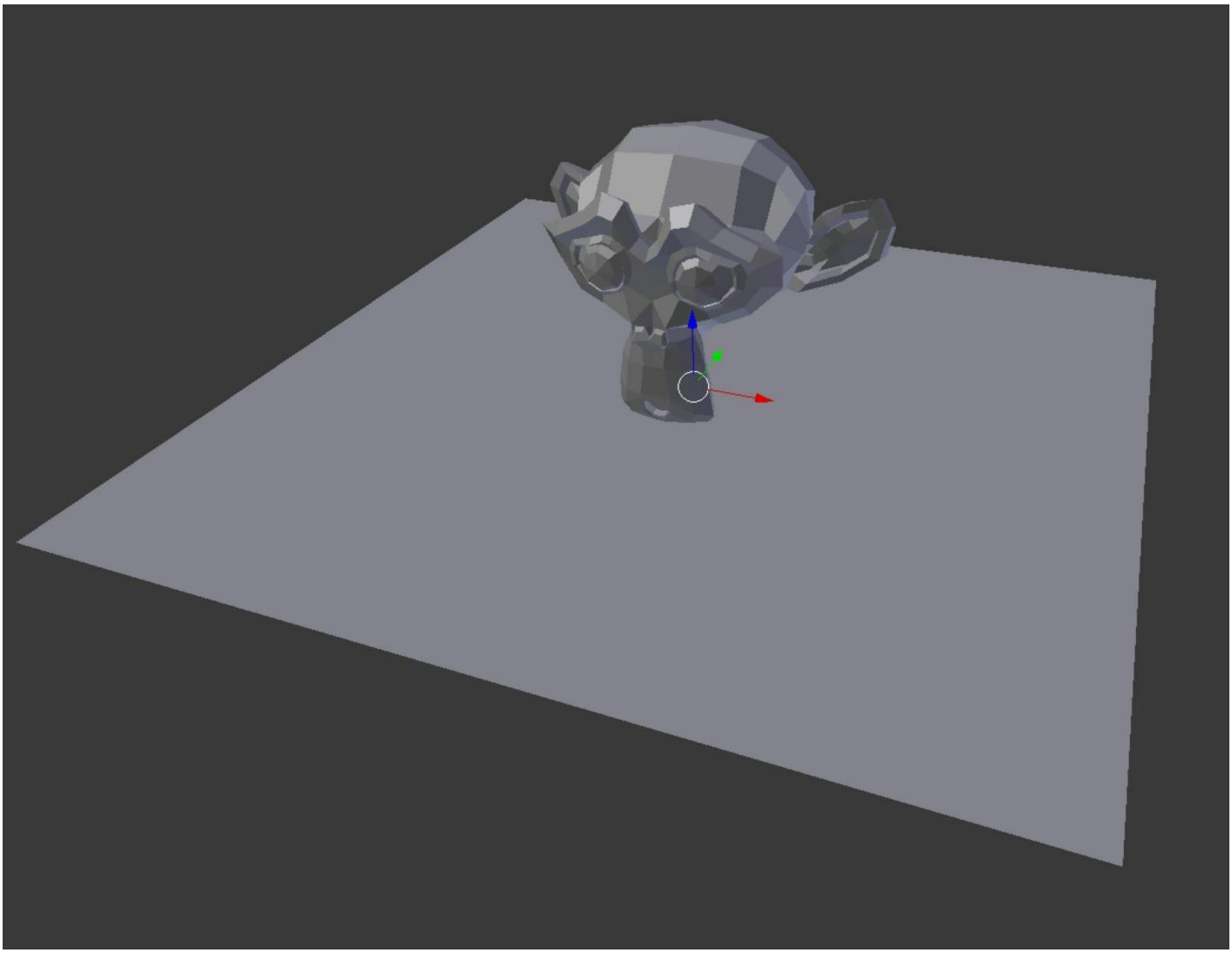
- Create a matrix of Cubes w ith varied size and location.
  - In the text editor click on the **+** icon to create a new textfile
  - Copy and paste the snippet below and click on **Run script** button
  - The results should look like the figure below

```
import bpy

for x in range(20):
    for y in range(20):
        bpy.ops.mesh.primitive_cube_add(radius = .1 + (x*y*.0005), location=(x, y, (x*y*.02)))
```



Cube matrix|


Monkey and plane |:---:|:---:|

- Delete all cube objects, add a *Monkey* object, and add a *Plane* object
  - Open a new text window or delete the contents of existing ones

```
import bpy

for object in bpy.data.objects:
    if "Cube" in object.name:
        object.select = True
        bpy.ops.object.delete()

bpy.ops.mesh.primitive_monkey_add(location=(0, 0, 0), radius = 3)
bpy.ops.mesh.primitive_plane_add(location=(0, 0, -3), radius = 10)
```

Properties editor

Properties editor allows you to modify the properties of the scene, rendering setting, transforming objects or changing their material or texture properties. The components that we will work with in the following examples are *Object, Material and Texture properties*.

Properties panel

Note: Properties editor's interface is dynamically changing according to the selected object. For example, if you select the light, the little sun icon will appear to set the light properties and similarly you should select camera to be able to see the camera tab and modify the properties.

Object properties tab allows you to transform the location, orientation and scale of the object, along with their display properties. You can use numeric input for transformation parameters.

`Example 3.` Basic object transformation using properties modifier

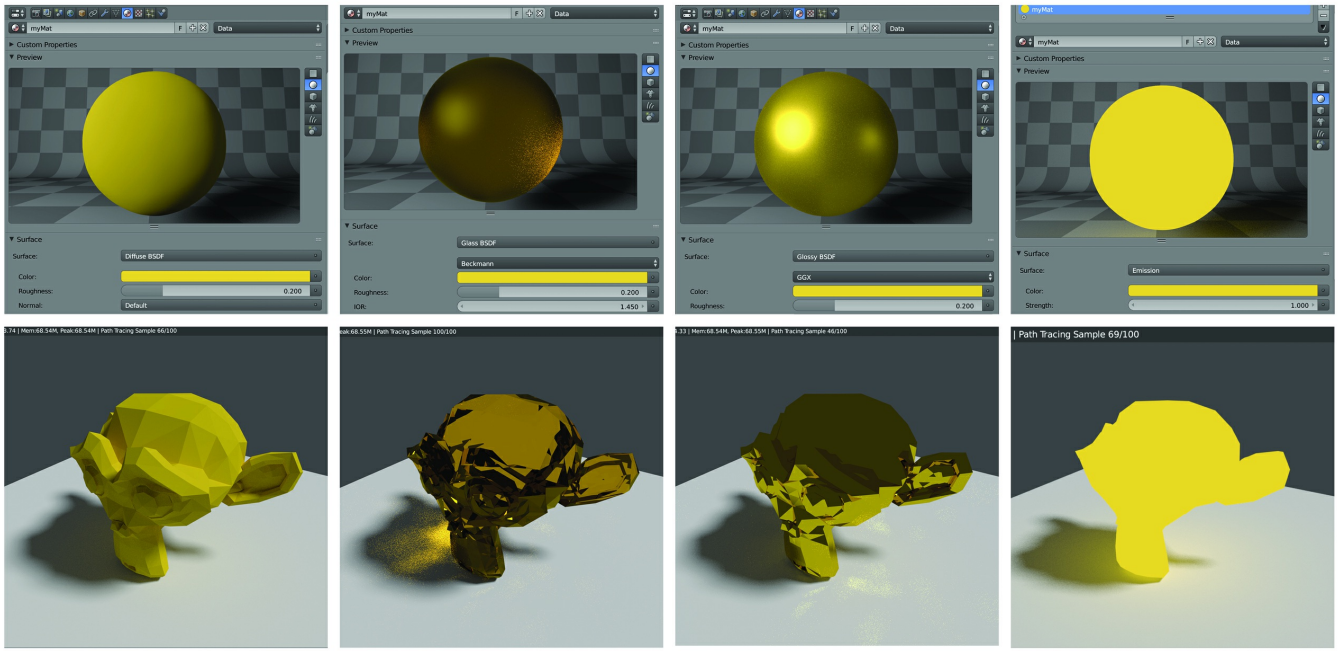
- Make sure that *Suzanne* object is selected. It should be highlighted in **outliner**
- Go to **Properties editor** > **Object tab** > expand the **Transform** panel
- Type *3, 2, 4* for *X, Y, Z* parameters, respectively.
- Change *Rotation* and *Scale* parameters to see how they affect the object

Materials tab allows you to assign or change an object's material. You can add and remove material, or use material browser to assign previously created materials to the object. Some very basic material parameters include *Diffuse and Specular*. You can adjust the diffuse parameters to change the color and shading of the material and with Specular adjust the glossiness of the material. Also, play with the shading and transparency parameters to see how it impacts your object. In this tutorial we briefly introduce two basic components of Materials. namely **Shaders** and **Textures**.

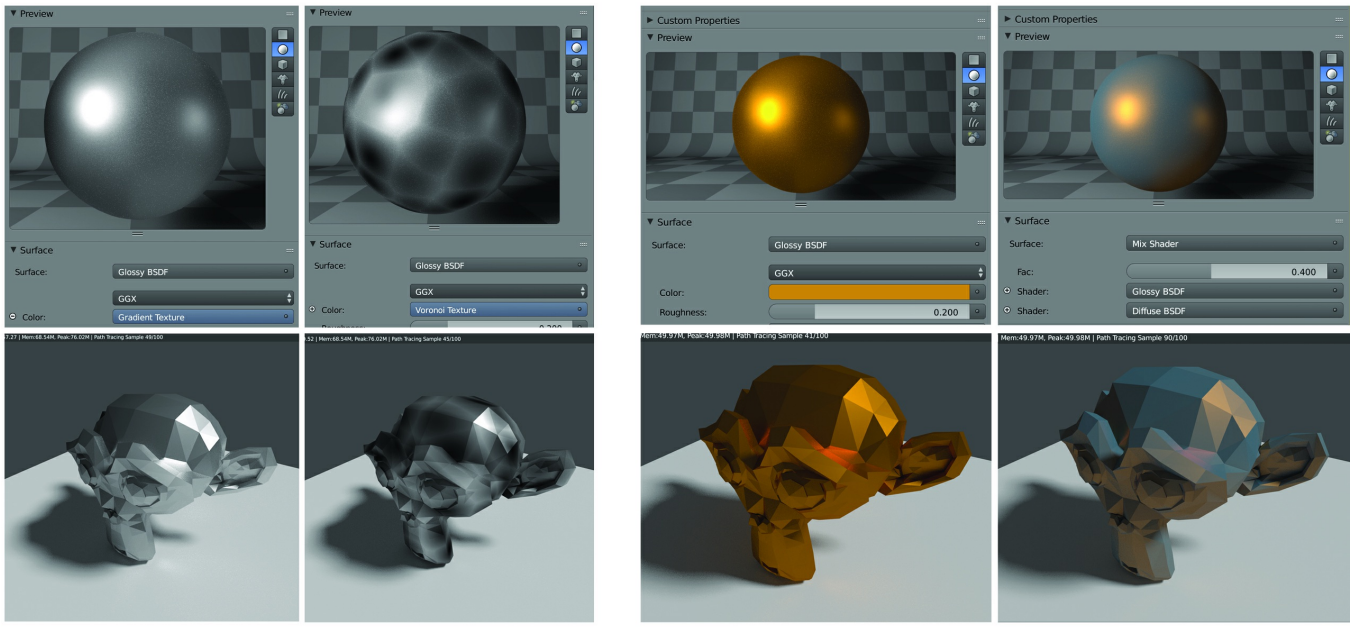
Shading (or coloring) allows you to adjust the base color (as modified by the diffusion and specular reflection phenomenon) and the light intensity. **Textures** . You can also assign texture to the materials. You can either select from preset textures already available in scene, or load a new one from hard drive.

`Example 4.` Assigning simple Shaders and Textures

- Shaders
  - From outlier select object make sure that *Suzanne* object is selected
  - Go to **properties editor** > **object tab** > click on the **+ New** button to create a new material
  - Double click on the material name (e.g., *Material.001*) and change it to *Mymat*
  - Expand the preview panel to see a live preview of the material as you are changing it
  - Change the color parameter to red
  - Go to 3D editor bottom **Header** > **Viewport shading** > **rendered** to see the object render in realtime
  - Change the color to yellow
  - Click on the **Diffuse BSDF** field in front of the surface parameter and select *Glass BSDF*
  - Now try *Emission BSDF* and *Glossy BSDF* shaders while the view port shader is on *Rendered* mode to see the effect on rendering. Your material preview and scene rendering should look like the figure shown below



- Textures
  - While the shader is still on “Glossy BSDF”, click on the radio button in front of the “Color” parameter. A widget with several columns will appear. From the texture column, select “Voronoi” to see how texture impact the rendering.
  - Now try “Gradient” texture. Your material preview and scene rendering should look like the left two columns in the figure below.

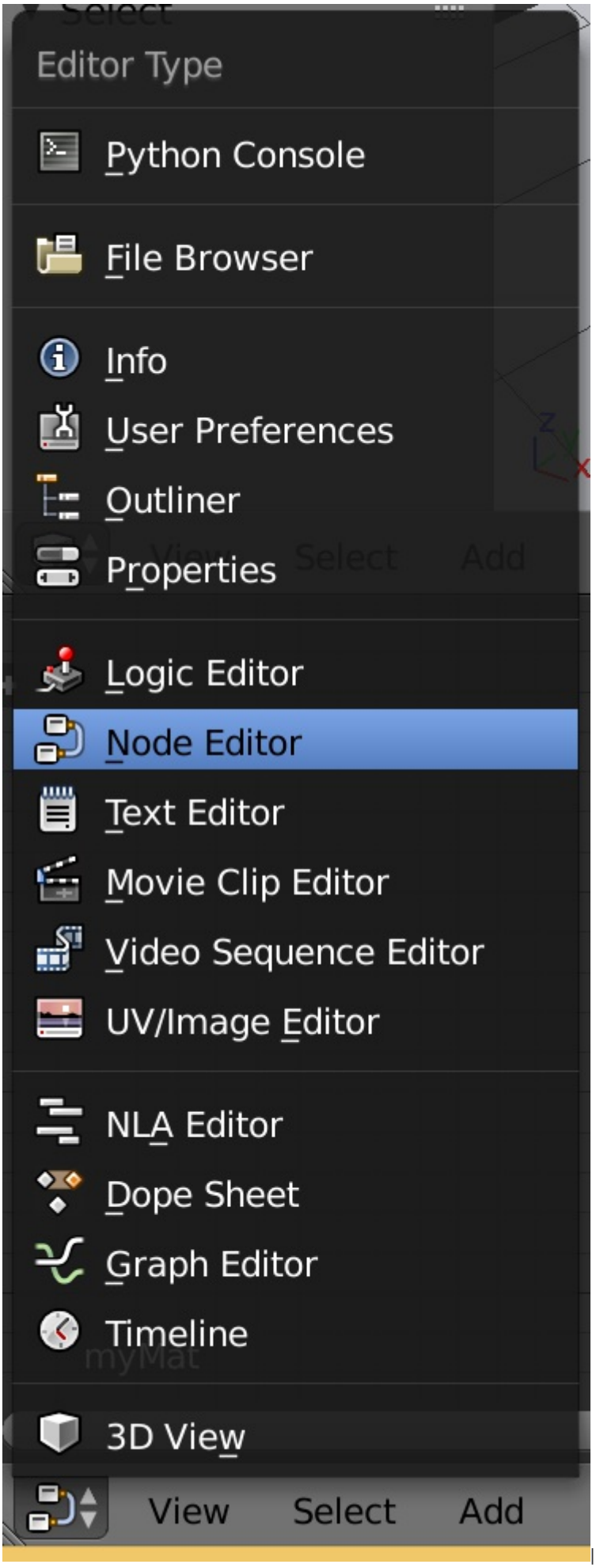


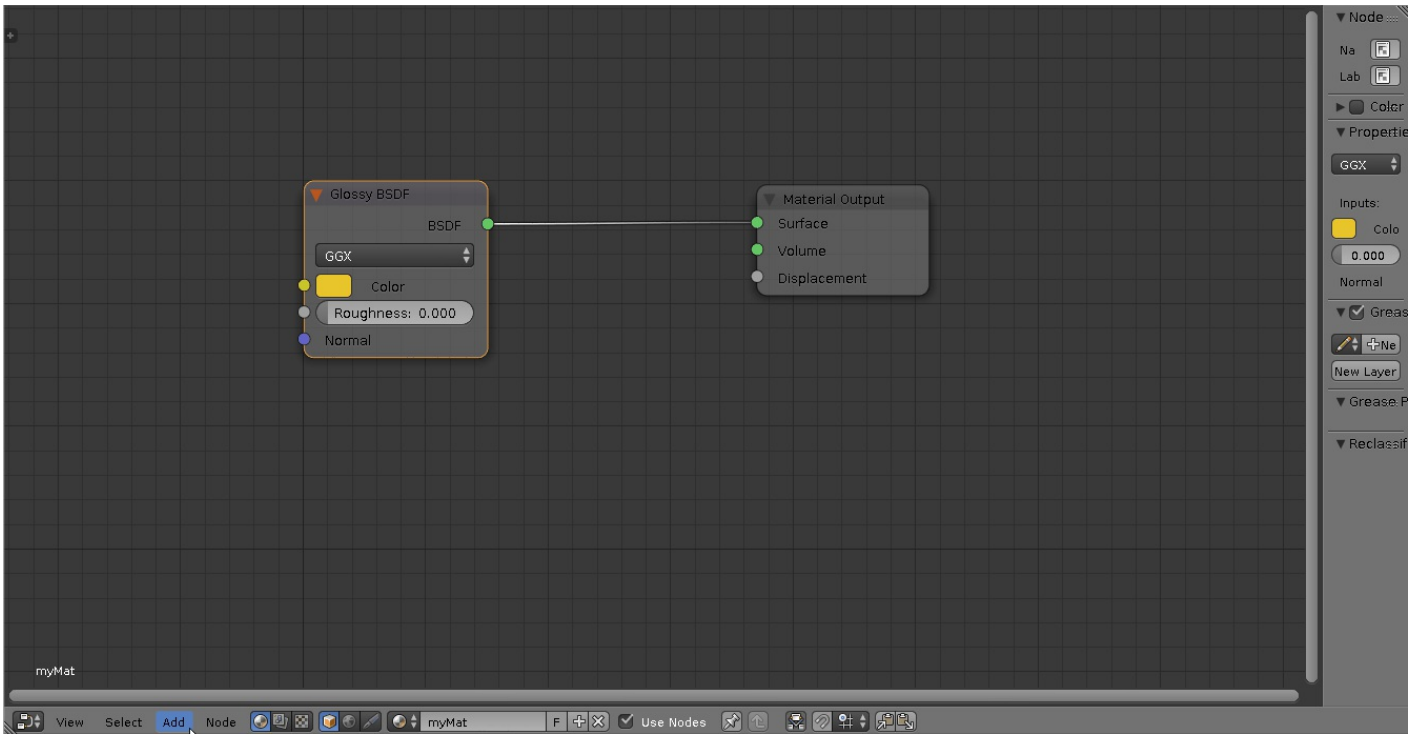
Node Editor (material)

In addition to creating materials as just described using all the settings on all the materials panels, in Blender you can create more sophisticated materials by routing basic materials through a set of nodes. Each node performs some operation on the material, changing how it will appear when applied to the mesh, and passes it on to the next node. In this way, very complex material appearances can be achieved.

`Example 5.` Setup a Mix Shader using node editor. In this example we mix a glossy shader with a diffuse shader to make a composite material.

- Right click on the Monkey object (*Suzanne*) to select it
- Switch the python console editor (bottom left area) to **Node editor** (figure below, left).
- In the node editor You will see the nodes we have already setup. The Glossy node shader's output is connected to the surface input of the Material output.
  We will now add two other shaders, a diffuse shader, and a mix shader.
- From the Noder Editor's bottom Header > **Add** > **Shader** > **Diffuse BSDF**
- From the Noder Editor's bottom Header >**Add** > **Shader** > **Mix shader**. You should be able to see both nodes have been added in to your node editor.
- Change the color value of the Diffuse node to *R: 0.075 G: 0.35 B: 0.50*
- Disconnect the **Glossy BSDF** input from the surface
- Connect BSDF output of both Diffuse and Glossy shaders to the inputs on the left side of the Mix (Shader)
- Connect Shader output (on the right side) to the Surface input of the Material output nodes (figure below, right).
- With the Fac parameter, you can adjust the mixture level.
- Your material should look like the right column of the above figure learn more about nodes


Editor Type
Python Console
File Browser
Info
User Preferences
Outliner
Properties
Logic Editor
Node Editor
Text Editor
Movie Clip Editor
Video Sequence Editor
UV/Image Editor
NLA Editor
Dope Sheet
Graph Editor
Timeline
3D View
View
Select
Add

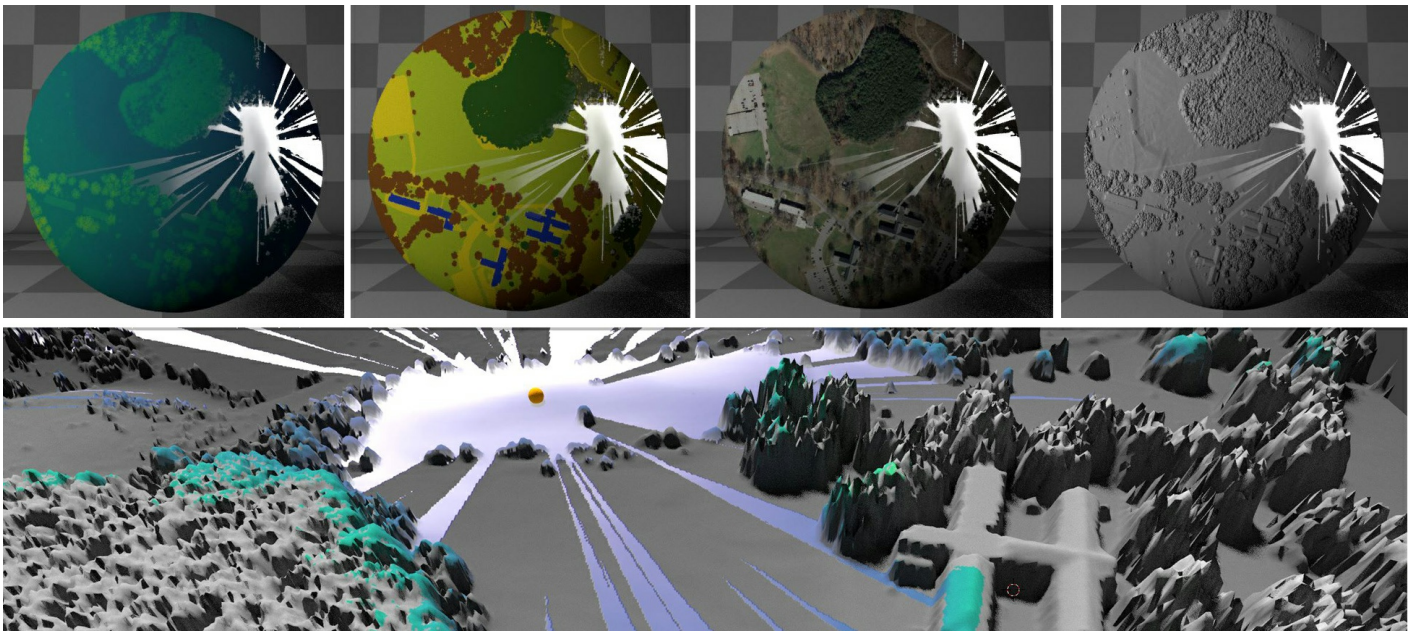


Node design w ith mix shader| |:---:|:---:|

Other Complementary resources for learning blender interface

Blender manual CG cookie

learn more about editors



Part 2. Processing, shading and rendering geospatial data

In this section w e w ill learn how to setup blender GIS addon, georeferences and importing raster and vector data and assigning simulations and orthophoto maps as textures. We w ill proceed w ith the instructions using a simple view shed assignment. The aim is to visualize and compare view sheds simulation computed for 4 different locations across a site. The general w orkflow is as follow ing.

I) Preparing the scene and lighting II) Georeferencing the scene III) Importing and processing the digital surface raster IV) Importing and processing the view point shapefile V) Draping the view shed map and orthophoto on the surface model

Note: View shed is a raster map show ing a surface's visible areas from a given locations.

There are two ways to complete the example; Scripting method (using blender's Python editor) and GUI (graphical user interface) method. For each step, the GUI procedure is listed as bullet points. Below that you can find the code snippet if you would like to follow the Scripting procedure. To execute the code snippet open a new text file in **Text Editor** and for each step directly copy-paste the code snippet into the editor and click on **Run Script** to execute the code.

Method	Duration	difficulty
GUI	1-2 hours	Complex
Python editor	10-15 minutes	Easy

Note For better learning complete the example with both methods but do not mix and match. Try to follow one method from the beginning to the end.

I. Setting up the scene

`GUI`

- Run Blender and open the file "Example_A.blend".
- Select the default **Cube** object in 3D viewport and delete it (right-click on the object > press delete > ok)
- Set **render engine** to "Cycles". You can find it in the top header, the default is "Blender Render"
- To increase the *Lamp* elevation and change the Lamp type to "Sun" for appropriate lighting:
  - Left click on the **Lamp** object in **Outliner** to select it
  - Go to **Properties editor** > **Object** (the orange cube icon) > **Transform** panel > in the **Location** matrix, change the *Z* value to 1000 (see below figure if needed)
- To change lamp type to Sun and increase the emission:
  - In **Properties editor** > **Lamp** (two icons to the right of the **Object** icon) > expand the **Lamp** panel > Change lamp type to *Sun*
  - Expand the **Nodes** panel > Click on **Use Nodes** to enable modifying Sun parameters.
  - Set the **Strength** parameter to 6.00

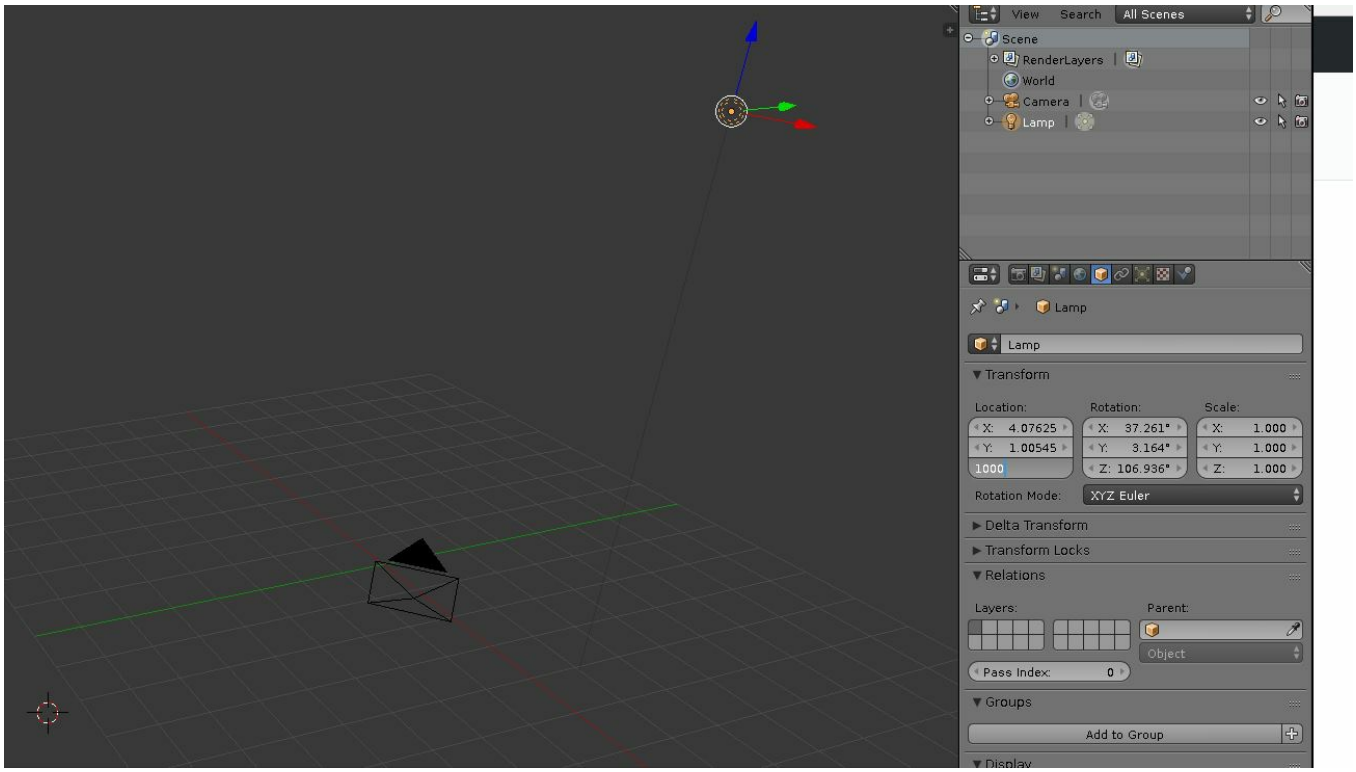
`Python editor`

```
import bpy
# remove the cube
cube = bpy.data.objects["Cube"]
cube.select = True
bpy.ops.object.delete()

# change lamp type and elevation
import bpy
lamp = bpy.data.lamps["Lamp"]
lamp.type = "SUN"
lampObj = bpy.data.objects["Lamp"]
lampObj.location[2] = 1000

# Setup node editor for lamp and increase the lamp power
lamp.use_nodes = True
lamp.node_tree.nodes["Emission"].inputs[1].default_value = 6

# Set render engine to cycles
bpy.context.scene.render.engine = 'CYCLES'
```



Changing the lamp elevation

II. Georeferencing the Blender Scene

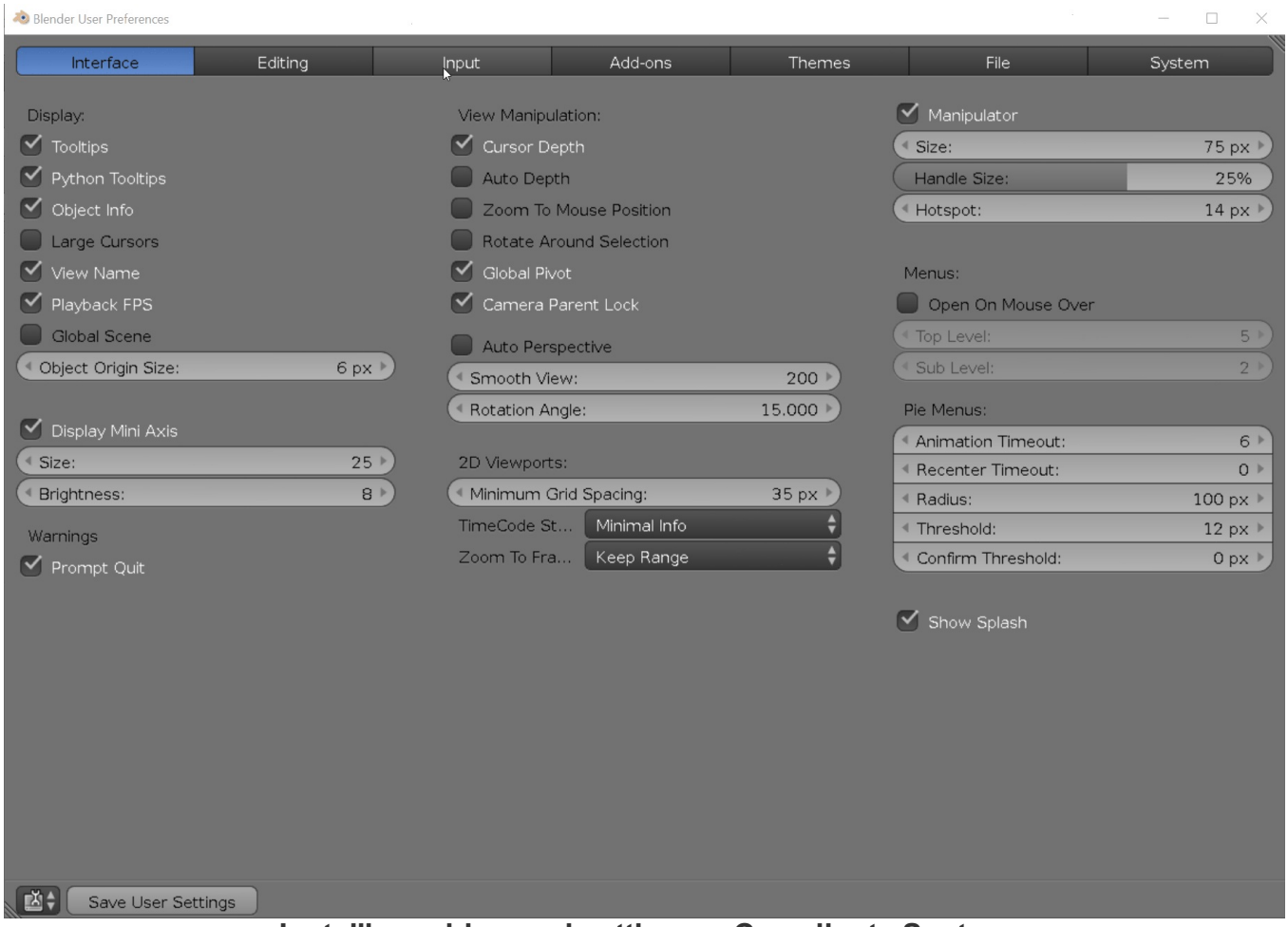
Setting up Blender GIS addon

- Dow nload the BlenderGIS addon
- Go to **file** > **user preferences** (`Alt + Ctrl + U`) > **Add-ons** > **Install from File** (bottom of the w indow)
- In the search tab, on top left type "gis" and make sure that in the **Categories** panel **All** is selected.
- In the search results you should be able to see **3Dview: BlenderGIS**. Select to load the addon.
- From the bottom of the preferences w indow click **Save User Settings** so the addon is loaded next time you open blender

Adding a new predefined coordinate reference system (CRS)

Before setting up the coordinate reference system of the Blender scene and configuring the scene projection, you should know the Coordinate Reference System (CRS) and the Spatial Reference Identifier (SRID) of your project. You can get the SRID from http://epsg.io/ or spatial reference w ebsite using your CRS. The example datasets in this exercise uses a NAD83(HARN)/North Carolina CRS (SSRID EPSG: 3358)

- In BlenderGIS add-on panel (in preferences w indow s), select to expand the **3D View: BlenderGIS**
- In the preferences panel find **Spatial Reference Systems** and click on the **+ Add** button
- In the add w indow put "3358" for **definition** and "NAD83(HARN)/North Carolina" for **Description**. Then select **Save to addon preferences**
- Select **OK** and close the User Preferences w indow



Installing addon and setting up Coordinate System

Learn more about Georefencing management in Blender

Setting up coordinate system

`GUI`

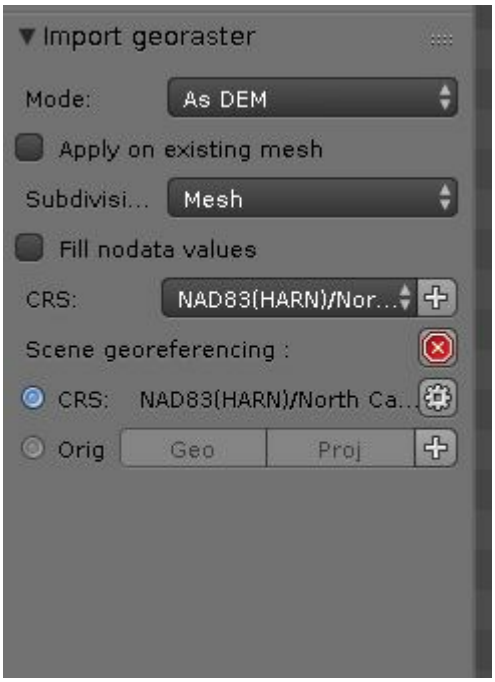
- Find and click on GIS addon's interface in 3D view port's left toolbar. In the "Geoscene" panel , click on the gear shape icon and switch to NAD83(HARN), click ok.

Georeferencing setup in Blender GIS

III. Importing digital surface model

`GUI`

- Go to **file** > **import** > **Georeferenced Raster**
- On the bottom left side of the window find **Mode** and select **As DEM**
- Set **subdivision** to *Mesh* and select *NAD83(HARN)* for georeferencing
- Browse to the 'ICC_workshop' folder and select 'example1_dsm.tif'
- Click on **Import georaster** on the top right header
- If all the steps are followed correctly, you should be able to see the terrain in 3D view window



Georaster import parameters

`Python editor`

```
import bpy
import os
filePath = os.path.dirname(bpy.path.abspath("//"))
fileName = os.path.join(filePath,'example1_dsm.tif')
bpy.ops.importgis.georaster(filepath=fileName,
                            importMode="DEM", subdivision="mesh",
                            rastCRS="EPSG:3358")
```

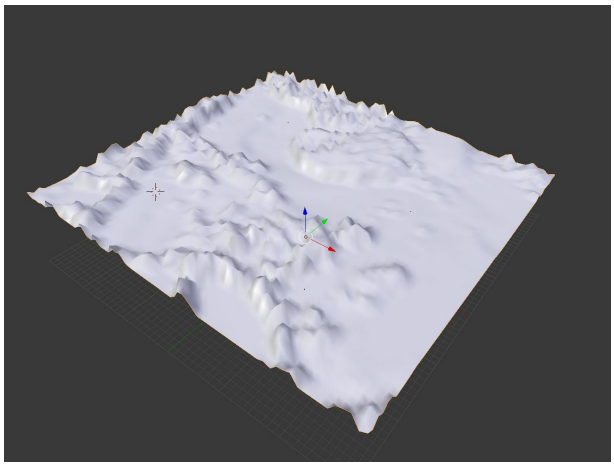
Surface subdivision and refinement

`GUI`

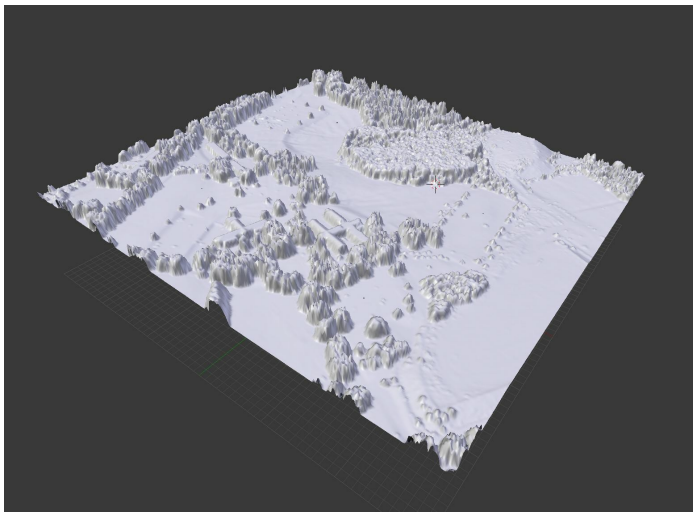
- Select surface model (right click on the object)
- Go to **3D view** editor's bottom toolbar > **Object interaction mode** > **Edit Mode**
- Switch to **Face select**
- If object is not orange in color (i.e., nothing is selected), go to **Select** > **(De)select All** (or press `A`) to select all faces (when the object faces are selected, they will turn orange)
- Go to **Tools** (left toolbar) > **Mesh Tools** > **Subdivide** . The subdivide dialogue should appear on the bottom left on the toolbar. Type "4" in the number of cuts tab
- Go to **3D view** editor's bottom toolbar > **Object interaction mode** > **Object Mode** . You should be able to see the surface details at this point (bottom figure, right image).

`Python editor`

```
import bpy
bpy.ops.object.mode_set(mode='EDIT')
bpy.ops.mesh.select_all(action='SELECT')
bpy.ops.mesh.subdivide(number_cuts=4, smoothness=0.2)
bpy.ops.object.mode_set(mode='OBJECT')
```



DSM surface after importing



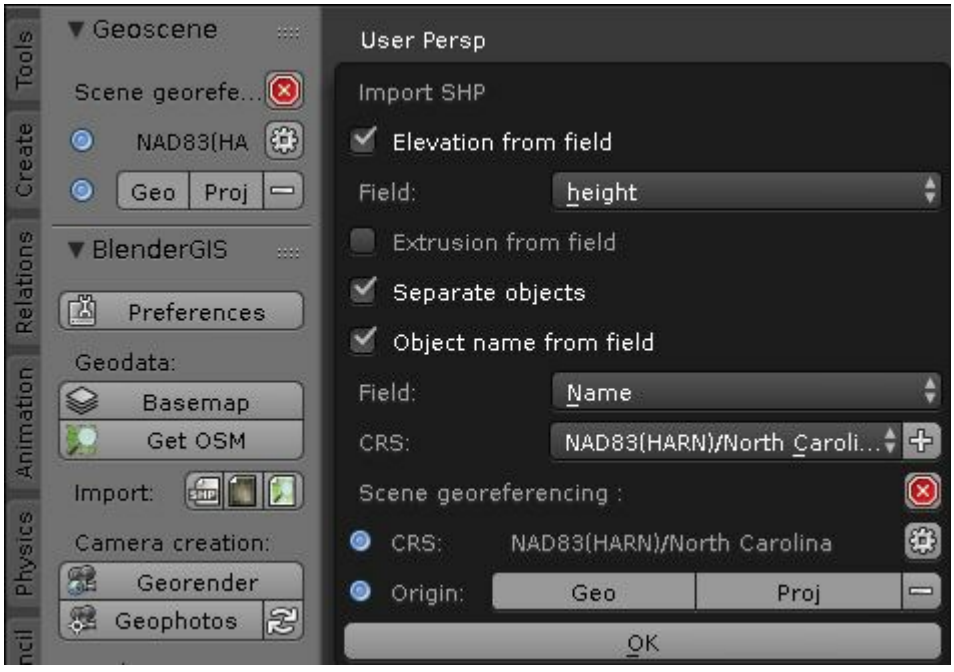
DSM surface after subdivision

IV. Importing viewpoint shapefile

In this step we will import view point locations as a point feature shapefile. The features represent the location from which the view sheds are computed on the digital surface.

`GUI`

- Import view point shape file
  - Go to **file** > **import** > **Shapefile**
  - Browse workshop data directory, select *vpoints.shp* and click on **Import Shp** . The shape import dialogue should appear in front of the GIS addon interface.
  - Activate “Elevation from field” and in field panel select “height”
  - Activate “Separate objects”
  - Activate “Object name from field” and in field panel select “Name”, you should be able to see 4 the points on the surface and 4 objects added to the Outliner with the names *Viewshed_1, Viewshed_2,Viewshed_3, Viewshed_4*
  - Select **OK**



Blender Gis shape import dialogue

`Python editor`

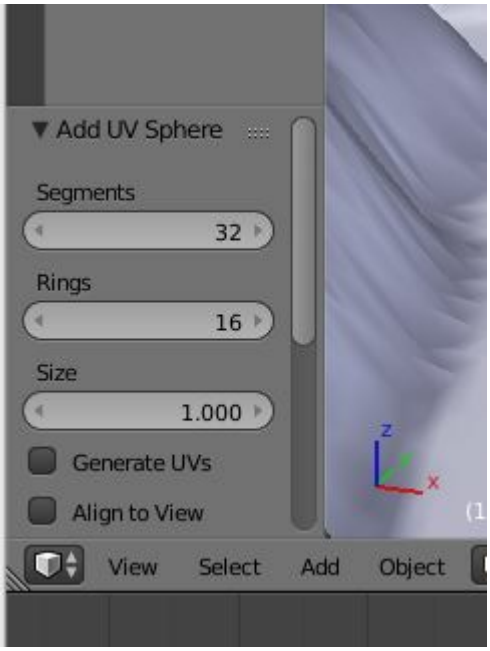
```
import bpy
import os
filePath = os.path.dirname(bpy.path.abspath("//"))
fileName = os.path.join(filePath,'vpoints.shp')
bpy.ops.importgis.shapefile(filepath=fileName,fieldElevName="height",fieldObjName='Name',separateObjects=True,shpCRS='EPSG:3358')
```

Creating viewpoint markers

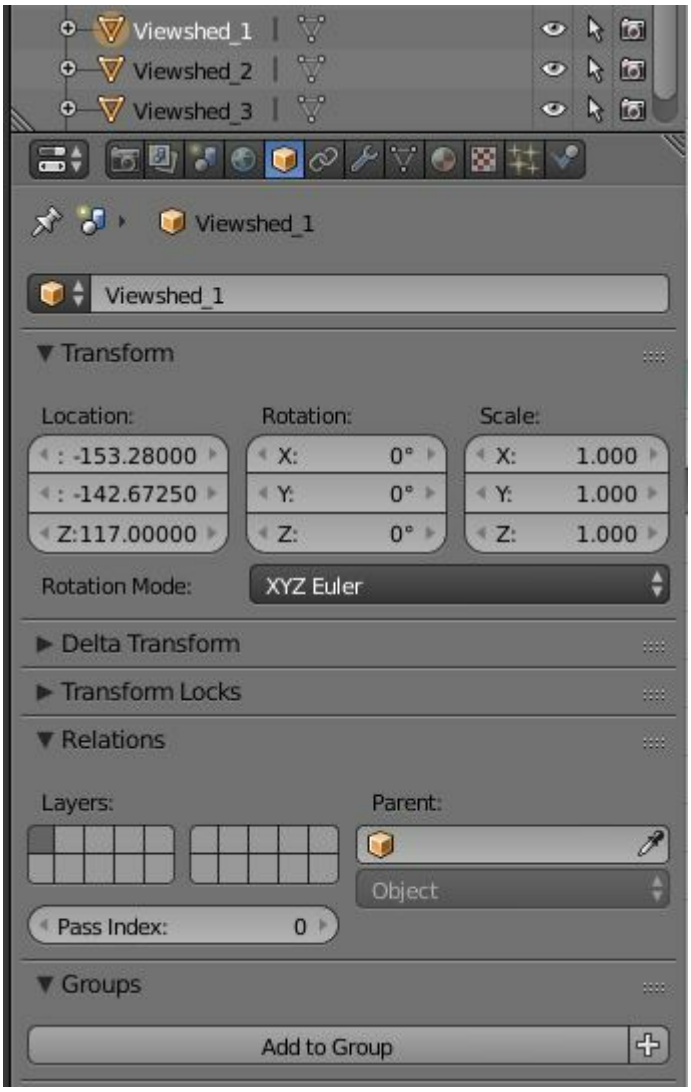
Imported points are 2D vectors that cannot be rendered as they don't have actual surfaces. Now we create 4 small spheres and match their location with the imported points to visualize observer locations in 3D.

`GUI`

- To create spheres on the view point location:
  - Go to 3D View port's **bottom header** > **Add** > **Mesh** > **UV sphere**. The Add UV sphere dialogue will open on the left side of the Toolbar. Set the Size parameter to 3.000
  - Select Sphere object (by clicking on it in **Outliner**) and press Shift + D or ctrl+c , ctrl+v to make a copy of the object, you should see the *Sphere.001* in the outliner Make 3 copies of the sphere and rename them to *Sphere1, Sphere2, ... , Sphere4*
  - From **Outliner** select the object *Viewshed_1*
  - Go to **Properties Editor** > **Object** > **Transform** > **Location** to retrieve the view shed point's coordinates (X,Y,Z)
- To move each of the 4 spheres to the corresponding view shed location:
  - Copy and paste the retrieved coordinates from View shed_1 into the location parameters for Sphere1
  - Add 2.0 extra units to the Z parameter (only for **Location**) to raise the spheres above the ground
  - Repeat this process for each View shed and each Sphere
  - You should now have 4 spheres aligned on the imported view shed points.



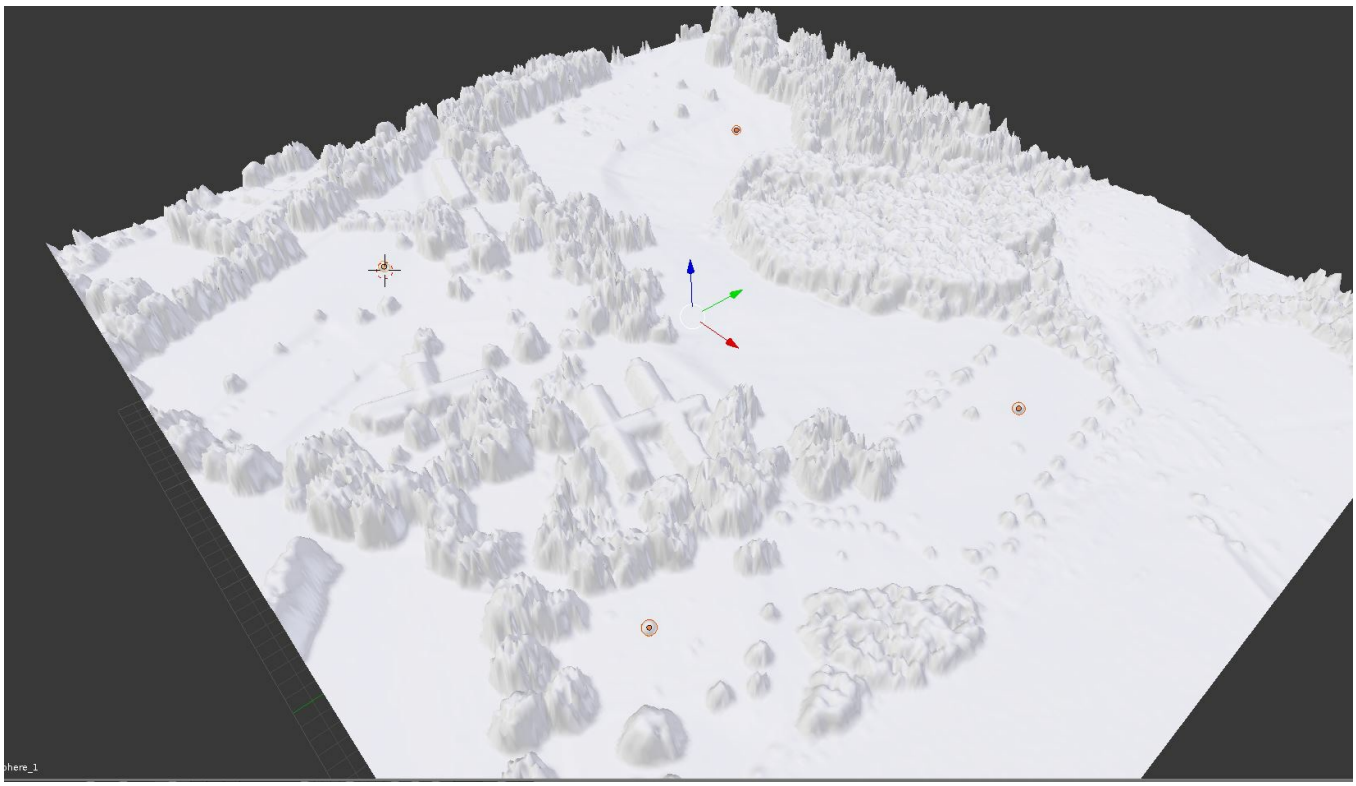
UV Sphere toolbar



Object transform panel in properties editor

`Python editor`

```
import bpy
# get get viewpoint objects create a sphere using their X,Y and Z+2 coordinates
for obj in bpy.data.objects:
    if "Viewshed" in obj.name:
        bpy.ops.mesh.primitive_uv_sphere_add(size=3.0, location=(obj.location[0],obj.location[1],obj.location[2]+2))
        sphere = bpy.context.active_object
        # rename the sphere
        sphere.name = "Sphere" + obj.name[-2:]
```



4 spheres representing observation points

Generating 4 copies of the surface and viewpoint spheres

In this step w e create 4 copies of the surface model and move each of the view point spheres to the corresponding surface

`GUI`

- Select DSM object and press `Shift + D` or `ctrl+c` , `ctrl+v` to make a copy of the object , you should see the example1_dsm.001 in the outliner
- Select the "example1_dsm001"
- While holding shift key select "Sphere_1" from outline to select both DSM and corresponding Sphere
- go to **Properties Editor** > **Object** (cube icon)
- In the **Transform** panel > **Location** > **X:** type *750* to move the duplicated surface 750 meters to the east
- Create another copy of the DSM , put -750 for Y parameter to move the duplicate surface 750 meters to the south
- Create another copy of the DSM, put 750 for X parameter and -750 in Y parameter. The final model should look like figure

`Python editor`

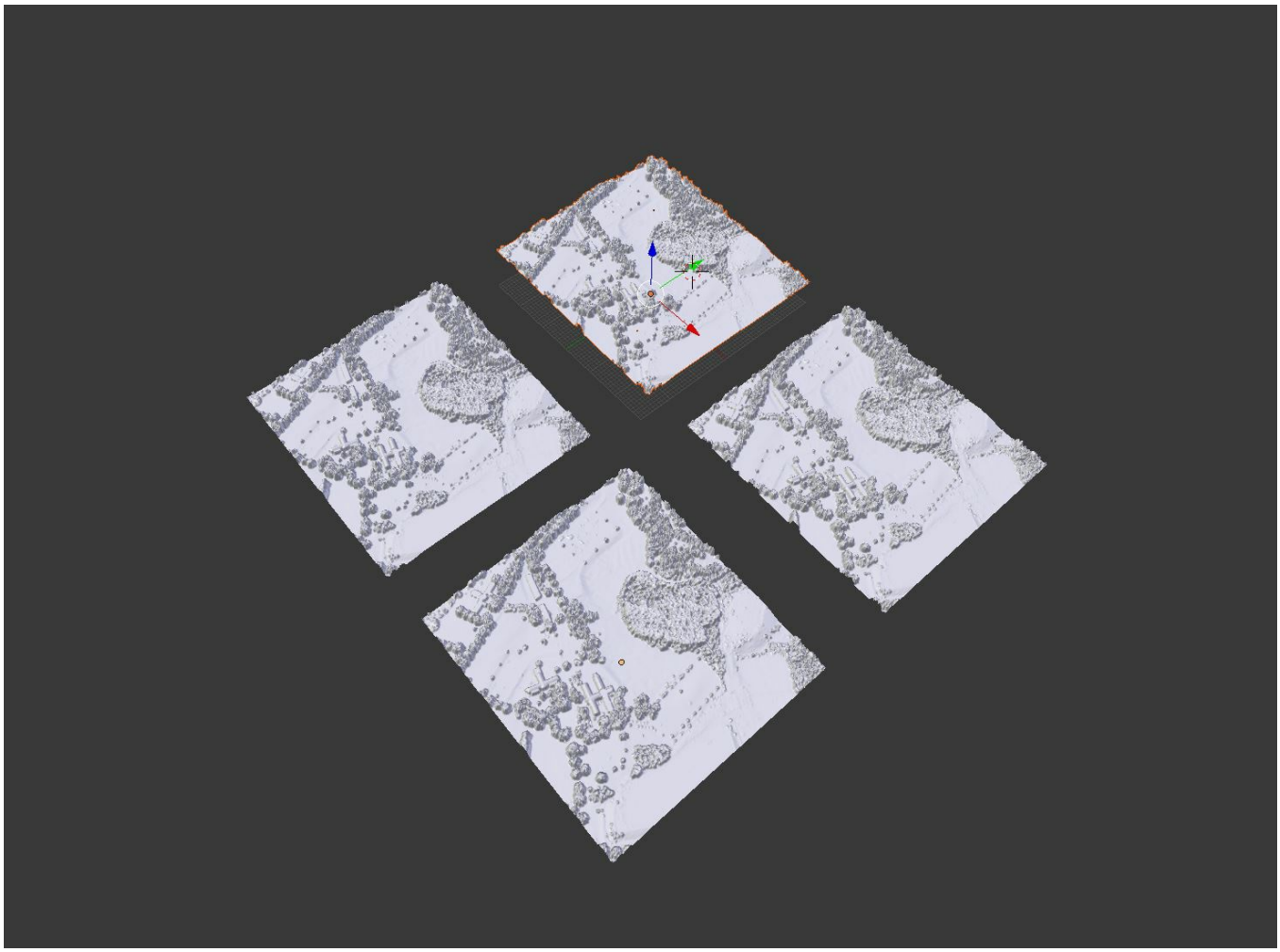
```
import bpy

for ob in bpy.data.objects:
    ob.select = False
obj = bpy.data.objects["example1_dsm"]
obj.name = "example1_dsm1"
obj.select = True
```

```
# create and rename 3 replicates of DSM, and move spheres to create 4 DSMs and spheres
bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(750, 0, 0 )})
bpy.data.objects ["example1_dsm1.001"].name = "example1_dsm2"
sphere2Obj = bpy.data.objects ["Sphere_2"]
loc = sphere2Obj.location
sphere2Obj.location = (loc[0]+750, loc[1], loc[2])

bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(0, -750, 0 )})
bpy.data.objects ["example1_dsm2.001"].name = "example1_dsm3"
sphere3Obj = bpy.data.objects ["Sphere_3"]
loc = sphere3Obj.location
sphere3Obj.location = (loc[0]+750, loc[1]-750, loc[2])

bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(-750, 0, 0 )})
bpy.data.objects ["example1_dsm3.001"].name  = "example1_dsm4"
sphere4Obj = bpy.data.objects ["Sphere_4"]
loc = sphere4Obj.location
sphere4Obj.location = (loc[0],loc[1]-750, loc[2])
```



Replicated models

V. Shading the scene

Now we will create a mixed material to combine Orthophoto and viewshed maps. We will use emission shaders to show viewsheds as glowing surfaces.

`GUI`

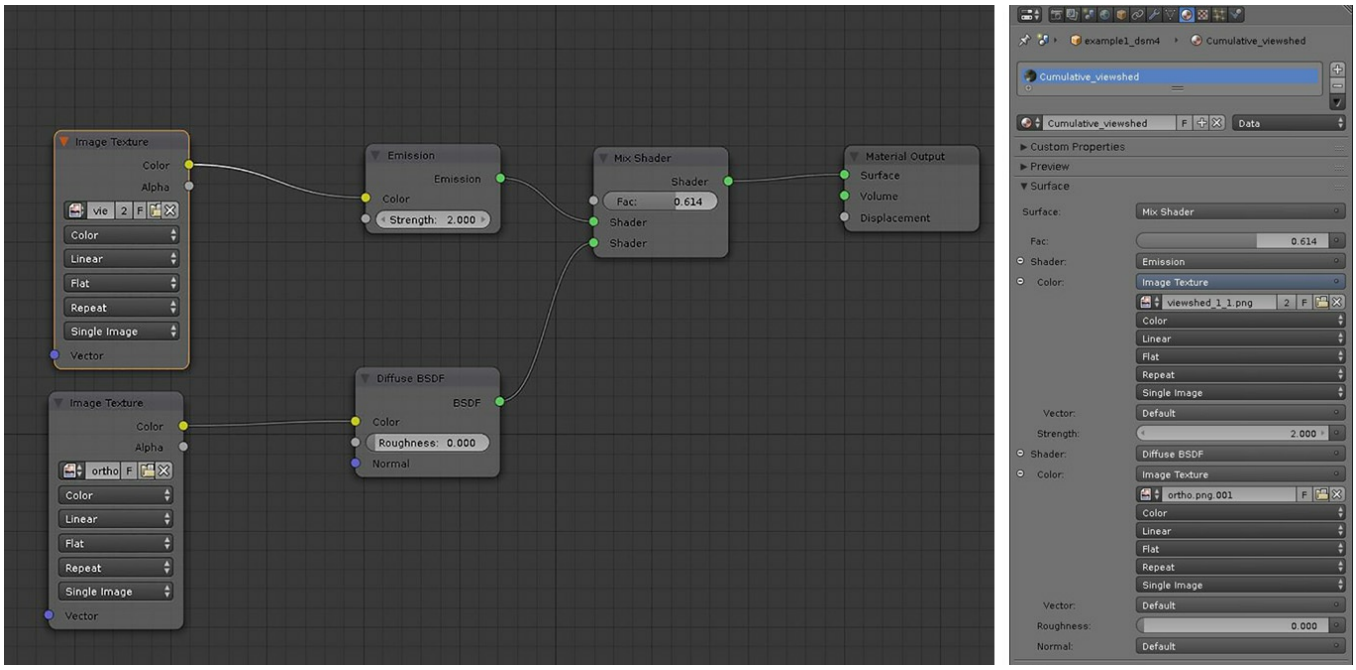
- Make sure that the **Render engine** is set to *Cycles* and 3D viewport **Shading** is set to *Material*
- Change the bottom editor panel to **Node editor**. This can be done by simply changing the Editor Type selector (bottom left hand side of the window).
- Select the first DSM object "example_dsm1"
- Go to **Properties tab** > **Material** Press **+ New** button to add material

- Rename the Material to "View shed1"
- Expand the **Surface** panel and click on the gray square shaped icon on the right side of the **Surface** parameter to see a popup w indow w ith texture parameters. Select **Mix Shader** . You should be able to see tw o **Shaders** added below the mix shader.

- Click on the first shader and select *Emission* from the dropdow n list
  - Click on the radio button on the right side of the **color** field > **texture** > **Image texture**
  - Click on **Open** and load "view shed_1_1.png". You should be able to see the view shed draped on the DSM surface
  - Change the **Strength** slider to 1.8 to increase the view shed's emission pow er
- Click on the second shader and select *Diffuse BSDF* from the dropdow n list
  - Click on the radio button on the right side of the **color** field > **texture** > **Image texture**
  - Click on **Open** and load "ortho.png". You should be able to see the view shed draped on the DSM surface

Now notice how the material logic and w orkflow is represented in Node editor. You can play w ith each of the individual nodes ,the links betw een them and the values.

- Play w ith the **Fac** slider on the **Mix shader** node to adjust the mixture level
- Repeat the shading procedure for the other 3 objects using "View shed_1_2.png", "View shed_1_3.png", "View shed_1_4.png"



Node editor and Properties panel

`Python editor`

```
import bpy
import os
filePath = os.path.dirname(bpy.path.abspath("//"))
ortho = os.path.join(filePath, 'ortho.png')

for obj in bpy.data.objects:
    if "dsm" in obj.name :
        obj.select = True
        fileName = "viewshed_{0}_1.png".format(obj.name[-1:])
        matName = os.path.join(filePath, fileName)
# Create a new material and assign it to the DSM object # 3

        mat = (bpy.data.materials.get(matName) or
               bpy.data.materials.new(matName))

        obj.data.materials.append(mat)

        # Get material tree , nodes and links #
        mat.use_nodes = True
        node_tree = mat.node_tree
        nodes = node_tree.nodes
        links = node_tree.links
        for node in nodes:
            nodes.remove(node)
```

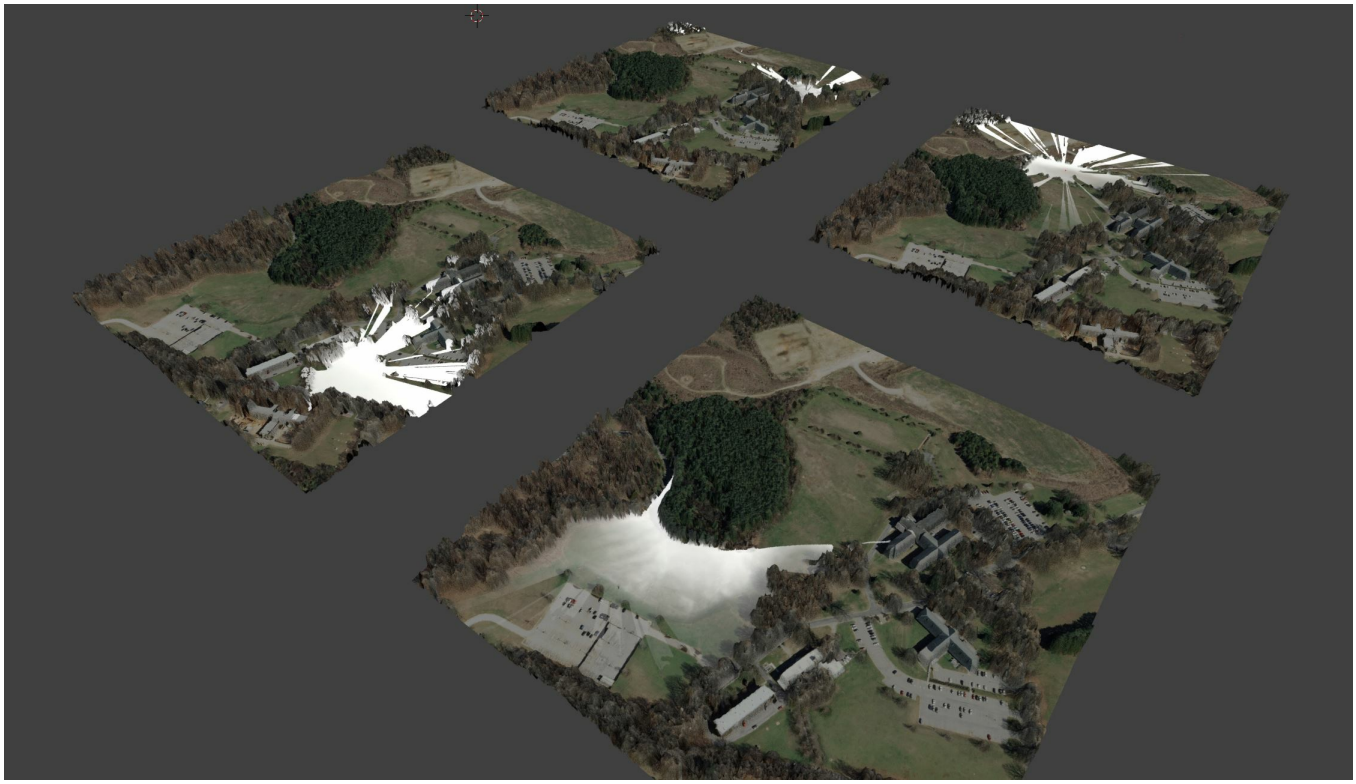
```
diffuseNode = node_tree.nodes.new("ShaderNodeBsdfDiffuse")
diffuseNode.location = (300,400)
#diffuseNode = nodes["Diffuse BSDF"]

# Add a new texture for viewshed #
viewshedNode = node_tree.nodes.new("ShaderNodeTexImage")
viewshedNode.select = True
node_tree.nodes.active = viewshedNode
viewshedNode.image = bpy.data.images.load(matName)
viewshedNode.location = (100, 100)

# Add a new texture for ortho #
orthoNode = node_tree.nodes.new("ShaderNodeTexImage")
orthoNode.select = True
node_tree.nodes.active = orthoNode
orthoNode.image = bpy.data.images.load(ortho)
orthoNode.location = (100, 400)
# Add a new mixshader node and link it to the diffuse color node#

mixShaderNode = node_tree.nodes.new("ShaderNodeMixShader")
mixShaderNode.location = (600, 250)
mixShaderNode.inputs["Fac"].default_value = .7
emissionNode = node_tree.nodes.new("ShaderNodeEmission")
emissionNode.location = (300, 100)
outputNode = node_tree.nodes.new("ShaderNodeOutputMaterial")
outputNode.location = (800, 250)
emissionNode.inputs[1].default_value = 2

links.new (viewshedNode.outputs["Color"], emissionNode.inputs["Color"])
links.new (orthoNode.outputs["Color"], diffuseNode.inputs["Color"])
links.new (emissionNode.outputs["Emission"], mixShaderNode.inputs[2])
links.new (diffuseNode.outputs["BSDF"], mixShaderNode.inputs[1])
links.new (mixShaderNode.outputs["Shader"], outputNode.inputs["Surface"])
```



Viewshed and Orthophoto draped on DSM surface using Mix shader

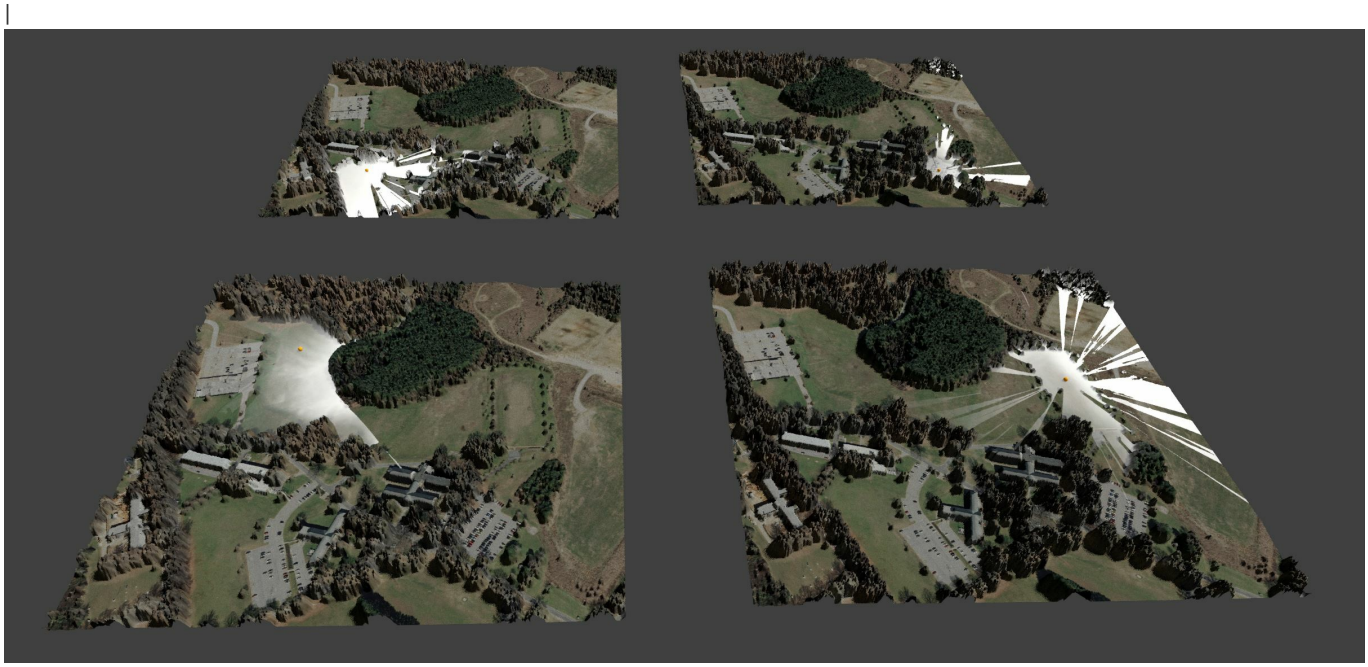
Shading Viewpoints

Now follow the same w orkflow to shade view point spheres but this time only use diffuse node (*Diffuse BSDF*) a w ith solid orange color.

- Select the first sphere, create a new material using nodes
- Change the surface color to
- repeat the same for all spheres

```
import bpy
for obj in bpy.data.objects:
  if "Sphere" in obj.name:
      obj.select = True
      matName = "sphere"
  # Create a new material and assign it to the DSM object # 3
      mat = (bpy.data.materials.get(matName) or
             bpy.data.materials.new(matName))
      obj.data.materials.append(mat)
      # Get material tree , nodes and links #
      mat.use_nodes = True
      node_tree = mat.node_tree
      nodes = node_tree.nodes
      links = node_tree.links
      nodes[1].inputs[0].default_value = (.8, .3, 0, 1)
      obj.select = False
```

|



View port render of the view shed | |:---:|

VI. Modelling made easy

Now lets try to run the entire procedure w ith a python file using **Text editor** and **Python console**

- From top header goto **file**> **New** to Open a fresh copy of Blender
- Save the blender file w ith you preferred name in the w orkshop directory. **Note**: This is an important step since your all the paths in python code are linked to that directory
- In the top header find **Layout** (right next to **help**) and sw itch the layout to *Scripting* The scripting layout includes : a **Text editor**(left), a **Python Console** (bottom) and **3D view** (right)
- Procedure for **Text editor**
  - In **Text editor** > **Open** > Go to w orkshop directory and find *example_a.py*
  - Click on **run script**
- Procedure for **Python Console**
  - type the follow ing lines in the console. Note that you need to type in the w orkshop path in you computer in the first line.

`Python Console>>>`

```
filename = "/full/path/to/example_a.py"
exec(compile(open(filename).read(), filename, 'exec'))
```

`Python editor`

```
import bpy
```

```
import os

filePath = os.path.dirname(bpy.path.abspath("//"))
dsmName = os.path.join(filePath,'example1_dsm.tif')
imgPath = os.path.join(filePath,'cumulative_viewshed.png')
vpointName = os.path.join(filePath,'vpoints.shp')
ortho = os.path.join(filePath, 'ortho.png')

# Setting up the scene
if bpy.data.objects.get("Cube"):
    cube = bpy.data.objects["Cube"]
    cube.select = True
    bpy.ops.object.delete()

# change lamp type and elevation
import bpy
lamp = bpy.data.lamps["Lamp"]
lamp.type = "SUN"
lampObj = bpy.data.objects["Lamp"]
lampObj.location[2] = 1000

# Setup node editor for lamp and increase the lamp power
lamp.use_nodes = True
lamp.node_tree.nodes["Emission"].inputs[1].default_value = 6

# Set render engine to cycles
bpy.context.scene.render.engine = 'CYCLES'
bpy.ops.importgis.georaster(filepath=dsmName,
                            importMode="DEM", subdivision="mesh",
                            rastCRS="EPSG:3358")

# Subdivision render engine to cycles
bpy.ops.object.mode_set(mode='EDIT')
bpy.ops.mesh.select_all(action='SELECT')
bpy.ops.mesh.subdivide(number_cuts=4, smoothness=0.2)
bpy.ops.object.mode_set(mode='OBJECT')

# import viewpoints
bpy.ops.importgis.shapefile(filepath=vpointName,fieldElevName="height",fieldObjName='Name',separateObjects=True,shpCRS='EPSG:3358')

# adding spheres
for obj in bpy.data.objects:
    if "Viewshed" in obj.name:
        bpy.ops.mesh.primitive_uv_sphere_add(size=3.0, location=(obj.location[0],obj.location[1],obj.location[2]+2))
        sphere = bpy.context.active_object
        # rename the sphere
        sphere.name = "Sphere" + obj.name[-2:]

for ob in bpy.data.objects:
    ob.select = False
obj = bpy.data.objects["example1_dsm"]
obj.name = "example1_dsm1"
obj.select = True

# create and rename 3 replicates of DSM, and move spheres to create 4 DSMs and spheres

bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(750, 0, 0 )})
bpy.data.objects ["example1_dsm1.001"].name = "example1_dsm2"
sphere2Obj = bpy.data.objects ["Sphere_2"]
loc = sphere2Obj.location
sphere2Obj.location = (loc[0]+750, loc[1], loc[2])

bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(0, -750, 0 )})
bpy.data.objects ["example1_dsm2.001"].name = "example1_dsm3"
sphere3Obj = bpy.data.objects ["Sphere_3"]
loc = sphere3Obj.location
sphere3Obj.location = (loc[0]+750, loc[1]-750, loc[2])

bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(-750, 0, 0 )})
bpy.data.objects ["example1_dsm3.001"].name  = "example1_dsm4"
sphere4Obj = bpy.data.objects ["Sphere_4"]
loc = sphere4Obj.location
sphere4Obj.location = (loc[0],loc[1]-750, loc[2])

# Shading Viewsheds
for obj in bpy.data.objects:
    if "dsm" in obj.name :
        obj.select = True
        fileName = "viewshed_{0}_1.png".format(obj.name[-1:])
        matName = os.path.join(filePath, fileName)
```

```
# Create a new material and assign it to the DSM object # 3

        mat = (bpy.data.materials.get(matName) or
               bpy.data.materials.new(matName))

        obj.data.materials.append(mat)

        # Get material tree , nodes and links #
        mat.use_nodes = True
        node_tree = mat.node_tree
        nodes = node_tree.nodes
        links = node_tree.links
        for node in nodes:
            nodes.remove(node)
        diffuseNode = node_tree.nodes.new("ShaderNodeBsdfDiffuse")
        diffuseNode.location = (300,400)
        #diffuseNode = nodes["Diffuse BSDF"]

        # Add a new texture for viewshed #
        viewshedNode = node_tree.nodes.new("ShaderNodeTexImage")
        viewshedNode.select = True
        node_tree.nodes.active = viewshedNode
        viewshedNode.image = bpy.data.images.load(matName)
        viewshedNode.location = (100, 100)

        # Add a new texture for ortho #
        orthoNode = node_tree.nodes.new("ShaderNodeTexImage")
        orthoNode.select = True
        node_tree.nodes.active = orthoNode
        orthoNode.image = bpy.data.images.load(ortho)
        orthoNode.location = (100, 400)
        # Add a new mixshader node and link it to the diffuse color node#

        mixShaderNode = node_tree.nodes.new("ShaderNodeMixShader")
        mixShaderNode.location = (600, 250)
        mixShaderNode.inputs["Fac"].default_value = .7
        emissionNode = node_tree.nodes.new("ShaderNodeEmission")
        emissionNode.location = (300, 100)
        outputNode = node_tree.nodes.new("ShaderNodeOutputMaterial")
        outputNode.location = (800, 250)
        emissionNode.inputs[1].default_value = 2

        links.new (viewshedNode.outputs["Color"], emissionNode.inputs["Color"])
        links.new (orthoNode.outputs["Color"], diffuseNode.inputs["Color"])
        links.new (emissionNode.outputs["Emission"], mixShaderNode.inputs[2])
        links.new (diffuseNode.outputs["BSDF"], mixShaderNode.inputs[1])
        links.new (mixShaderNode.outputs["Shader"], outputNode.inputs["Surface"])
# shading viewpoints
for obj in bpy.data.objects:
    if "Sphere" in obj.name:
        obj.select = True
        matName = "sphere"
    # Create a new material and assign it to the DSM object # 3
        mat = (bpy.data.materials.get(matName) or
               bpy.data.materials.new(matName))
        obj.data.materials.append(mat)
        # Get material tree , nodes and links #
        mat.use_nodes = True
        node_tree = mat.node_tree
        nodes = node_tree.nodes
        links = node_tree.links
        nodes[1].inputs[0].default_value = (.8, .3, 0, 1)
        obj.select = False

# Switch to shading mode
for area in bpy.context.screen.areas:
    if area.type == 'VIEW_3D':
          for space in area.spaces:
              if space.type == 'VIEW_3D':
                  space.viewport_shade = 'MATERIAL'
```

Acknowledgment

This work is built upon great contributions and support of Blender team, Blender GIS addon developers (domlysz/BlenderGIS) , Center for Geospatial Analytics, NC State's Geoforall lab and Garrett Millar.