Guidelines for City Builders

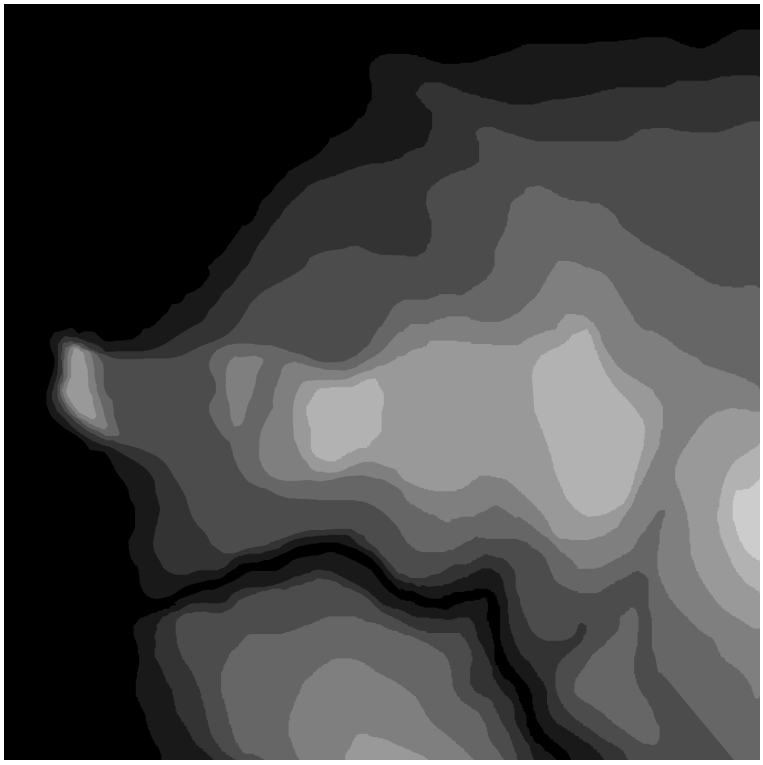
With examples from the Gancaloon Project

The techniques described below suppose the use of the following programs:

- Gimp or any other paint program;
- GeoControl to generate height_field maps. Other programs like Leveller may do the same thing though. Such a program is quite essential for generating detailed landscapes and road tracks;
- Inkscape for generating POV-Ray prisms from image_maps;
- A modeller like Moray, Blender, SiloPro, Wings3D, or Kerkythea for building mesh objects, and Poseray for converting those to mesh2{} format;
- POV-Ray for building and rendering scenes.

1. Landscape

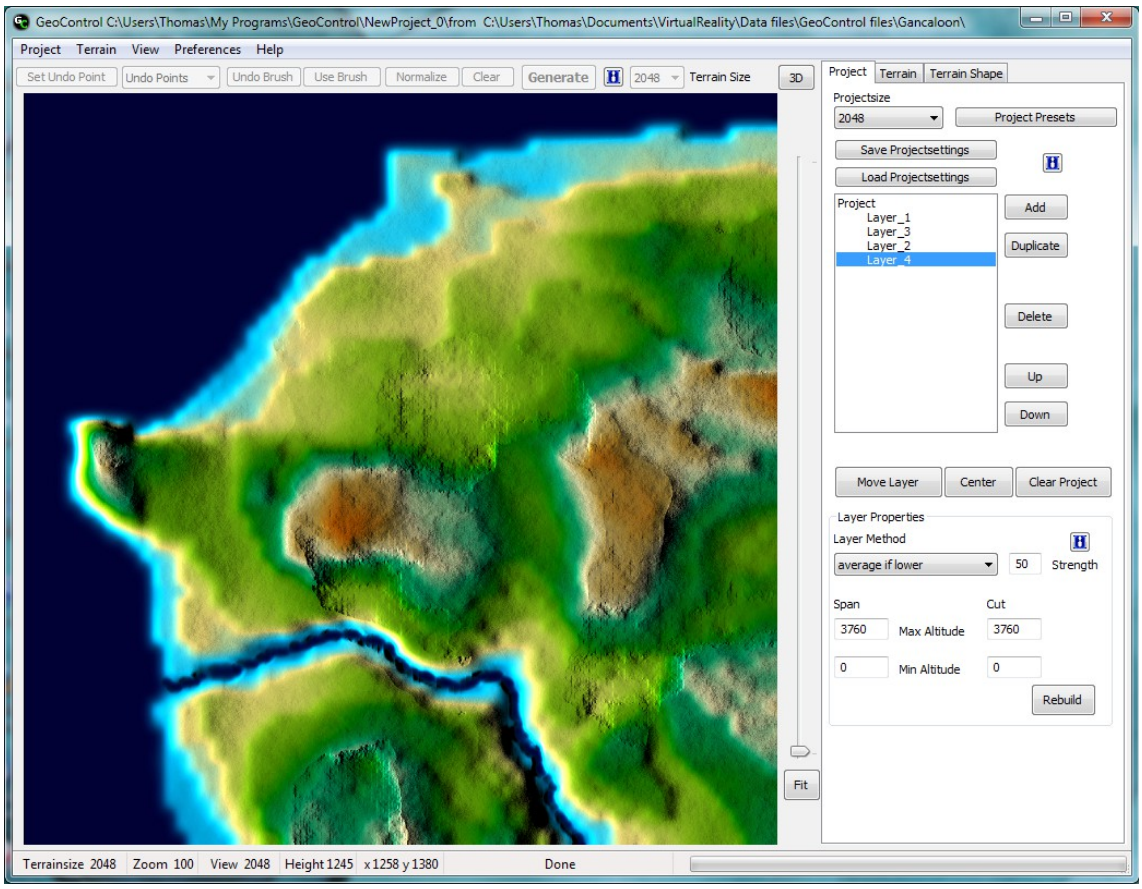
Step 1: Drawing a contour map of the landscape using Gimp.



The contour map can be as detailed as necessary but serves as basis for further processing. A large map (here 2048x2048 pixels) is better than a smaller one of course.

Step 2: The contour map is imported in the landscape program GeoControl and processed using different layers (each containing the same contour map!) and different filters, until an acceptable landscape is obtained. The result is exported in tga format.

© Thomas de Groot, 2012

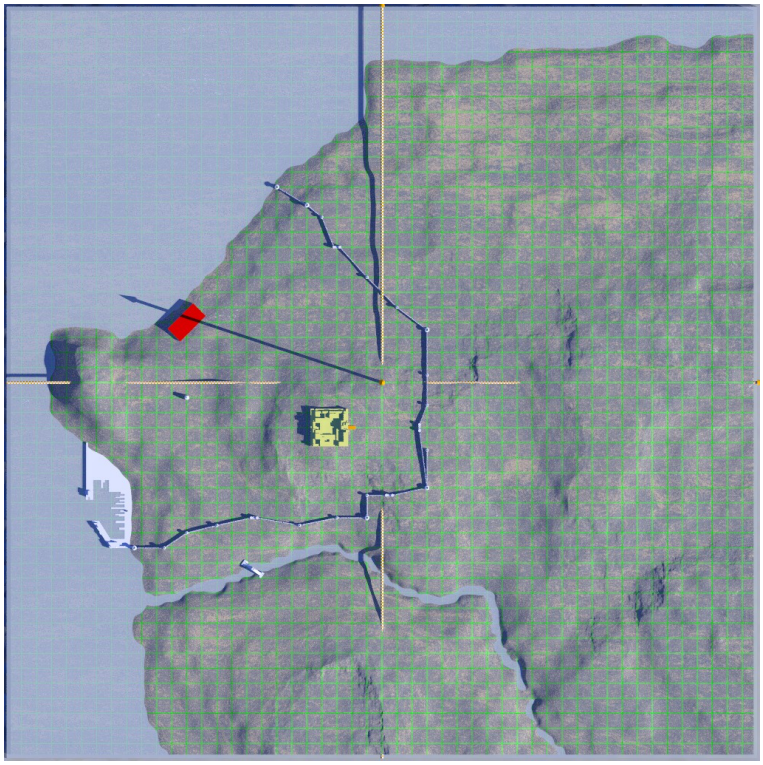


Step 3: In POV-Ray, the image_map is turned into a height_field function. Example:

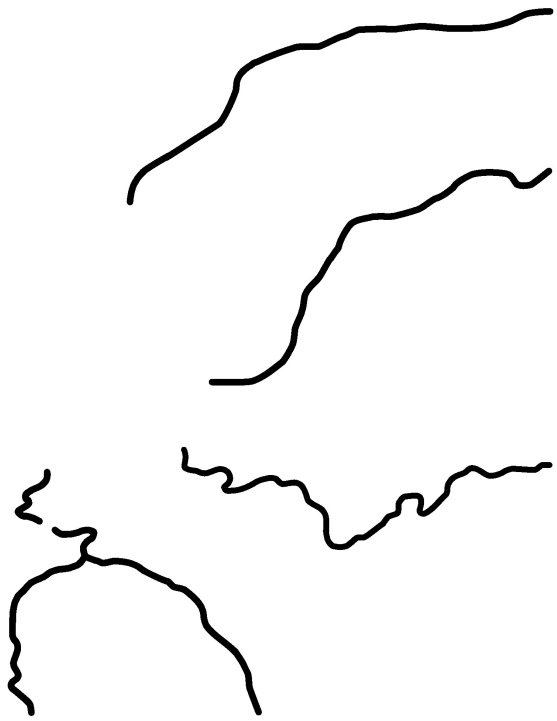
```
#declare F_topo =
function {
  pigment {
    image_map {
      tga "MyLandscape.tga" gamma 1.0
      map_type 0  interpolate 2
    }
    warp {repeat x}  warp {repeat y}
    scale 50
    warp {
      turbulence 0.2
      octaves 1 //[6]
      lambda 1  //[2]
      omega 0.2 //[0.5]
    }
    scale 1/50
  }
}
#declare MyHeightfield =
  height_field {
    function MyRes, MyRes {F_topo(x,y,z).hf}
    smooth
    translate <-0.5, 0, -0.5>
    transform {MyTrans}
  }
```

© Thomas de Groot, 2012

The landscape textures (using slope{}) are then also added. As a help, a raster and axes can be added to the landscape before rendering an orthographic view. Example:



Step 4: Use the orthographic view of the landscape to draw roads. Example:



© Thomas de Groot, 2012

This roads image_map is then used as an additional layer in GeoControl to “erode” the roads into the landscape. To do this:

1. Go to the Project tab and add a layer.
2. Within that layer, go to the Terrain tab and import the road image_map. Do not change anything else and do not press the Generate button.
3. Go back to the Project tab. If roads are black on white, set Layer Method to “add”, else to “substract”. Set Strength to 1 or 2.
4. Press the Rebuild button.

An identical roads image_map but coloured differently can be used as a road surface texture layer over the landscape texture.

2. Fake urbanism

“Fake” urbanism, obviously, is a poorly detailed urbanism only meant as a distant illusion of a crowded cityscape. Follow this work flow to generate one (adapt details as needed):

1. In e.g. GIMP and using an orthographic view of the landscape as a background, draw a black/white city block bitmap layer following a street pattern layer (alternatively, the street pattern can be drawn afterwards. See below).
2. Export the bitmap and scale it up 10x.
3. Add additional features according to need.
4. Import the bitmap in Inkscape
5. Apply Filters/Blurs/Blur content
6. Apply Paths/Trace Bitmap with (e.g.): Brightness Cutoff = 0.500; Suppress Speckles: size 2; Smooth Corners: threshold 0.40; Optimize Paths: tolerance 0.20
7. Apply Paths/Simplify if needed (Beware: this rounds and shrinks the buildings)
8. Save as POV file (if you want to keep the svg file, save it as well)
9. Open in POV-Ray and change the object{} content as needed (e.g.): scale x an z by 1/10; scale up y; add textures; etc
10. translate to correct position, first by centering the object (use dimensions given in the file) then by trial-and-error until correct (use an orthographic vertical view) and apply the intersection with the “cutoff function” created from a height_field and additional functions. Because of the trial-and-error positioning it may be easier to draw the street pattern afterwards.

To generate the “cutoff function” , using the same height_field as the landscape, the following code can be used (with thanks to Christian Froeschlin):

```
#declare F_cells =
function {
  pigment {
    cells
    color_map {
```

© Thomas de Groot, 2012

```
      [0.1 rgb 0.4]
      [0.9 rgb 0.6]
    }
    scale 1/HF_scale.x //three scales compensating for prism object
    scale 20
    scale 0.1
    scale 0.5 //scale adapting to urbanism
  }
}

#declare cell_size = 0.002;

#declare F_flatroof =
function {
    F_topo (int(x/cell_size)*cell_size, int(y/cell_size)*cell_size, int(z/cell_size)*cell_size).hf
  + (F_cells(x,y,z).hf)*0.1
}

#declare hf_copy =
height_field {
  function MyRes, MyRes {F_flatroof(x,y,z)}
  smooth
  translate <-0.5, 0, -0.5>
  transform {MyTrans}
}
```

The “fake” urbanism is then generated by an intersection:

```
intersection {
  object {Urbanism //the prism object generated by Inkscape
    translate <-AllShapes_CENTER_X/10, 0, -AllShapes_CENTER_Y/10>
    translate <MyX, 0, MyZ>
  }
  object {hf_copy
    texture {T_var scale 0.1 scale 0.5}
    scale <1, 1, 1>
    translate -0.5*y
  }
}
```

3. Standalone urban scenes

Standalone urban scenes can be rendered separately from the master scene as long as little or nothing is visible from the landscape. Before building the scene, it is good to have at least some notion about the underlying topography of the scene (see also paragraph 4).

To be able to embed the city scene (CityScene.pov) into the master scene (MasterScene.pov) which contains the landscape height_field, an important switch is necessary in both.

At the beginning of CityScene.pov, just after the #version declaration, add:

```
#ifndef (Standalone)  #declare Standalone = on;  #end
```

In MasterScene.pov, this will correspond to the following lines, anywhere in the scene where you want CityScene.pov to be rendered:

© Thomas de Groot, 2012

```
#declare Standalone = off;
#include “CityScene.pov”
```

Now, in CityScene.pov you can put within an #if statement every scene element that should only be switched on when the scene is rendered independently, such as global_settings{}, camera{}, sky_sphere{}, light{}, etc. Example:

```
#if (standalone)
  global_settings {...}
  camera {...}
  light{...}
#end
```

Any other element not included in the switch will be rendered whether the switch is on or off. However, as some transformation is probably necessary when rendered in MasterScene.pov, declare a union{} of all those elements in CityScene.pov. Example:

```
#declare MyStreet =
union {
  object {MyHouses}
  object {MyFigures}
  object {MyStreetFurniture}
  etc
}

#if (Standalone)
  object {MyStreet}
#end
```

In MasterScene.pov, the street scene can now be rendered with the following code:

```
#declare Standalone = off;
#include “CityScene.pov”

#declare Norm = <0,0,0>;
#declare Street_pos = trace (MyHeightField, <MyX, 1000, MyZ>, -y, Norm);

object {MyStreet
  scale MyScale
  rotate y*MyAngle
  translate Street_pos
}
```

This is the basic method. Additional switches can be defined according to need and/or the complexity of the street scene.

4. Urbanism on an uneven topography

To position urbanism on an uneven topography, it may be worthwhile to use a high-resolution ground mesh during modelling and construction of the urbanism. This mesh can be used directly in POV-Ray of course if CSG objects are used, although you probably would work directly with the height_field in that case, but it is more particularly useful when using modellers like Moray, Wings3D, SiloPro, or Blender.

© Thomas de Groot, 2012

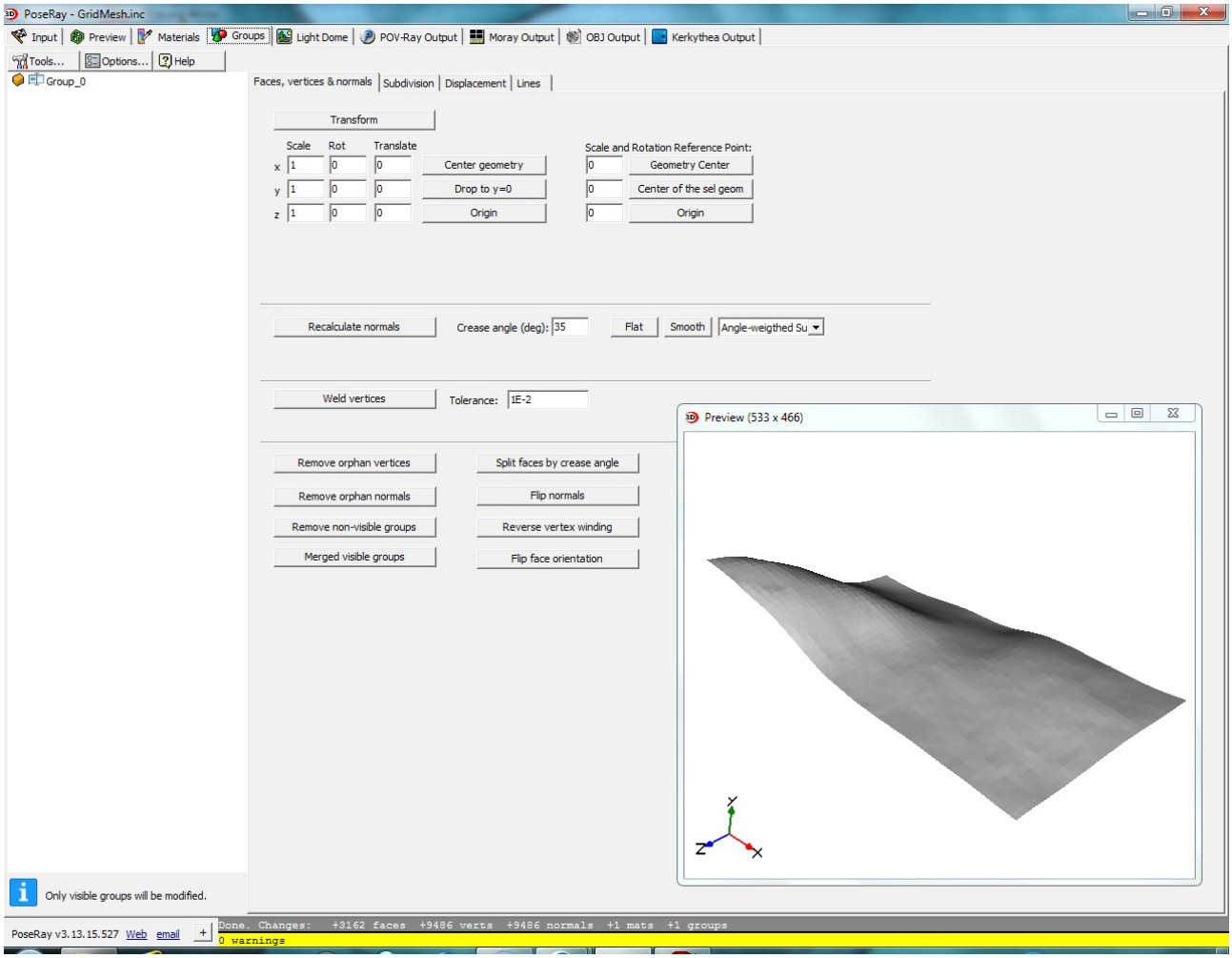
As height_fields cannot be exported directly as meshes (and they would be too large for our purpose anyway) these have to be built separately from corresponding sections of the height_field. To do this, the following little macro can be used:

```
#macro MeshGen(Surf,MinX,MinZ,MaxX,MaxZ,Incr)

  #debug "   writing grid mesh...\n"
  #fopen GridMesh "GridMesh.inc" write
  #write (GridMesh,"mesh {\n")

  #local Z = MinZ;
  #while (Z <= MaxZ)  //start of Z loop

    #local X = MinX;
    #while (X <= MaxX)  //start of X loop
      //triangle A:
      #local Nor = <0, 0, 0>;
      #local Loc = <X, 1000, Z>;
      #local Pos = trace (Surf, Loc, -y, Nor);
      #write (GridMesh,"  triangle {<",vstr(3,Pos,", ",0,6),">, ")
      #local Z = Z + Incr;
      #local Nor = <0, 0, 0>;
      #local Loc = <X, 1000, Z>;
      #local Pos = trace (Surf, Loc, -y, Nor);
      #write (GridMesh,"<",vstr(3,Pos,", ",0,6),">, ")
      #local X = X + Incr;
      #local Nor = <0, 0, 0>;
      #local Loc = <X, 1000, Z>;
      #local Pos = trace (Surf, Loc, -y, Nor);
      #write (GridMesh,"<",vstr(3,Pos,", ",0,6),">}\n")
      //triangle B:
      #write (GridMesh,"  triangle {<",vstr(3,Pos,", ",0,6),">, ")
      #local Z = Z - Incr;
      #local Nor = <0, 0, 0>;
      #local Loc = <X, 1000, Z>;
      #local Pos = trace (Surf, Loc, -y, Nor);
      #write (GridMesh,"<",vstr(3,Pos,", ",0,6),">, ")
      #local X = X - Incr;
      #local Nor = <0, 0, 0>;
      #local Loc = <X, 1000, Z>;
      #local Pos = trace (Surf, Loc, -y, Nor);
      #write (GridMesh,"<",vstr(3,Pos,", ",0,6),">}\n")
      #local X = X + Incr;    #end  //of X loop

    #local Z = Z + Incr;
  #end  //of Z loop

  #write (GridMesh,"}\n")
  #fclose GridMesh

#end  //of MeshGen
```

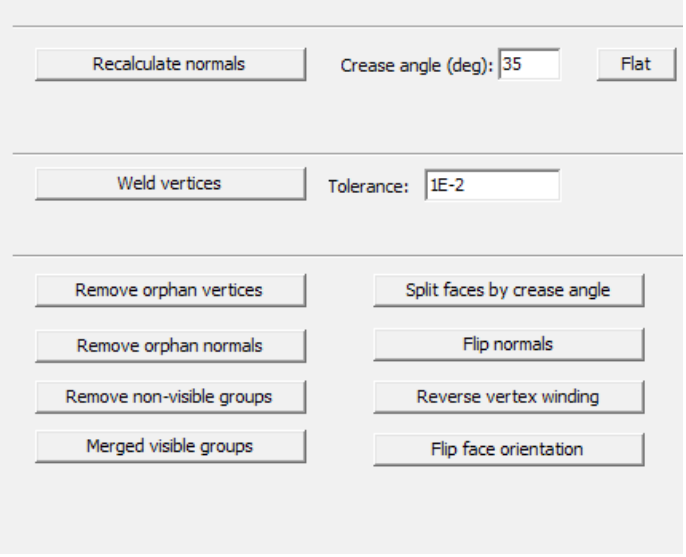
The macro generates an include file containing a mesh{} object. Input parameters are “Surf” (the height_field{} object); “MinX”, MinZ”, “MaxX”, and “MaxZ” (the minimum and maximum coordinates of the mesh on the height_field); “Incr” (the incremental value determining the resolution of the mesh: lower values mean higher resolution).

© Thomas de Groot, 2012

The mesh file can be imported into Poseray and prepared for export:

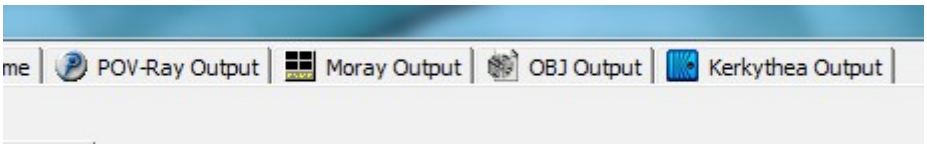


In the Groups tab of Poseray, the only necessary action is to apply “Weld Vertices” as the individual triangles in the original mesh{} are not bound together:



© Thomas de Groot, 2012

You have now the option to export to POV-Ray, Moray, OBJ, or Kerkythea, depending on the modeller you want to use, and import the object there for further work:



© Thomas de Groot, 2012