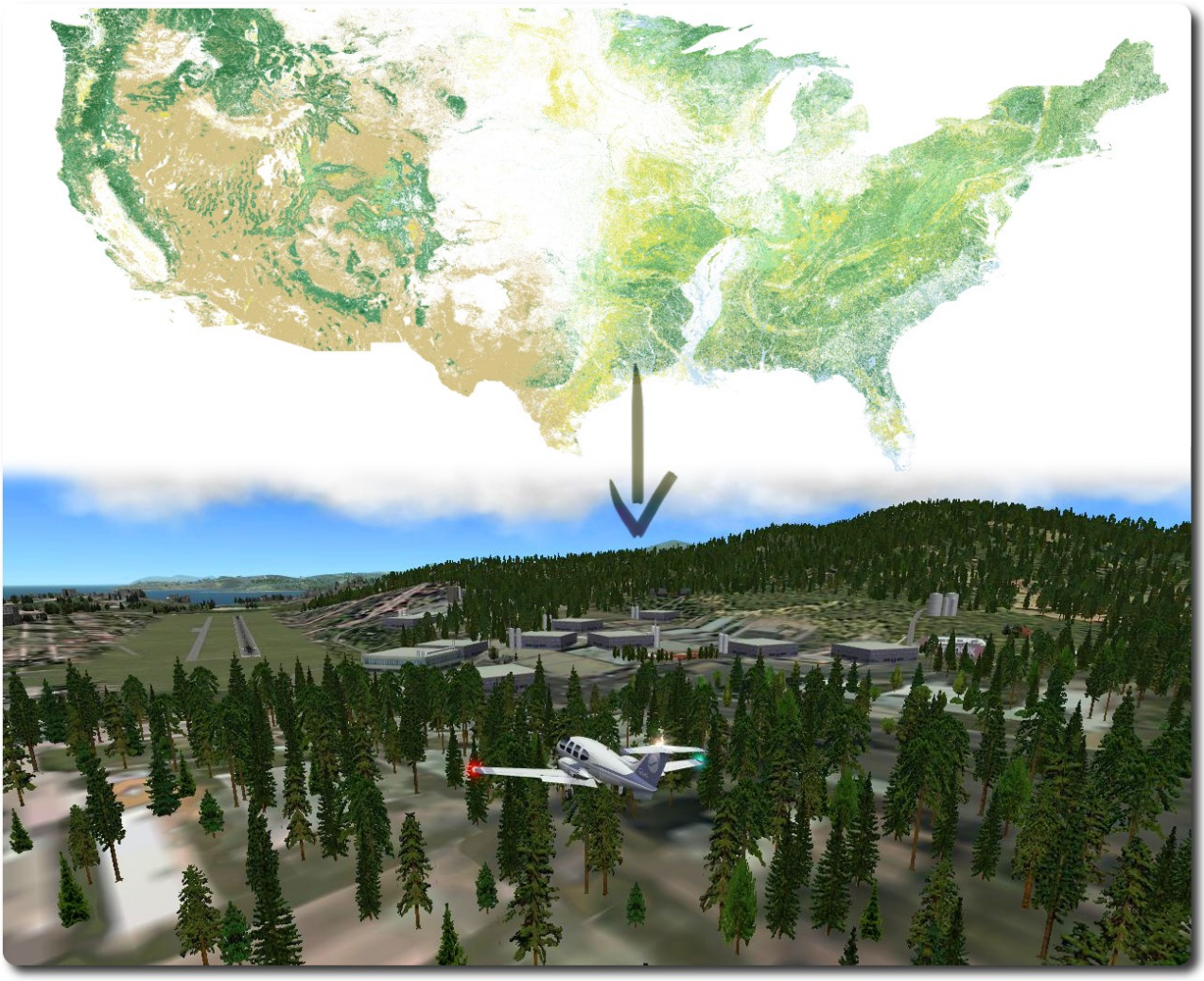
Vectorization of NLCD raster data and forest scenery creation for X-Plane in the USA

by Andras Fabian / www.alpilotx.de
November 2007



Contents

1 Overview....3
2 Data Sources....4
2.1 NLCD 2001....4
2.2 Bayley's Ecoregions....5
2.3 Vector data from X-Plane....6
3 Creating the scenery....8
3.1 Vectorization....8
3.1.1 Preparing the raw data....8
3.1.2 Vectorizing the raster data....10
3.2 Scenery Transformation....19
3.2.1 Extracting line features, airport areas and objects from global scenery....19
3.2.2 Selecting needed ecoregions....21
3.2.3 The core scenery extraction process....22
3.2.3.1 The landclass loop....22
3.2.3.2 Reading the GML file....22
3.2.3.3 The ecoregions loop....22
3.2.3.4 Selecting forest polygons in ecoregions....22
3.2.3.5 Selecting line features, objects and airports crossing forests....24
3.2.3.6 Buffering the lines....24
3.2.3.7 Cutting out line features and airports from the forests....25
3.2.3.8 Exporting to the DSF format....27
4 Visual artwork creation....31
5 List of illustrations....32

Sidenote:

As of November 2007, the techniques described in this document also – mostly – apply to the creation of the European forests. They differ in the raw data (CORINE vs. NLCD) used, and how it is prepared for later processing (for example, CORINE data doesn't need a treatment like the vectorization described in 3.1, as that data already comes in vector format). There is also – of course – also a difference in the used ecoregions (Bayleys vs. Biogeographical Regions, Europe 2005).

1 Overview

This document should help to understand the complex process of generating forest overlays for the flight simulator X-Plane 8.x. It covers the whole process of preparing the raw data, through the generation of scenery files and finally some aspects of the artwork. It also gives some good examples on how to use the free GIS package GRASS.

illustration 1: before and after adding forests to X-Plane

2 Data Sources

There are many differences in how the scenery for the USA was created – in comparison to Europe. One of the biggest such differences is, that unlike for Europe, where the European Environment Agency (EEA) already provides ready made vectorized data in form of the Corine land cover 2000 vector by country (CLC2000), in the USA I had to work with raster data. Though, there exists vectorized data for the USA in form of the LULC dataset, but some research showed, that this data is not the best when it comes to details, and is very old (originates from the time around 1970-1980).

2.1 NLCD 2001

The National Land Cover Database 2001 (NLCD 2001) showed up just in the right time, when I started work on this project and it brought some very good aspects like:

- data is up-to-date
- has a very high resolution of 30m/pixel
- a companion dataset with canopy density is available

Of course it also has/had some disadvantages for the project like:

- it is raster data instead of vectorized polygon data
- some nice layers – know from the European project – like *fruit trees* or *olive groves* are "missing".

The following illustrations are two examples of the NLCD 2001 landcover and canopy data.

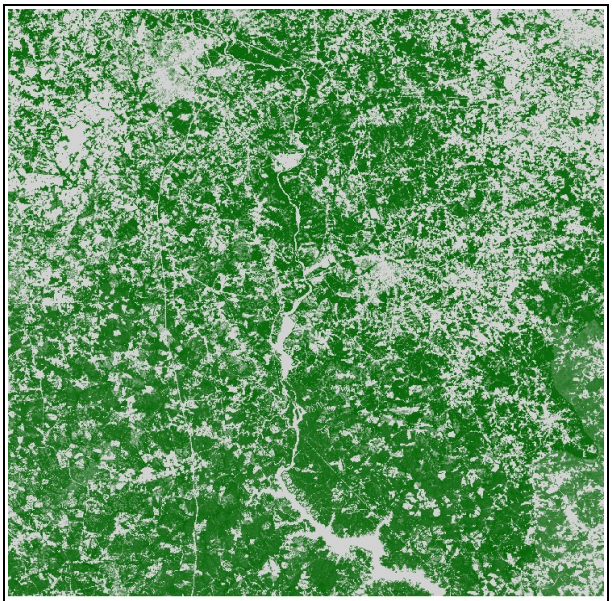


illustration 2: canopy example

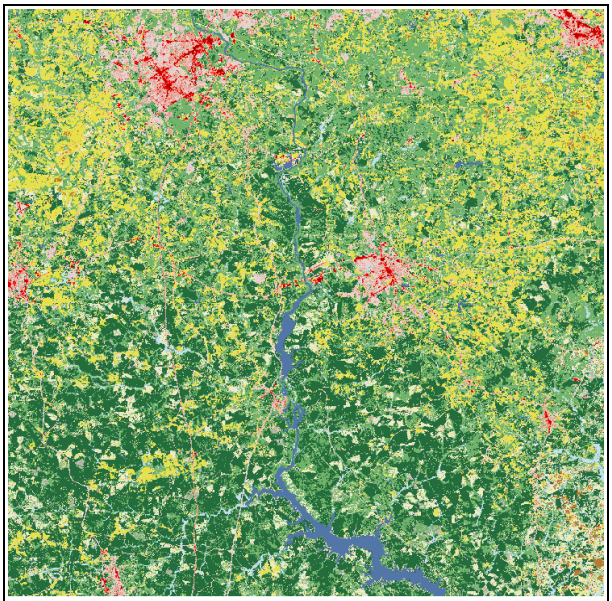


illustration 3: landcover example

The NLCD 2001 dataset has 6 layers which are relevant for the creation of a forest scenery. Those layers are:

- deciduous *(NLCD Layer: 41)*
- evergreen *(NLCD Layer: 42)*
- mixed *(NLCD Layer: 43)*

- shrub *(NLCD Layer: 52)*
- pasture *(NLCD Layer: 81)*
- woody wetland *(NLCD Layer: 90)*

You can read more about the NLCD 2001 classes in the documentation: NLCD 2001 Land Cover Class Definitions

2.2 Bayley's Ecoregions

The 6 layers in the NLCD 2001 landcover data are not enough on their own, to have a differentiated rendering of the forests. You can imagine, that making an evergreen forest in Washington state look like one in souther California wouldn't be too realistic. For this reason – exactly like in the Europe project – I looked for an additional dataset to better differentiate the forests for all the ecoregions. It turned out, that there are many different such datasets for the USA which all differ in their detail and the number of ecoregions / climatic zones they show. I had too keep in mind, that in the end when it comes to define the look of the forests, I will have to deal with a combination (well, essentially a crossproduct) of all – or most – forest types and ecoregions. So I had to find the right balance (in expected versatility of the look of the final scenery and the artistic work needed to achieve it). This lead me to the Bailey's Ecoregions and Subregions of the United States, Puerto Rico, and the U.S. Virgin Islands dataset. This dataset also features 3 levels of detail, and some experiments showed, that it should be sufficient to go with Level 2 the Ecosystem Divisions.

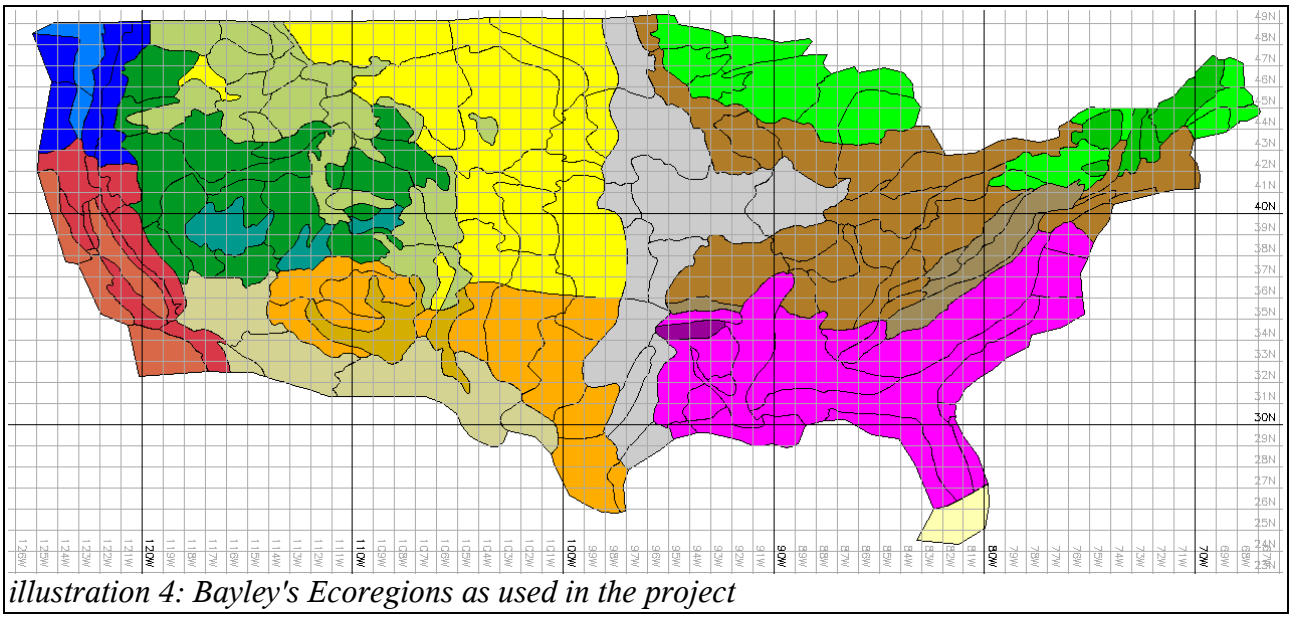


illustration 4: Bayley's Ecoregions as used in the project

The original Bayley's Ecoregions dataset uses a fairly detailed coastline, which is nice if used on it's own, but can become a problem, when trying to combine it with the other NLCD 2001 data. Namely, that at the coastlines they sometime do not perfectly overlap. And as the scenery creation process matches the forest polygons against the ecoregions polygons, it can happen, that it cuts off some forest polygons at the coastlines which are not covered by ecoregions polygons. So some hand “preprocessing” was needed, in which the coastlines were generalized and extended out. This also has the nice side effect, that the number of vertices was vastly reduced in the ecoregions data, and as such this improved the processing speed later during the scenery creation.

Finally a “short” overview of the ecoregions used:

- 210 Warm Continental Division

- M210 Warm Continental Division - Mountain Provinces
- 220 Hot Continental Division
- M220 Hot Continental Division - Mountain Provinces
- 230 Subtropical Division
- M230 Subtropical Division - Mountain Provinces
- 240 Marine Division
- M240 Marine Division - Mountain Provinces
- 250 Prairie Division
- 260 Mediterranean Division
- M260 Mediterranean Division - Mountain Provinces
- 310 Tropical/Subtropical Steppe Division
- M310 Tropical/Subtropical Steppe Division - Mountain Provinces
- 320 Tropical/Subtropical Desert Division
- 330 Temperate Steppe Division
- 340 Temperate Desert Division
- M340 Temperate Desert Division - Mountain Provinces
- 410 Savanna Division
- M410 Savanna Division - Mountain Provinces
- 420 Rain Forest Division (only Hawaii)

2.3 Vector data from X-Plane

This is also new to the USA forest generation process. In comparison, the European forests relied on the external data from VMAP0 (for roads, railroads and city boundaries) and DAFIF (for airports). This datasets worked fairly well for Europe, but they also showed some weaknesses. Namely, that DAFIF didn't know about all airports of the X-Plane Global Scenery and thus, some smaller airfields weren't cleared from trees (they were overgrown with a forest or it's parts). Them same applied – although only in rare situation – to roads. Now some research and discussion with one of the X-Plane developers (Ben Supnik) showed, that going this way – in the USA with TIGER/Lines data, which the Global Scenery uses too – wouldn't be a good idea. On the one side because of the reasons listed above (lesson learned from Europe) and on the other, because it became clear that processing the TIGER/Line data would be a fairly complicated (and an error prone) task.

This lead to the new idea, to “not reinvent the wheel”. I mean, all the data which X-Plane renders out of the Global Scenery is already available there. I only needed to “re-engineer” it and make it available to GRASS. Luckily, the specification of the scenery is open and well documented at the X-Plane 8 Scenery Central. There is also a tool available – DSF2Text – which not only let's compile text files to scenery, but also the other way around text out of scenery. Using this tool, and a lengthy GAWK script finally made it possible, to extract all the line features and airport grass areas (the polygons beneath airports) from the scenery, and make them available to GRASS in the GML format. The processing also made possible to somewhat classify all the line features and give them different widths (or even leave out the thinnest ones).

Below is an example, that shows such an extracted line network and airport polygons.

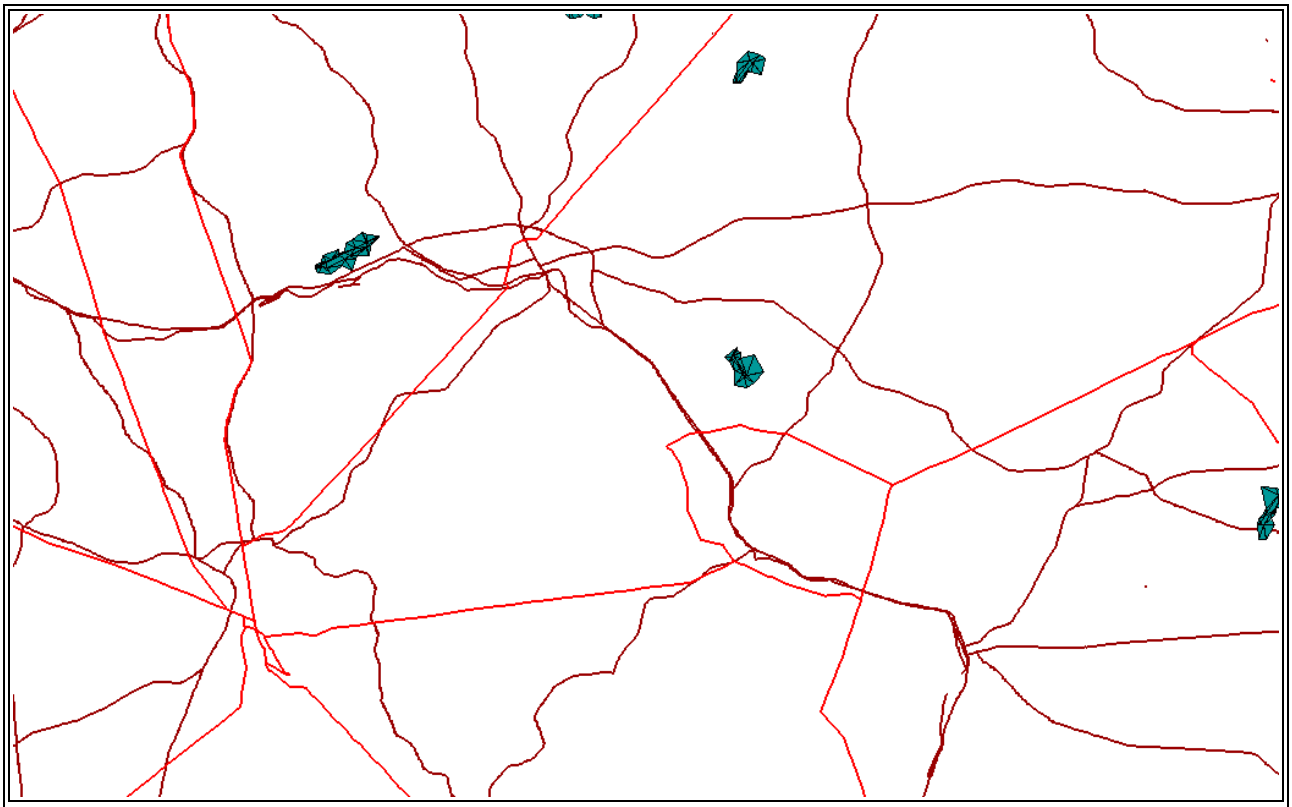
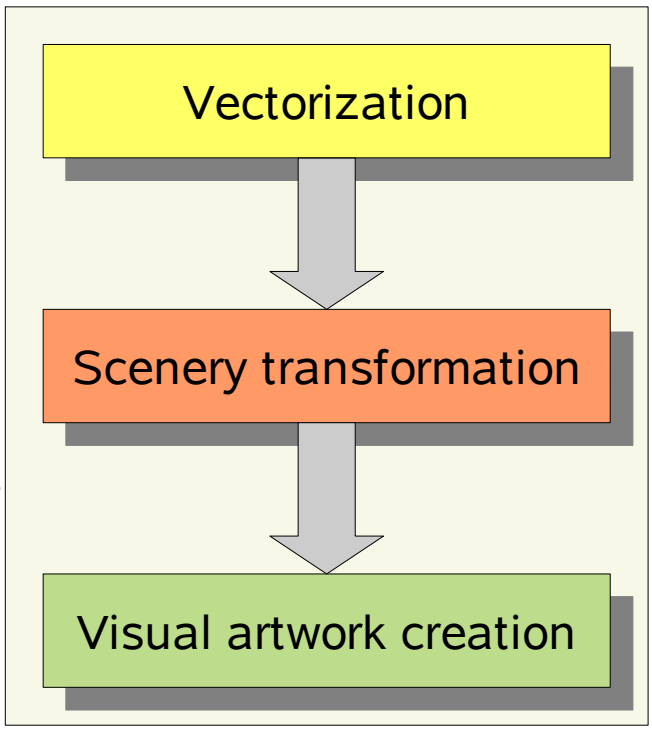


illustration 5: example for extracted line features and airport polygons

3 Creating the scenery



The scenery creation process is comprised of three main tasks. They are the building blocks and each of them represents a more or less closed task.

1. Vectorization
2. Scenery transformation
3. Visual artwork creation

3.1 Vectorization

Depending on the data source which is available, the amount of work to prepare the data for the scenery transformation can vary largely. A good example for a slightly easier approach is the European Forest project. There I could use ready made polygons from Corine land cover 2000 vector by country (CLC2000), which offloaded the vectorization process to the European Environment Agency (EEA).

In the USA everything was a “little” different. Although, there exists a fantastic (because of it's detail) data source in form of the National Land Cover Database 2001 (NLCD 2001) dataset it introduced a new problem. The NLCD 2001 data is only available in raster format, and thus it somehow has to be transformed to polygonal data. This is the part, which I will describe here as the **Vectorization** process.

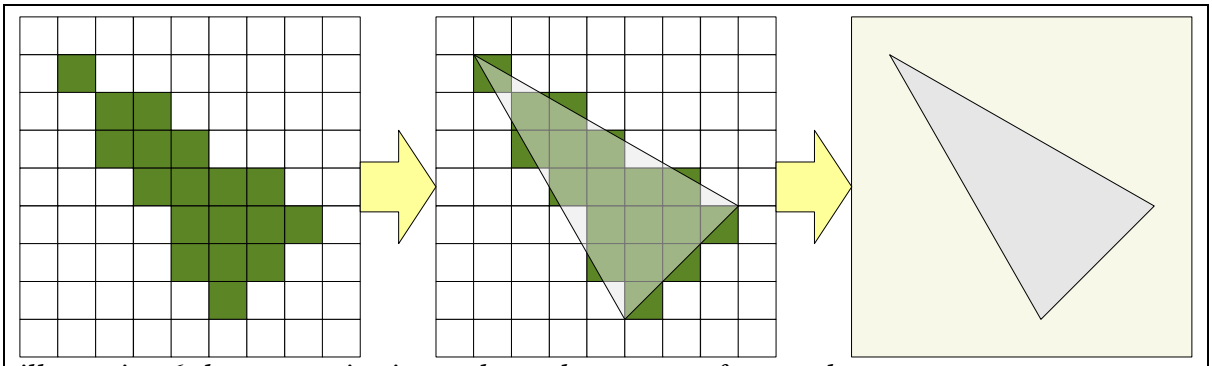


illustration 6: how vectorization makes polygons out of raster data

3.1.1 Preparing the raw data

Acquiring the data is fairly simple, when using the National Land Cover Database 2001 (NLCD 2001) website. Although there is a link to the MRLC Interactive Viewer, I rather recommend to use their FTP download. This allows to directly download the large zone files one-by-one (without Alaska, this are 14 zones). In every zone, you can choose what to download, but for the purposes of the project, “only” each *Tree Canopy zip files* and the *Land Cover zip files* are needed. This zipped data (for the 14 continental USA zones) will already take up 3.9 Gbytes of disk space! The unzipped data needs between 20 and 30 Gbytes (I can't remember the exact number).

The next step is, to bring the raster datasets in the needed projection. The default projection of the NLCD 2001 datasets is:

```
Map_Projection_Name: Albers Conical Equal Area
Albers_Conical_Equal_Area:
Standard_Parallel: 29.500000
Standard_Parallel: 45.500000
Longitude_of_Central_Meridian: -96.000000
Latitude_of_Projection_Origin: 23.000000
False_Easting: 0.000000
False_Northing: 0.000000
```

But X-Plane works in the much simpler **WGS84 Geographical Coordinate System**. Luckily there is a nice command line tool named *gdalwarp* included with the GDAL package, which can take care of this task. As an example, such a transformation (and also format conversion to GeoTIFF) looks like this:

```
gdalwarp -tps -t_srs '+proj=longlat +ellps=WGS84 +datum=WGS84
+no_defs' -s_srs '+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=23.0
+lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +datum=NAD83 +units=m
+no_defs' -srcnodata 255 -dstnodata 255 -tr 0 0
/data/Maps/NLCD2001/area_10_landcover/landcover10_011007_0fix.i
mg /data2/Projects/X-Plane_nlcd/raw/landcover10.tif
```

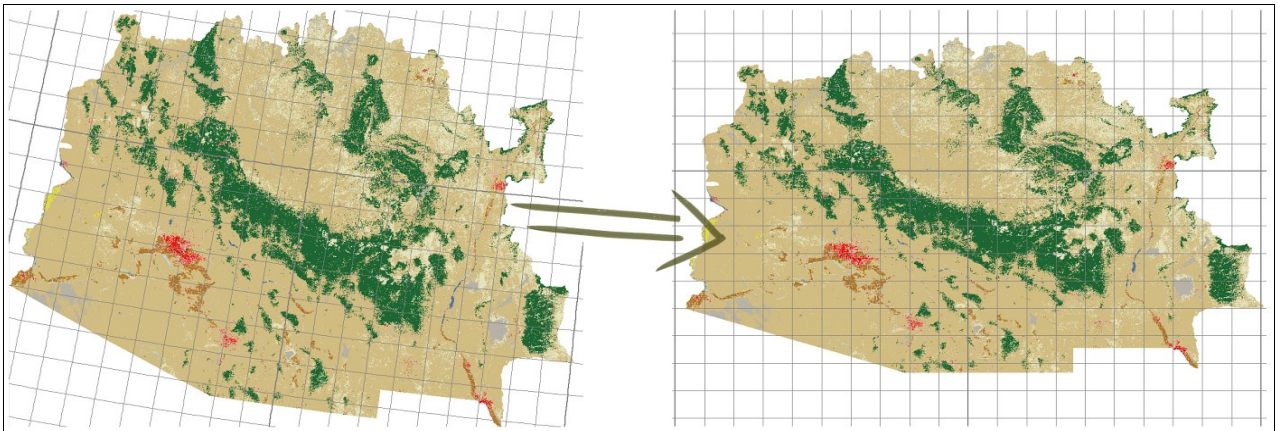


illustration 7: (real) example for the coordinate transformation

After the transformation, the data can be loaded to GRASS gis, for further processing. This is again a fairly simple step and is best done (looped over all 14 landcover and canopy files!):

```
r.in.gdal -o input=/rawdir/canopy1.tif output=canopy1
```

This of course again takes a lot of time and a lot of disk space. Now the last step in the preparation is, to put all the 14 zones (for landcover and canopy) together to one big dataset. Tis is very important, because this way, we avoid problems along the borders of the zones. Later, when we extract 1x1 degree tiles, those don't align with the borders of the zones but rather overlap them. So when we have a 1x1 degree tile, that is at the border of two adjacent zones, we would have difficulties to get the data from both of them. But if they are already patched together, we will not need to care about this anymore.

Before we can use the r.patch utility from GRASS, we will have to do some recoding of the raster datasets which we have just imported. The reason is, that the “no data” areas in NLCD 2001 are coded as the value 127. But the *r.patch* utility wouldn't see this as transparency, and would overwrite real data values. So we need to recode 127 to 0 (which *r.patch* can see as transparency). Again, here is an example (which has to be looped over all 14 zones):

```
g.region rast=canopy1
r.recode in=canopy1 out=canopyC1 < /projhome/raw-recode-
rules.txt
r.colors map=canopyC1 rast=canopy1
g.remove rast=canopy1
```

The *raw-recode-rules.txt* file contains only two simple lines:

```
0:126:0:126
127:127:0
```

Now we are really ready to patch all the zones together. This can be done with the following code lines:

```
MAPS=`g.mlist type=rast sep=, pat="canopyC*"`
g.region rast=$MAPS
r.patch -z in=$MAPS out=canopy –overwrite
```

So in the end we should only have (the rest can be deleted to recover disk space) the two big raster layers ***canopy*** and ***landcover.***

3.1.2 Vectorizing the raster data

In this part I will describe all the needed steps, to get – finally – some polygons out of the raster data. An important point is, that all the processing in the following steps is based on **1 x 1 degree tiles**. The main reason is, that the scenery system of X-Plane is working this way, but it also makes the whole processing easier and each single step much faster (*Divide and conquer* should come here to mind). So, in the beginning we are starting with the two gigantic datasets ***canopy*** and ***landcover***, which we prepared in the previous step. These are our pools of raw data, where we pick out a **1 x 1 degree tile** in each iteration.

Setting the region for our work is very straight forward in GRASS with g.region :

```
g.region w=$x1 s=$y1 e=$x2 n=$y2 res=0.00052164
```

There is also an important little variable showing up, namely the *res* (it stands for resolution). It can help us to change the resolution of the data in the following steps. The value in the above example is about 1.5-times of the original resolution – in map units (GRASS works with units) – of the raw NLCD2001 data (which has a resolution of *0.00034776*). So the next processing step will already, lower the resolution a bit (but which is still extremely detailed).

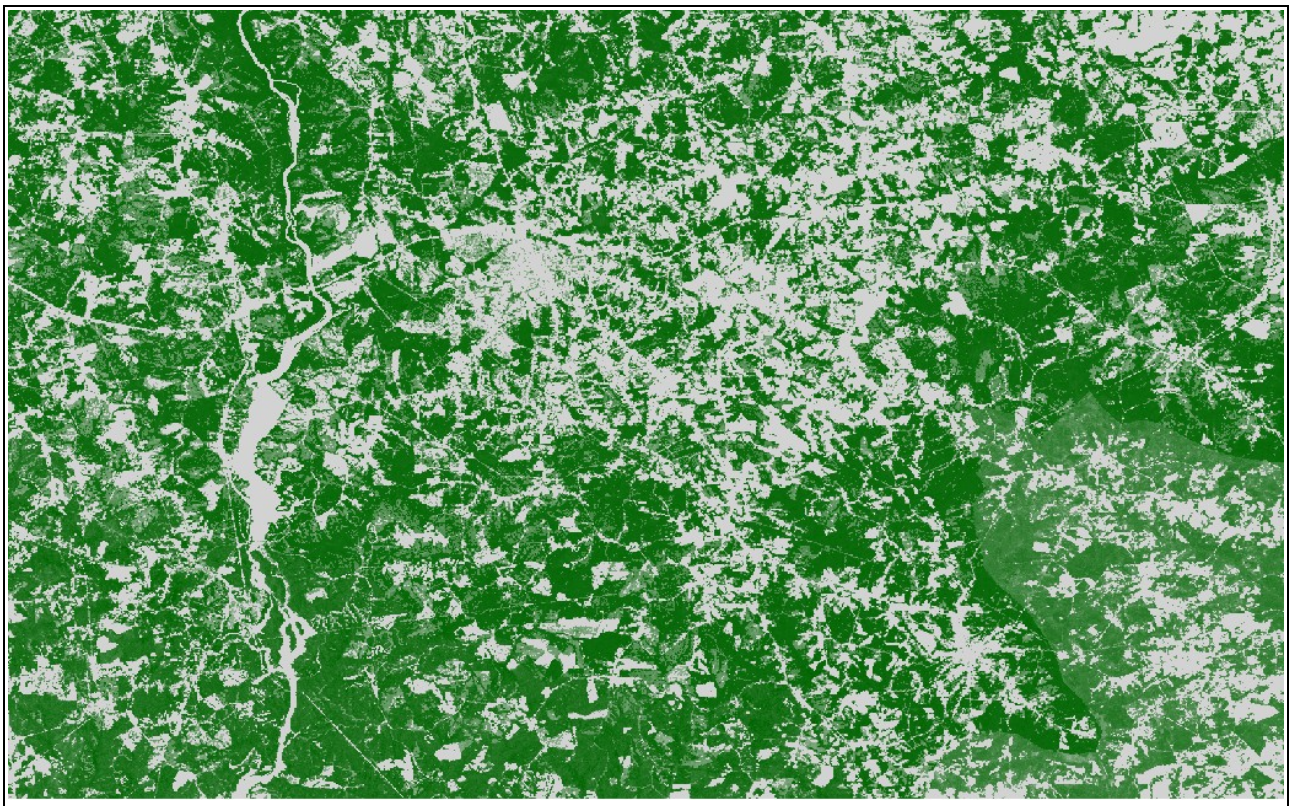


*illustration 8: our example **canopy** area*

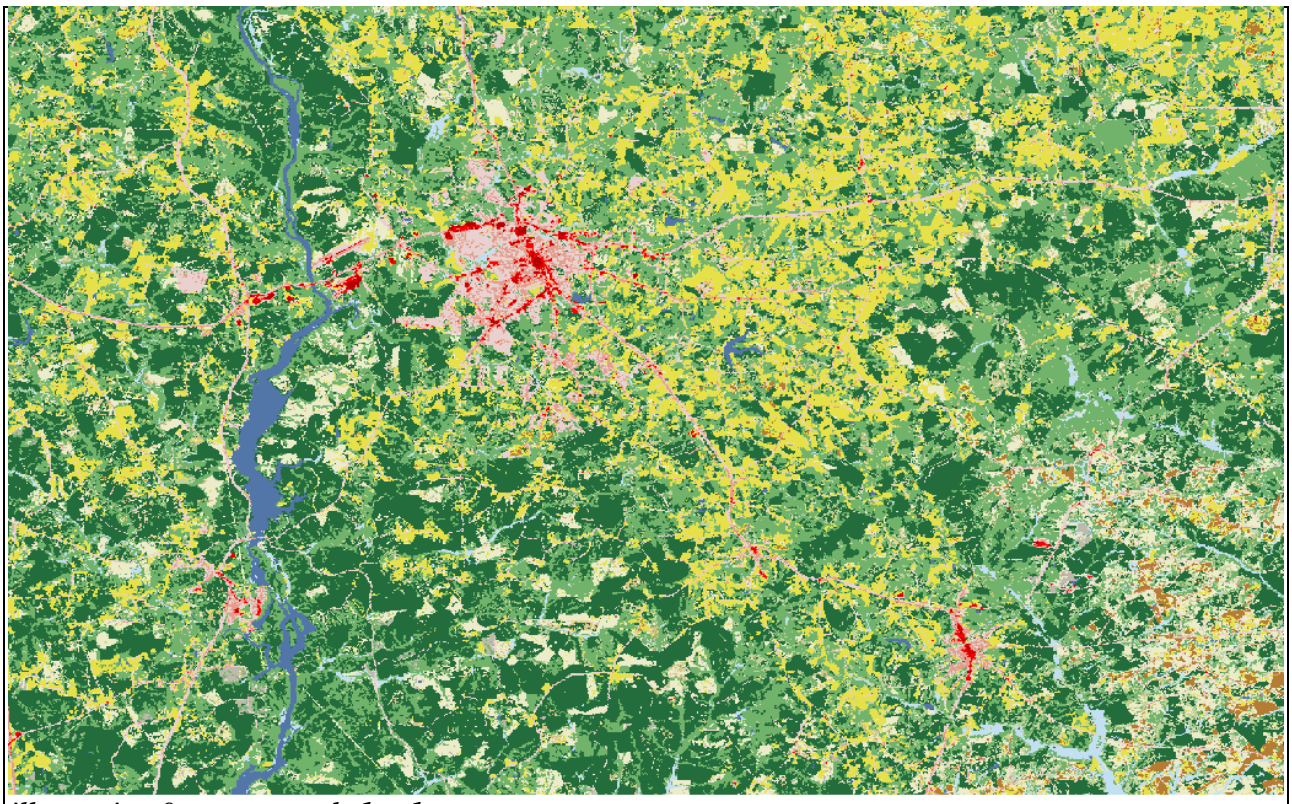


*illustration 9: our example **landcover** are*

Above you can see two example for a **canopy** and a **landcover** area (both are from the same region, if I remember correctly it is around *Charlotte,NC*) which I will work – well with one exception later on – throughout the documentation.

At first we have all the 20 layers of the NLCD2001 data in the **landcover** dataset, and the **canopy** dataset with 100 very fine steps. But as described earlier (in chapter **2.1**) we only need 6 layers from the **landcover** dataset:

- deciduous *(NLCD Layer: 41)*
- evergreen *(NLCD Layer: 42)*
- mixed *(NLCD Layer: 43)*
- shrub *(NLCD Layer: 52)*
- pasture *(NLCD Layer: 81)*
- woody wetland *(NLCD Layer: 90)*

And from the **canopy** layer we can't work with all those fine steps, but we need to discretize them and then combine them with the *deciduous, evergreen, mixed* layers, so we get from each of them a **_sparse** and **_dense** version (as you can see from the two variations, this already suggests, that I only discretized those fine steps to two categories). So we would like to end up with 9 different layers, namely:

- deciduous_dense *(our layer: 41)*
- deciduous_sparse *(our layer: 1)*
- evergreen_dense *(our layer: 42)*
- evergreen_sparse *(our layer: 2)*
- mixed_dense *(our layer: 43)*
- mixed_sparse *(our layer: 3)*
- shrub *(our layer: 52)*
- pasture *(our layer: 81)*
- woody wetland *(our layer: 90)*

Now, how do we get the new reclassified data? At this point, one of the fascinating features of GRASS come to our help. It is the so called r.mapcalc function which lets do a large variety of arithmetic operations on raster layer (and of course combinations of them). In the end, it is one – although a little lengthy – call, which can do all the reclassification for us:

```
r.mapcalc "landcover_reclass=if( ((landcover==41 ||
landcover==42 || landcover==43) && canopy>=85) || landcover==52
|| landcover==81 || landcover==90,landcover,
( if((landcover==41 || landcover==42 || landcover==43) &&
canopy<85, landcover-40,0)) )"
```

What we do is, to only pick out the needed layers, and also – for *deciduous, evergreen, mixed* – decide with a threshold value (85 in the example – and also in the real project – which was a result of some experiments), whether the layer is *sparse* or *dense*. The new dataset **landcover_reclass** then already holds everything we need for the rest of our work.

illustration 10: the landcover dataset after the reclassification

Although – as described at the beginning of this chapter – we have already lowered the resolution of our working set, it is still too detailed and speckled for the vectorization. It would result in too many and too detailed polygons, which then in the end would be hard to handle by X-Plane (or even impossible). So, in the next step, we apply some filters, which do "average" out the fine details. This filters (and their variables) were picked out ofter many experiments, and give a good balance between detail and amount of polygons at the end of the vectorization process.

Those filters are ***mode*** and ***median***, and were applied through r.neighbors:

```
r.neighbors input=landcover_reclass output=landcover_proc1
method=mode size=5 --overwrite

r.neighbors input=landcover_proc1 output=landcover_finished
method=median size=3 --overwrite
```

The size parameter sets the number of neighboring pixels, which are used in the calculation of each filter. As told above, the values were picked after numerous experiments (and they can greatly influence/modify the outcome of the filters!).

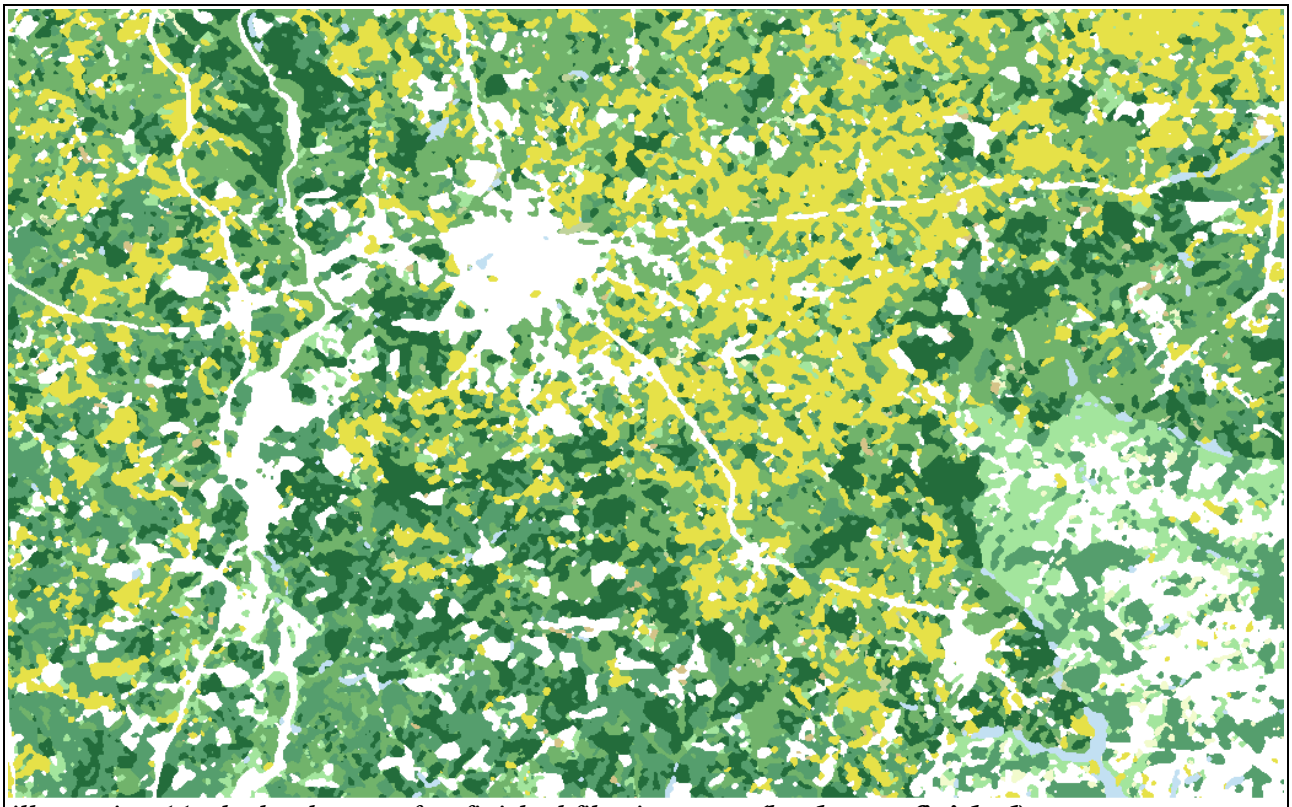


illustration 11: the landcover after finished filtering steps ***(landcover_finished)***

After this preparation steps, we have arrived at the point where the vectorization itself can be done. Of course, not even this step is as easy as it sounds.

The program I used for this is called **KVEC**, which is shareware and does a very good job, when it comes to vectorization (although I suspect it was originally not intended for this kind of use ... who knows). But after many experiments it also turned out, that the vectorization doesn't work very well on the composite layer, but rather on separate layers.

This means, that we need to loop over all of our 9 layers, separate them out of **landcover_finished** and then pass them to KVEC for the vectorization.

This is achieved – just like with the reclassification – by using the r.mapcalc command.

```
r.mapcalc
    "landcover_shrub=if
    (landcover_finished==
    52,1,0)"
```

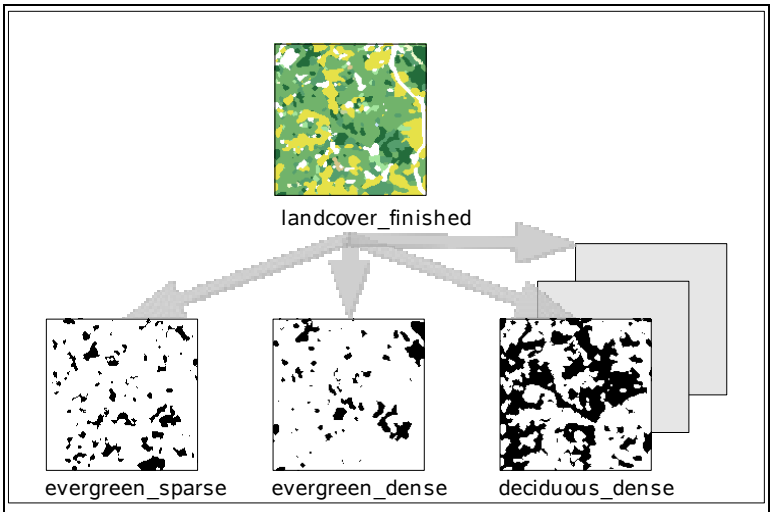


illustration 12: How the 9 layers are split up

The resulting "temporary" layers are then exported as *tiff* files, so that KVEC can finally begin processing them.

Using KVEC is a little bit tricky, as it has so many parameters which can be used, to tweak it to our needs. So, some of my time also went into experiments, to find out which settings bring the best

results (well, they are maybe not the best, but turned out to be sufficiently good enough for this project). As an example, I will show here the parameter set, I used for producing the final scenery:

```
-coord pixel
-fill solid
-format svg
-grit 6
-monitor
-reduce all
-sort local
-resolution high
-vblack
-tcolor color 255 255 255
-maxpoints 20000
-winding optimized

(the meaning of all the settings can be looked up in the
documentation of KVEC)
```

The result of this conversion is – as the *format* clause suggests – in SVG format. This can already be viewed in most vector editing softwares (like Inkscape) but can't be directly imported into GRASS. The good part is – and this was the reason why I choosed SVG as an intermediate format – that SVG is an XML Graphics format. As such it can easily be manipulated (even without DOM fiddling and using only some basic text tools) and converted to GML (Geography Markup Language). And GML can readily be imported by GRASS.

This SVG-to-GML conversion was achieved by a small little self written script which is mostly based on a bunch of perl calls.

Finally, the freshly vectorized data can be read into GRASS. Though, there still remains a problem, namely the coordinate space in which our vector data resides. KVEC's shortcoming is, that it loses the knowledge about geographic data. It interpreters our raster data as an image, which has – for example – coordinates X: 0..1200 and Y: 0..1200 but no reference to a geographic location. As a result it also returns vector data in the image coordinate space (basically but not exactly – nevertheless the returned “coordinate space” can easily be determined). And finally our vector data comes exactly that way into GRASS. Or put in an easier way: when our raster image had before a lower left coordinate of +28N -90W the vectorized data has a lower left coordinate of “0N” “0W” (and so on). But as we still are have a cartesian coordinate system at hand, the vector data can easily be translated back to our normal coordinates. Basically, we only need to offset the coordinates back to their right origin, and rescale them. Even better, this is a built in feature of GRASS, so we only need to parametrize it's command v.transform to get the correct result.

```
v.transform input=landcover_classname
output="landcover_classname_tr" points=transform.points –
overwrite
```

The file ***transform.points*** is dinamically generated to contain the correct offset values, and could look like this:

```
# Linear transformation from XY to WGS84 coordinates:
# 4 maps corners defined
# UL NW
# UR NE
# LR SW
# LL SE
```

```
0 612  -81 34
612 612  -80 34
612 0 -80 35
0 0 -81 35
```

The illustration below should give a rough overview, what is happening in this step of – implicit and explicit – coordinate transformations.

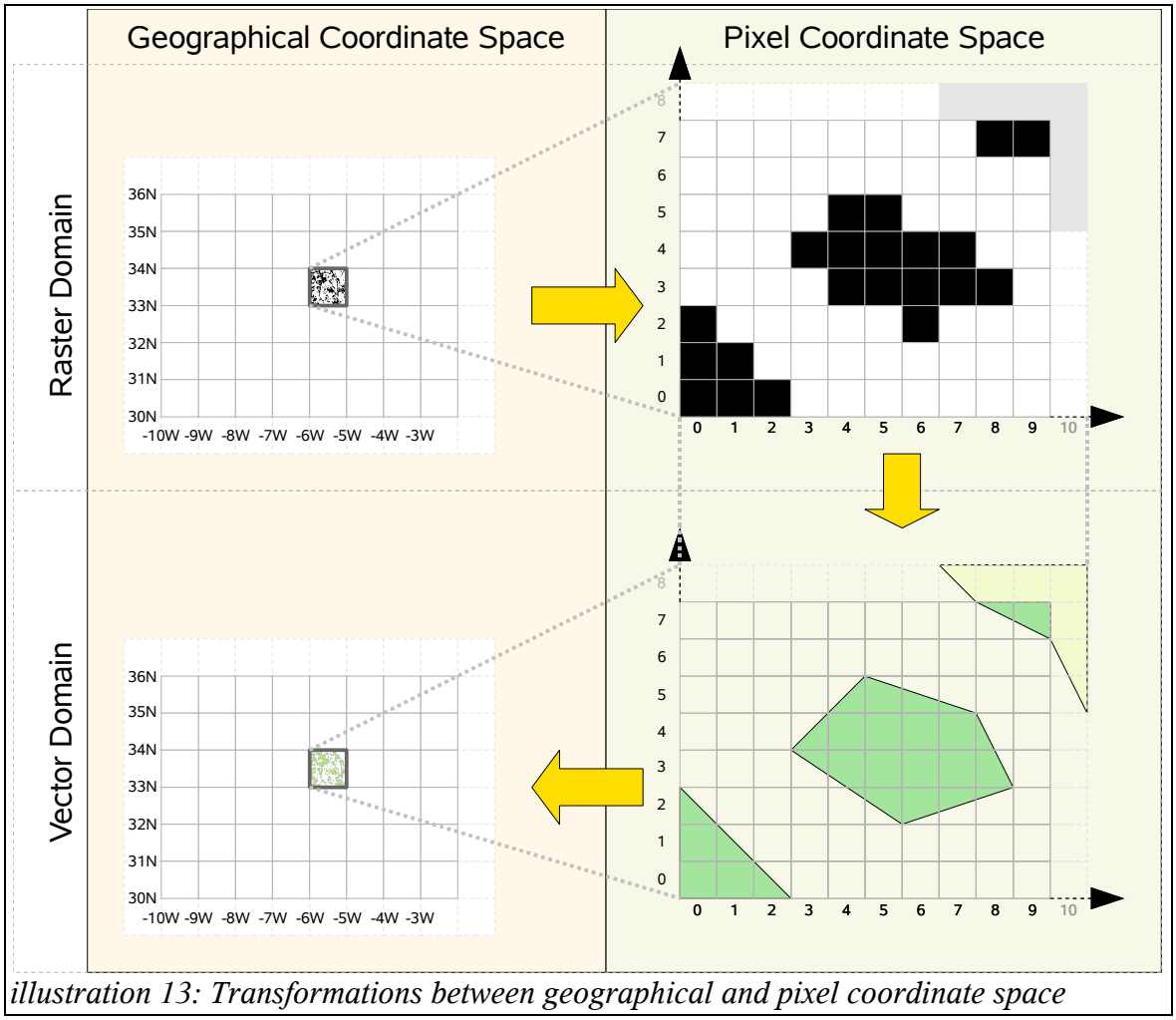


illustration 13: Transformations between geographical and pixel coordinate space

With this, we have a correctly referenced vector data set in GRASS, which already fulfills most of our needs. Only one step remains at this point, which is needed because of some restrictions with X-Plane. It is the *"no more than 255 windings in one polygon"* limitation. Windings just mean islands or holes in one big polygon. And my vectorization process just turned out to return some very big polygons, which – often – had far more than 255 windings (especially in the dense and large forests of the north west).

There are two choices, to deal with this situation. Either we throw away some windings (but which ones, and how to decide?) or we reduce the size of the polygon by dividing them in smaller parts. I decided to go the latter way, because it is far more easier to implement and I don't lose windings. Easily put, I have constructed a regular grid – built from rectangular polygons – which I used to cut up all my polygons, by using **v.overlay ... operator=and ...** The cutting grid had a size of 10x10 rectangles (this turned out to be a good value after some experiments – and it handled all the winding issues in the USA project!).

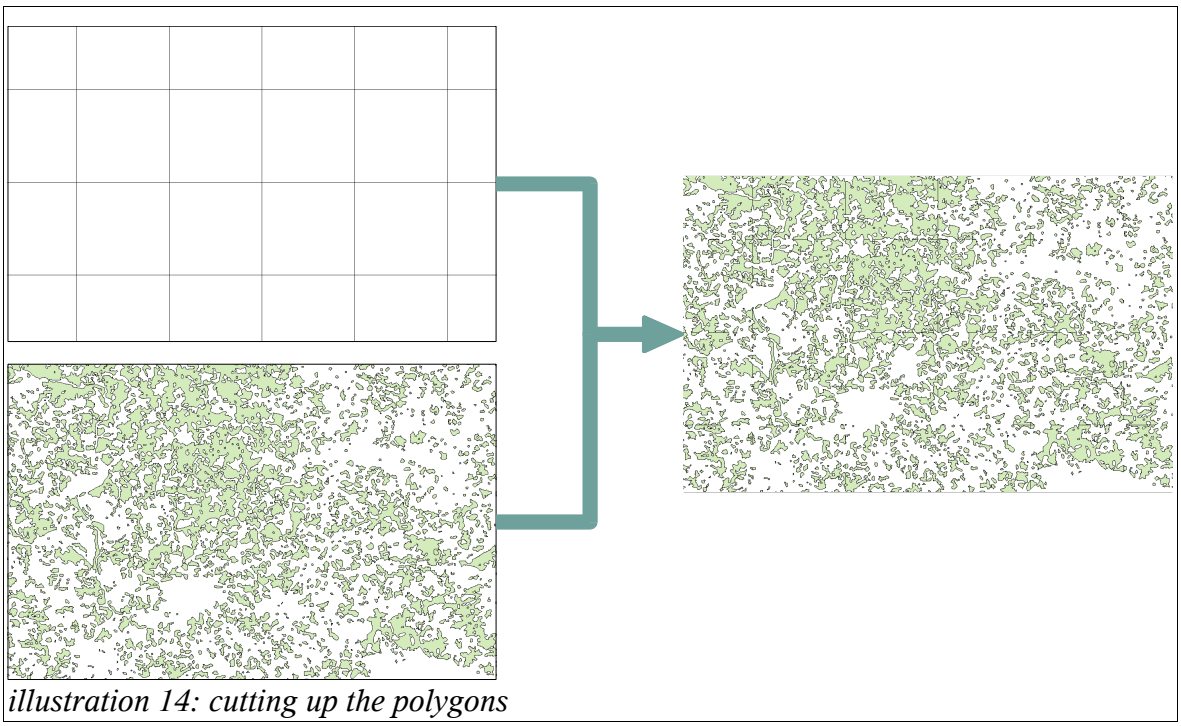


illustration 14: cutting up the polygons

After all this steps, the polygons are reexported in the GML format, and after all forest types in the working area (the 1x1 degree tile) have been vectorized, transformed, cut and exported they are zipped together in a package and put in a directory for the later use in the **scenery transformation** step.

On the next page, *illustration 15* gives a graphical overview of the whole vectorization process, as described on the previous pages.

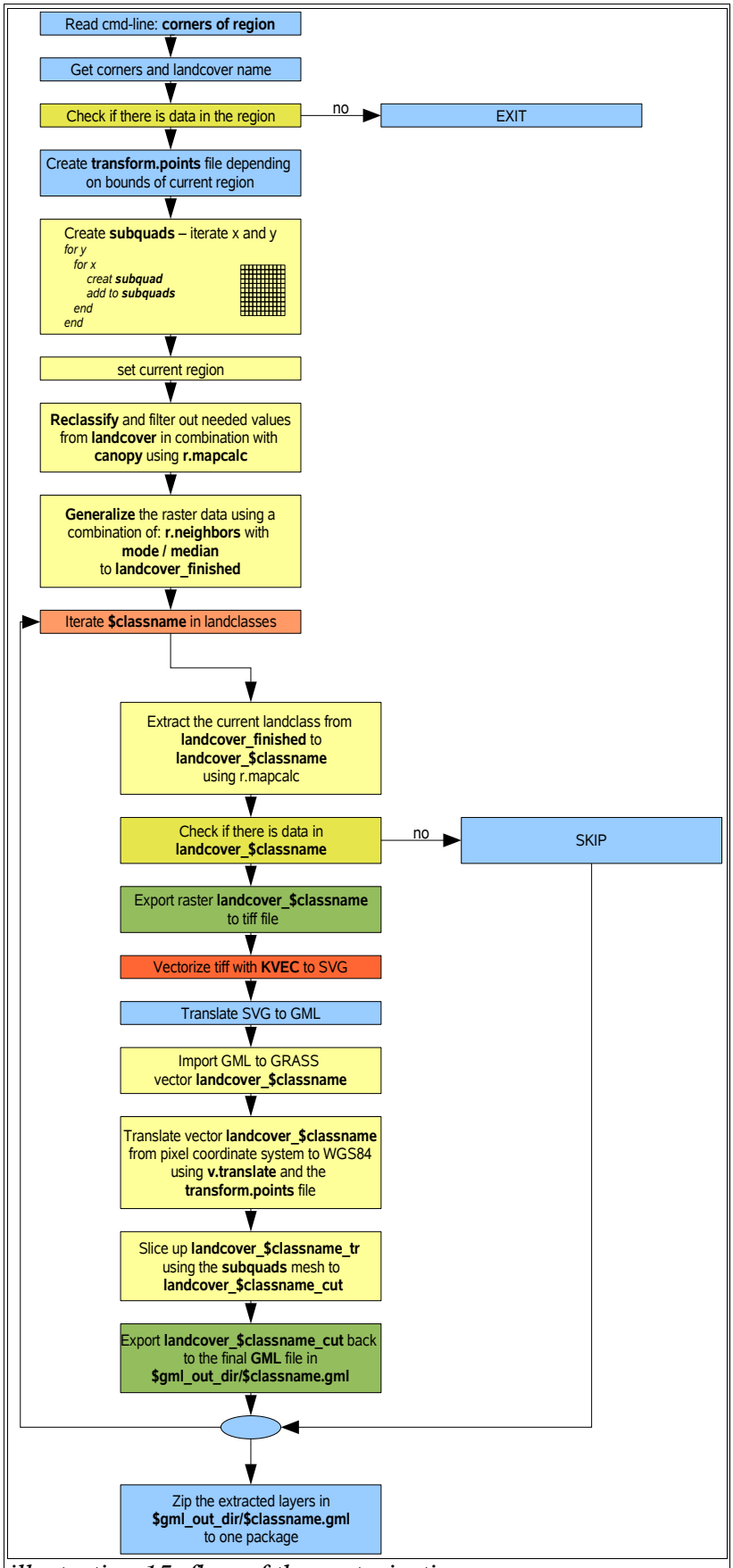


illustration 15: flow of the vectorization process

3.2 Scenery Transformation

The second big step in forest scenery creation is the transformation of all the extracted vector data into the DSF scenery format. In this step, all the 1x1 degree vector data packages will be used, which were generated in the previous vectorization process. This also defines the size of our working area throughout the whole process: still, always **1x1 degree.**

Before the main loop of scenery generation can begin, there are two prerequisite tasks, which have to be completed each time.

3.2.1 Extracting line features, airport areas and objects from global scenery

Although all the forest polygons – which we extracted in the vectorization step – are fairly detailed and accurate, they still – vary often – don't represent very fine features like roads, railroads and power lines (cut out). They also often disagree with the exact area of the airports used in X-Plane, and if not taken in consideration, we would often end up with trees on runways or somewhere else at airports (where we definitely don't want to see them). Finally, there are also a lot of buildings represented in the base scenery (Global Scenery) which the forests don't know about (and trees would often "grow" through the buildings) So, we need accurate data about this features, so we can cut them out from our forest polygons at processing time.

This is one of the new developments in the USA forest scenery. In earlier renderings of the European forests I have used data from external sources (like VMAP0 and DAFIF) which were fairly well suited for the task, but as it later turned out, not always agreed with what X-Plane used in the Global Scenery. So for the USA I "invented" a new process, which uses the data directly from the Global Scenery. This is very straight forward, and will guarantee that the cut out areas in the forests align precisely with what X-Plane renders.

So the most important task was, to find a way to revers engineer the DSF files, and gain access to the needed data. Luckily, the DSF scenery format is open and so it can be decompiled. With the existing definitions it was also possible to extract the needed features.

The first, and simplest step is to "decompile" the DSF files into the human and script readable format TXT. This can be accomplished with the little tool DSF2Text.

The more complex part was, to extract the features from it. To achive this, I decided – after some research – to us the C like, stream oriented scripting language GAWK. It allowed me to write a single pass parser (***extract-dsf.awk***), which had to go through the DSF TXT file only once, and could already write out the road / railroad / power line vector lines and airport area polygons to GML format!

Roads (and other line features) are comprised of line segments, which are defined through *BEGINE_SEGMENT / END_SEGMENT* pairs in DSF. From this, the line features can be reconstructed. For all the line features, I also implemented a very primitive classification, so I could sort all the line features in 4 categories (adding it as the *roadwidth* in the GML file) depending on their width. As we will see, this gives a bit of – much needed – flexibility when it comes to the cutting process later on. Recognizing the line width is not too complicated. In the line segment definitions of the DSF format, every line segment begins with a

```
BEGIN_SEGMENT <type> <subtype> <node id> <longitude> <latitude>
<elevation>

example:
BEGIN_SEGMENT 0 47 65293 -83.649628 34.606718 411.196759
```

The *<type>* - as its name suggests – tells the simulator, what type of line it should draw. This type definitions are in an extra – so called – "Road Network definition" (files ending with *.net*).In this definitions one can find the width of the lines:

```
ROAD_TYPE <subtype>  <width> <length> <texture>  <red> <green>
<blue>

example (from X-Plane 8.60/Resources/default scenery/820 roads/
road.net):

ROAD_TYPE 47   6.000000 15.000000 1   1.0 1.0 1.0
-------------------^
the width is: 6m
```

Based on the definitions from the *road.net* file, I could build in the classification in my GAWK script for all the line types.

Objects represent all the buildings in the scenery. In the scenery file they are represented by points (referring to the center of the objects), the orientation of the object and a numeric (index) reference to an *OBJECT_DEF* in the header of the scenery (this *OBJECT_DEFs* are then the "pointers" to the definition of the object). The referred definition files (like "/lib/global8/us/park_250_120a.obj") are unique and can be used to resolve the size of the objects (well essentially their radius, as that is more convenient for buffering – radius = sqrt((width / 2)^2 + (depth / 2)^2)). With this information, the extractor script can output a simple *objects.ascii* list which holds 3 values for each object: *lat, long, radius*

After the object extraction the script also immediately creates the buffered object polygons (with the *radius* and a scaling factor to map units – essentially something like a conversion between meters and degrees). Because of some "issues" with GRASS, there are two sets of objects created (the ones which succeed in the first pass, and then a second set containing the rest - or at least most of it). That's why after this step we have to work with **objects_buffered1** and **objects_buffered2**.

Airports differ from roads and objects in that they are already polygons and they are defined as such in the DSF files (enclosed in *BEGIN_PRIMITIVE* / *END_PRIMITIVE* pairs). Finding them is not very hard, because they belong to a specific set of terrain definitions (*TERRAINF_DEF's* are always at the beginning of DSF files, and predefine how different terrain features should be rendered). The code has to search for all *TERRAIN_DEF*'s which begin with the string "/lib/g8/terrain/apt*", and then extract the corresponding polygons (as, these are the airport areas we are looking for). One important aspect of airport polygons is – this applies to most terrain polygons – that there are two kinds of them. The ones which depict the real terrain area (in our case the airport footprint), and a set of outer polygons surrounding the inner ones, which are only used for texture overlapping (these are the overlay polygons). Of course we only need the former. They are marked by a specific flag in *BEGIN_PATCH,* where the 4. parameter equals **1** (this is described in the DSF File Specification).

At the end this script puts out the needed line features and airport area datasets. Imported in GRASS (and of course with the buffering of objects in GRASS), they can look – for example – like this:

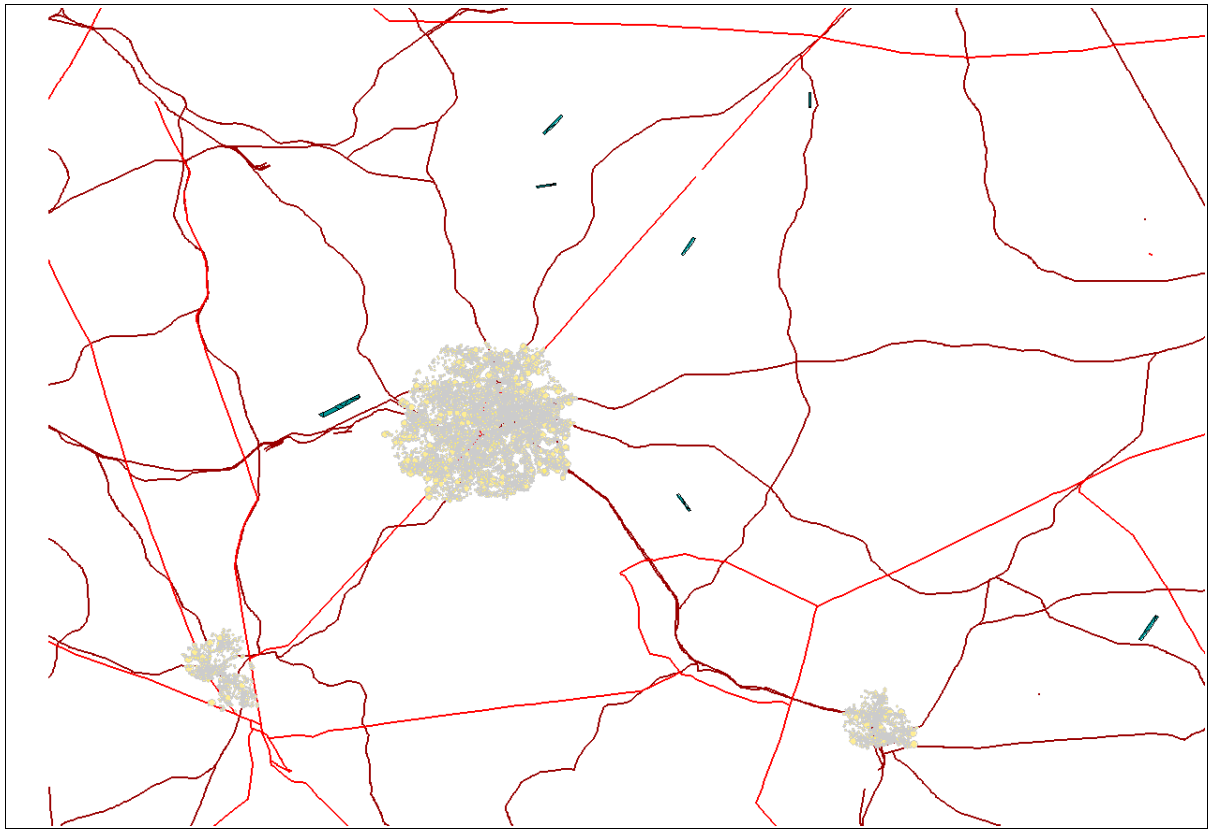


illustration 16: Example for extracted line features and airport areas

3.2.2 Selecting needed ecoregions

As the project knows about 20 different ecoregions, it is important to filter them at an early point, so the whole process has only to deal with those, which are covering our 1x1 degree working area. This is done with a loop over all known ecoregions, and selecting and cutting with the help from GRASS. In script it looks like the following:

```
for ecoregion in $(v.db.select -c map=ecoregions_current
column=ecoregion|sort|uniq)
do
        v.select -t ainput=eco_$ecoregion atype=area alayer=1
        binput=clipareaextended btype=area blayer=1
        output=eco_select operator=overlap –overwrite
        v.overlay -t ainput=eco_select atype=area alayer=1
        binput=clipareaextended btype=area blayer=1
        output="eco_clipped_"$ecoregion operator=and
        olayer=1,0,0 --overwrite

done
```

And to better understand, here is also a – real life – graphical example:

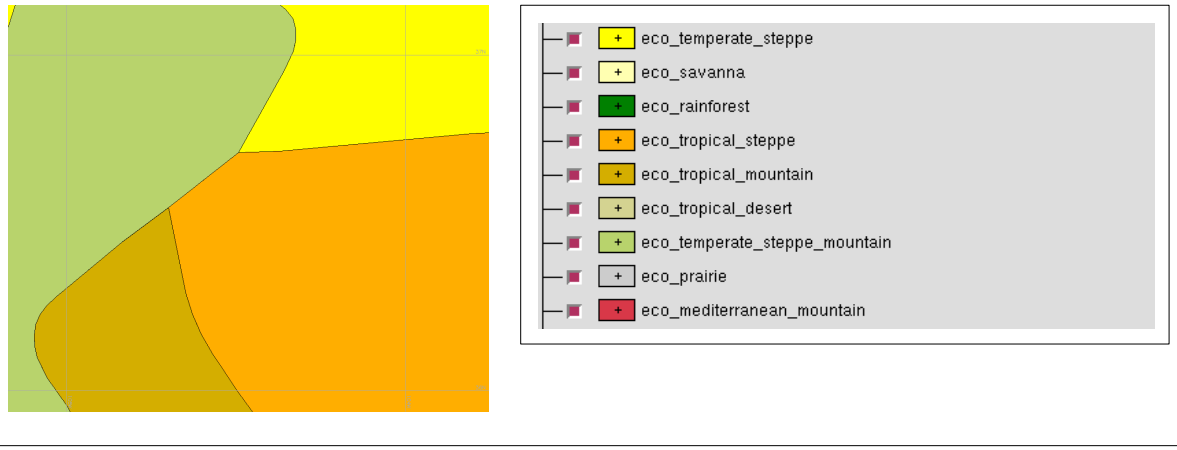


illustration 17: Example of selecting the ecoregions

3.2.3 The core scenery extraction process

This is the core part of the scenery extraction process, as it describes how I turn GIS based land class polygons (the forests) into the X-Plane scenery DSF tiles. I will try to follow a procedural approach in my description, so one can follow the "data path" the script is going down.

3.2.3.1 The landclass loop

Outside, there is a loop which iterates over all types of landclasses (types of forests) which are present in the working region of the 1x1 degree scenery tile. The landclasses are determined by the GML files included in the vectorized polygons zip file package (to remember, go back to the end of the vectorization process, where I describe, how all vectorized polygons – final GML files - are packaged into one ZIP file for later use – landclass by landclass). Their name is standardized, so that the process can use them directly as input.

3.2.3.2 Reading the GML file

Inside the loop, the process reads the GML files into a GRASS vector layer, so that the next steps can directly access this data

```
The import looks like:

v.in.ogr -t -o dsn=$wrk2dir/$classname.gml output=lc_$classname
min_area=0.0001 snap=-1 --overwrite
```

3.2.3.3 The ecoregions loop

In this inner loop, the process iterates over all ecoregions, which are present in the current 1x1 degree working area (remember, we have picked them already in "**Selecting needed ecoregions**" preprocessing step).

3.2.3.4 Selecting forest polygons in ecoregions

Now we have picked a landclass (outer loop) and an ecoregion (inner loop) and have now to build an intersection of them. The reason is, we would like to work only with the forests which belong to the current ecoregion (outlook: at the end, this enables us to assign different forest definitions to the

forests in X-Plane, depending on their ecoregion). The illustration 18 shows in a – real world – example, how this process should be thought of (but of course, at this place in the loop we always only work with one ecoregion, and not with all four at once as in the illustration) .

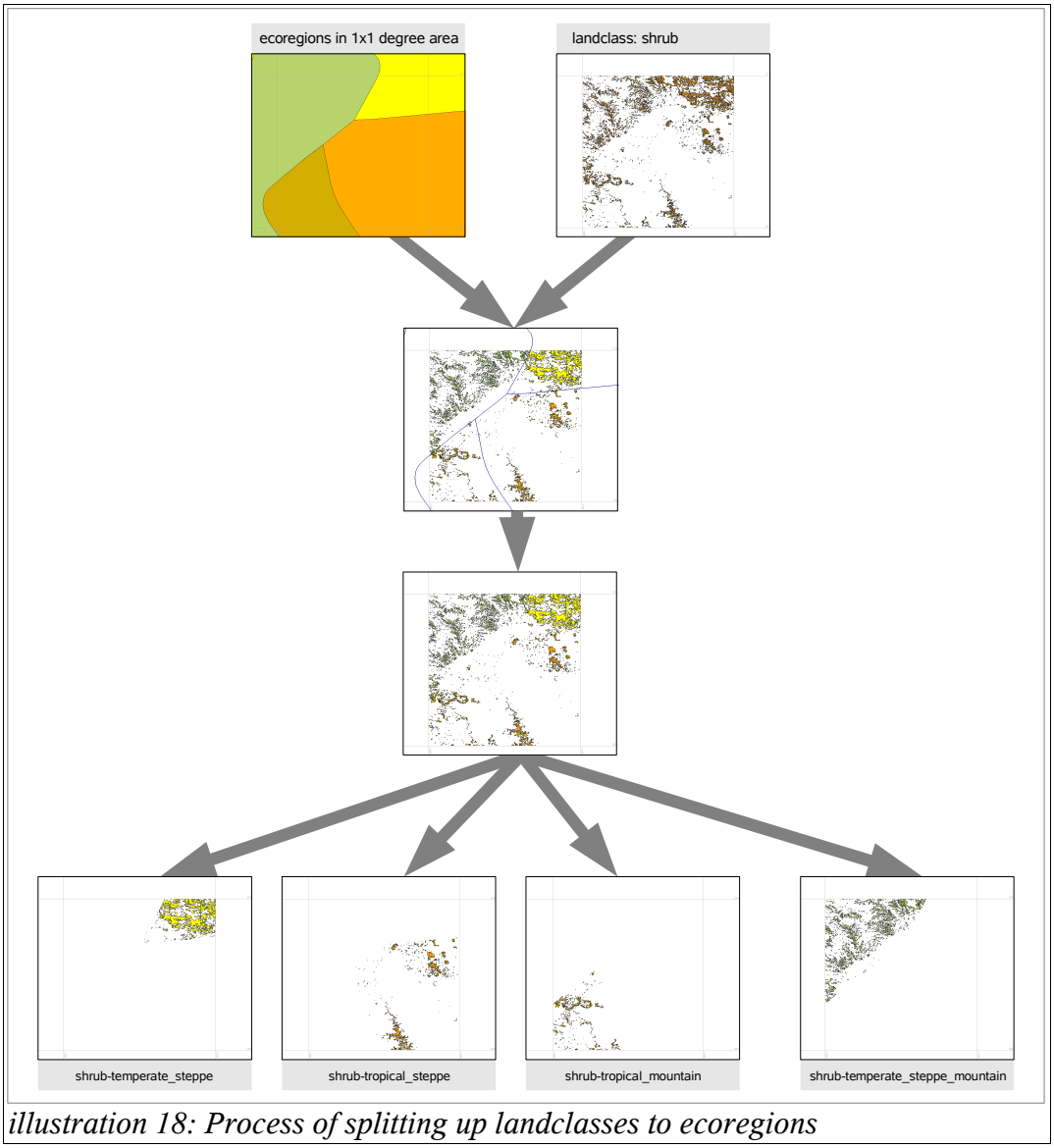


illustration 18: Process of splitting up landclasses to ecoregions

And again I would like to insert some example code, to help understand how I used GRASS for all this:

```
v.select -t ainput=lc_$classname atype=area alayer=1
binput="eco_clipped_"$ecoregion btype=area blayer=1
output=polygons_select operator=overlap --overwrite

v.overlay -t ainput=polygons_select atype=area alayer=1
binput="eco_clipped_"$ecoregion btype=area blayer=1
output=polygons_current operator=and olayer=1,0,0 –overwrite
```

As you can see, I used a two step process, by first v.select-ing and then cutting (using v.overlay with the **and** operator). This helps to speed up the process, as preselecting reduces he cardinality of the polygons which the cutting process has to take care of (I used this “trick” in many places of my script).

At this point we “could” already take a shortcut, and output the “selected” landclass_ecoregion polygons to a DSF file. But this wouldn't take into consideration, that line features, objects footprints (the default buildings in the scenery) and airports are not cut out. This will be done in the next steps.

3.2.3.5 Selecting line features, objects and airports crossing forests

It would be easy – from a pure algorithmic point of view – to just take all line features and airports from the preprocessing step, and cut them out from the forest polygons. But this – in many cases – would waste processing time, as often, many line segments don't even cross a forest (and as such, wouldn't cut out anything). For that reason I have built in a preselection step, very similar to the one where I reduce the forests to each ecoregion.

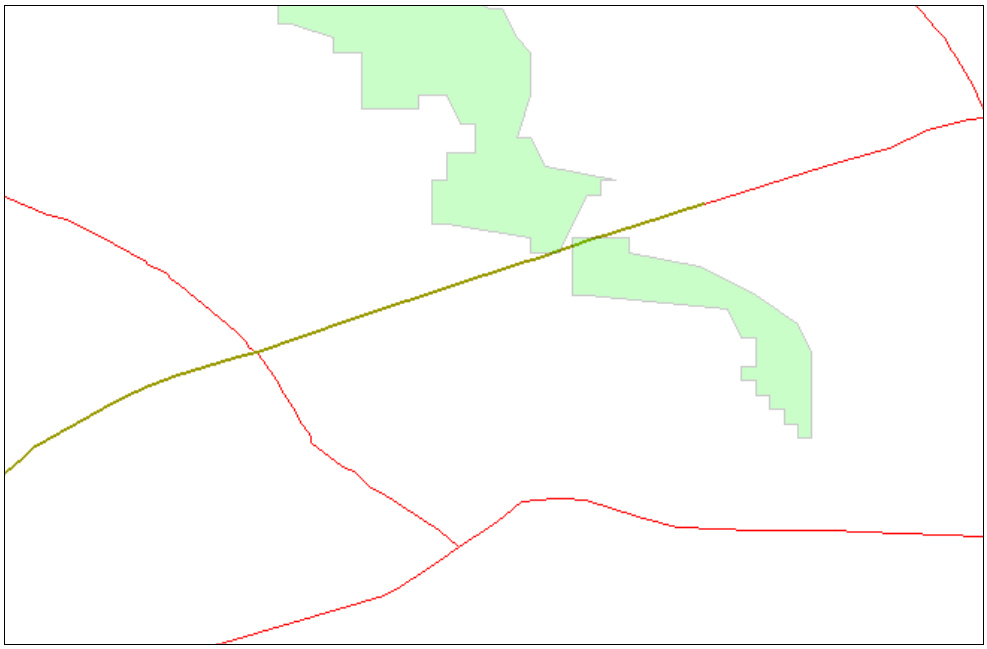


illustration 19: Selecting the line segments, which cross a forest polygon (dark yellow line)

This is done equally for all three line types – earlier in the line feature extraction description I wrote about classifying lines depending on their width – and also for airport areas and the objects (building footprints).

Such a selection would look like this, in GRASS speak:

```
v.select -t ainput=lines1 atype=line alayer=1
binput=polygons_current btype=area blayer=1
output=lines_current1 operator=overlap
```

3.2.3.6 Buffering the lines

Now before we can cut out line features from the forest polygons, the lines have to be transformed to polygons to have some “real” area. This process in GIS is called “buffering” – as it give a buffer to lines – and is well supported in GRASS though v.buffer:

```
v.buffer input=lines_current1_clean output=lines_buffered1
buffer=0.0002 type=line scale=1.0 tolerance=0.1 --overwrite
```

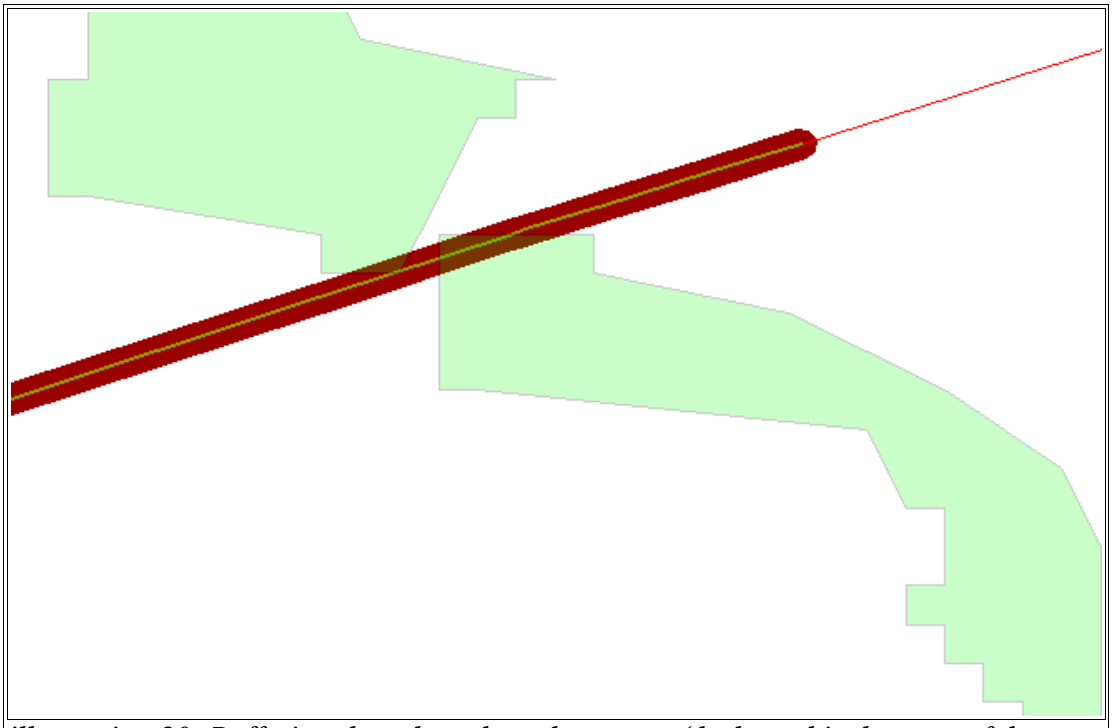


illustration 20: Buffering the selected road segment (dark read is the area of the new road polygon)

This is also the point where I use the information from the three different line classes. Depending on their numbering, I can give them different widths in the buffering process. This property is influenced by the *v.buffer* parameter called **buffer=0.0002** in the example above. And as you can see, **buffer** has a value which doesn't seem to directly correlate with a meter or other value. It is given in "map units" (in degrees), and I have derived the right value(s) with some experiments (testing in X-Plane, how well they come out).

Of course, airports don't need to go through this processing, as they are already areas.

3.2.3.7 Cutting out line features and airports from the forests

Finally we can use all the prepared polygons (our lines are now polygons too – remember the buffering step) to cut them out from our forest polygons. This is done in four steps, as I apply the line1, line2, line3 and airport polygons (of course there I have built in a checking logic, which tests whether or not there is something to do at all or not, and skip in the latter case). These are the most time consuming parts of the whole script, as GRASS needs long time to do the cutting. And this was also the reason for an important "content decision":

Content decision:

It turned out, that there is often a very big number of small roads present in the USA global scenery (roads below 6m). When buffering all of them, and putting in the "cutting process" two mayor issues surfaced:
1. The processing took an enormous long time
2. The resulting forest polygons increased in vertex count so much, that the output DSF TXT files often went over 100-150 Mbytes !!!

For this reasons I decided, not to include the roads which are smaller than **6m** in width (this are the class 0 lines in the script). For this reason, for the majority of the roads there are no cut out clearings in the forests. But experiments showed, that this is barely noticeable and results in

normal processing times (in the scenery creation), and later on doesn't kill X-Plane with the gigantic forest overlay scenery tiles.

Now the cutting itself is fairly straight forward and uses the – in the meanwhile – well known v.overlay with the **not** operator:

```
v.overlay -t ainput=polygons_current atype=area alayer=1
binput=lines_buffered1 btype=area blayer=1
output=polygons_current_cut1 operator=not olayer=1,0,0 –
overwrite
```

In case of the buffered object polygons (as described in 3.2.1.) we need to first create an **objects_all** from **objects_buffered1_selected** and **objects_buffered2_selected** which is can for example done with

```
v.overlay ainput=objects_buffered1_selected atype=area alayer=1
binput=objects_buffered2_selected btype=area blayer=1
output=objects_all operator=or olayer=1,0,0 --overwrite
```

The result would look like this

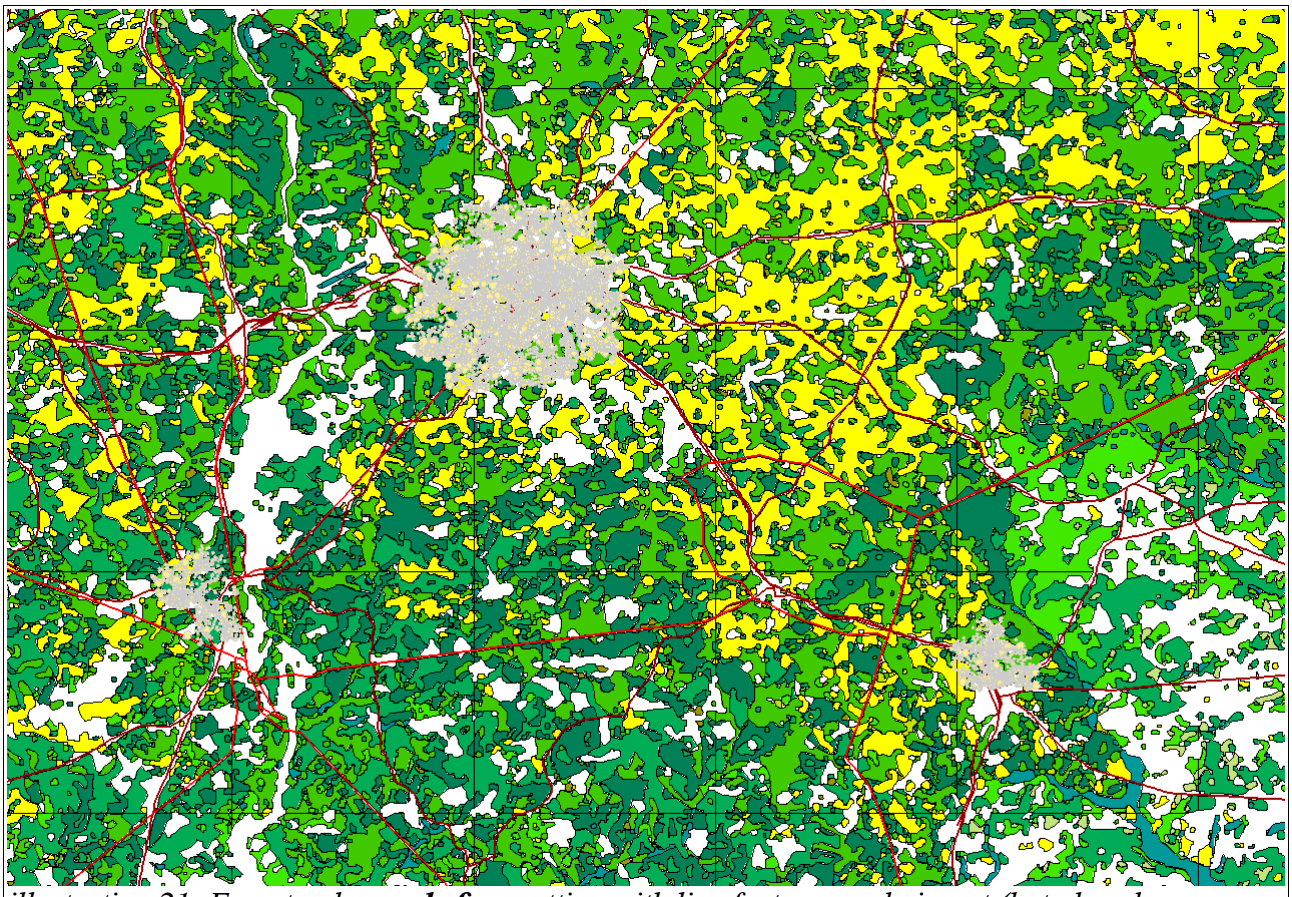


*illustration 21: Forest polygons **before** cutting with line features and airport (but already overalyed as "preview")*

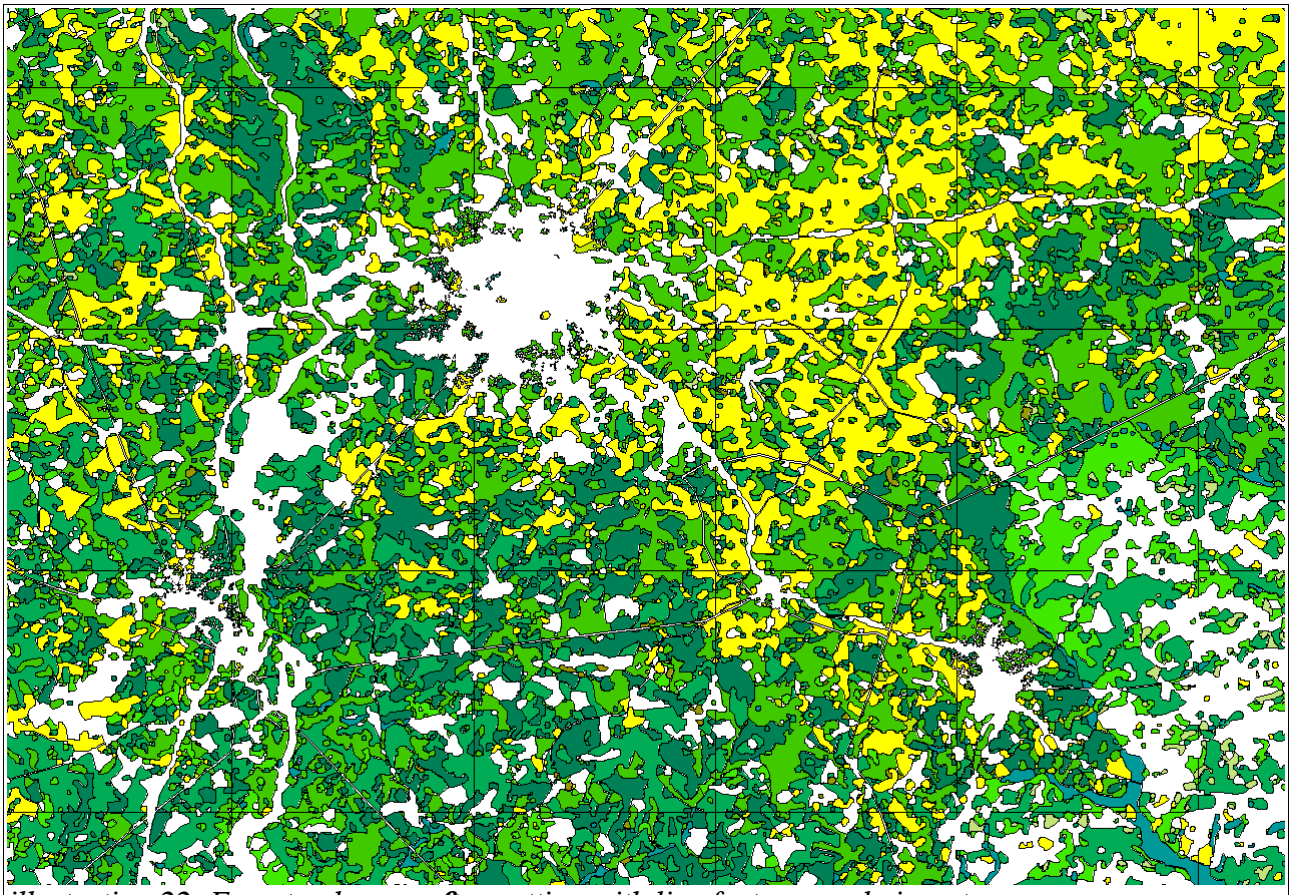


*illustration 22: Forest polygons **after** cutting with line features and airport*

Now we have finally arrived at the point, where the data preparation has been finished, and “only” extracting the data to DSF has to be done.

3.2.3.8 Exporting to the DSF format

In this step, we need to assemble one .txt file, which finally contains the whole DSF 1x1 degree scenery file informations and can be compiled to a DSF file (with the DSF2Text tool from X-Plane).

First I would like to give a “little” background information about the DSF structure – at least when it comes to overlay polygons (the way forest polygons are treated by the simulator). It is comprised of two main parts.

1. At the beginning – let us call it the **header** – there is a list of forest definitions which are used in the scenery tile (there are some more header informations, but they are straight forward, so I won't talk here about them). And their ordering and position in the list is very important, as it implicitly also defines a their reference number. The first forest definition entry has the 0, the second 1, the third 2 and so on ....
2. The second part – the **body** – is the list of polygon definitions, where each polygon is composed from a list of vertex points. And in each polygon “open tag” there is a reference - with the number(!) – back to the forest definition. This defines, how each polygon will be rendered by the simulator.

In the illustration 23 you can see a more detailed logical structure of the DSF file.

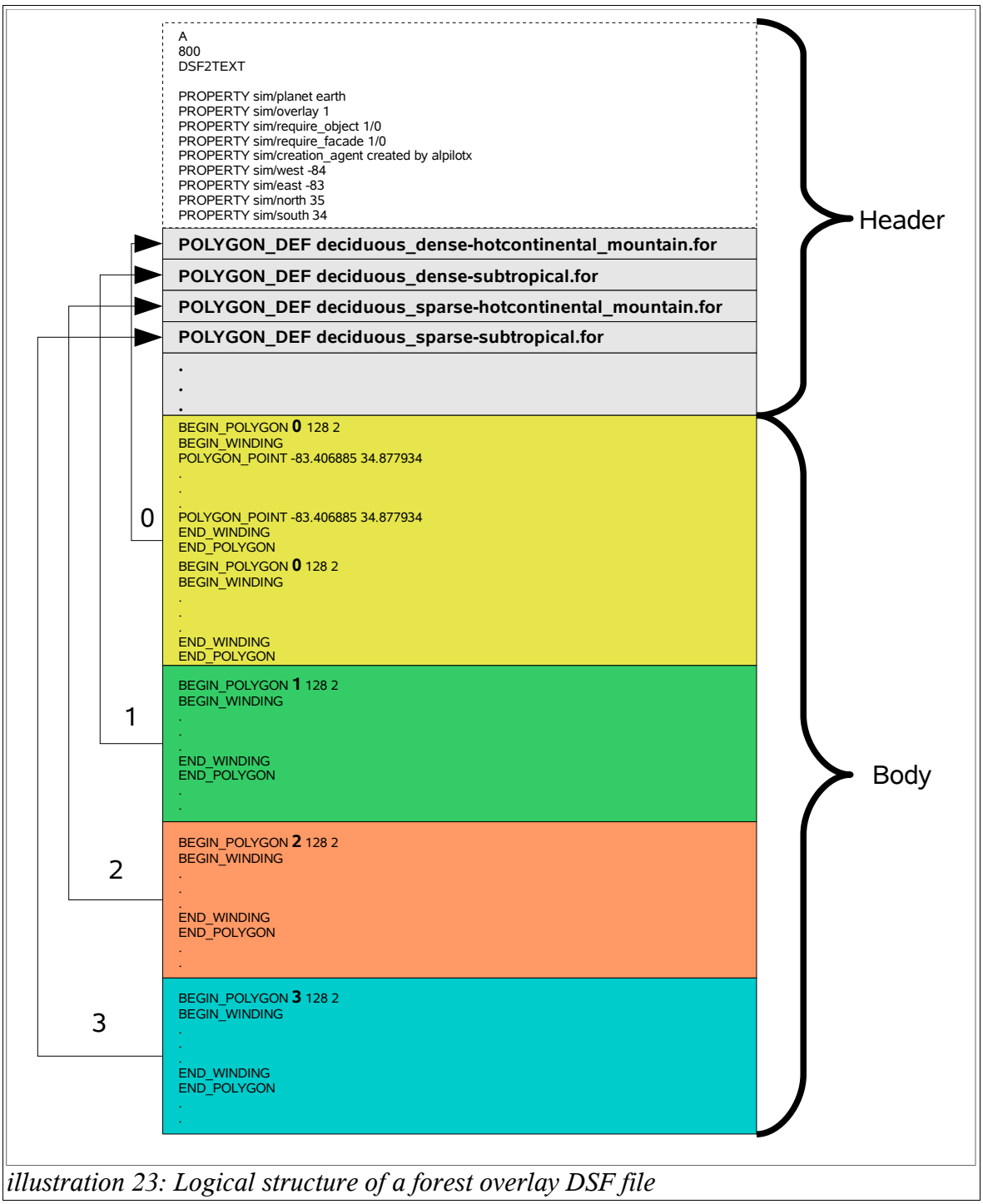


illustration 23: Logical structure of a forest overlay DSF file

Now the transformation process – as it is iterative – creates this structure with two help files. The *header* file and the *body* file. This allows to continually append to both of them, as we iterate through all landclasses and ecoregions. It also retains a counter, which holds the numbering of the current forest definition (as described above) and inserts it in each **BEGIN_POLYGON** statement. This part of the code also outputs an empty .for file (if not already exists) with the correct name in the form *landclass-ecoregion.for*. This allows to see at the end, what all the forest definitions are that were referenced inside the new DSF files. So later one has “only” to fill the .for files to come to a final result in X-Plane.

The transformation process also uses – again – the GML format as an intermediate export format when exporting all the forest polygons. GML comes in very handy here, as it allows – based on it's

well formed XML structure – a fairly straight forward conversion to the DSF text format. It can be achieved by some fancy text replacements which use the Unix/Linux on board tools ***grep*** and ***perl.***

And at the end the process concatenates the *body* and *header* files, and stores the resulting – final – DSF text file in a dedicated directory.

This finishes the description of the whole scenery transformation process. Of course we still need to "compile" the .txt DSF files into real, X-Plane readable .dsf files, but that is achieved easily with the tool DSF2Text (I built a little additional script, which iterates over all .txt files, compiles them into .dsf files and sorts them in the right subdirectory structure, as X-Plane needs it with scenery files).

On the next side, illustration 24 reiterates all the steps of the scenery transformation process in a flow chart like diagram.

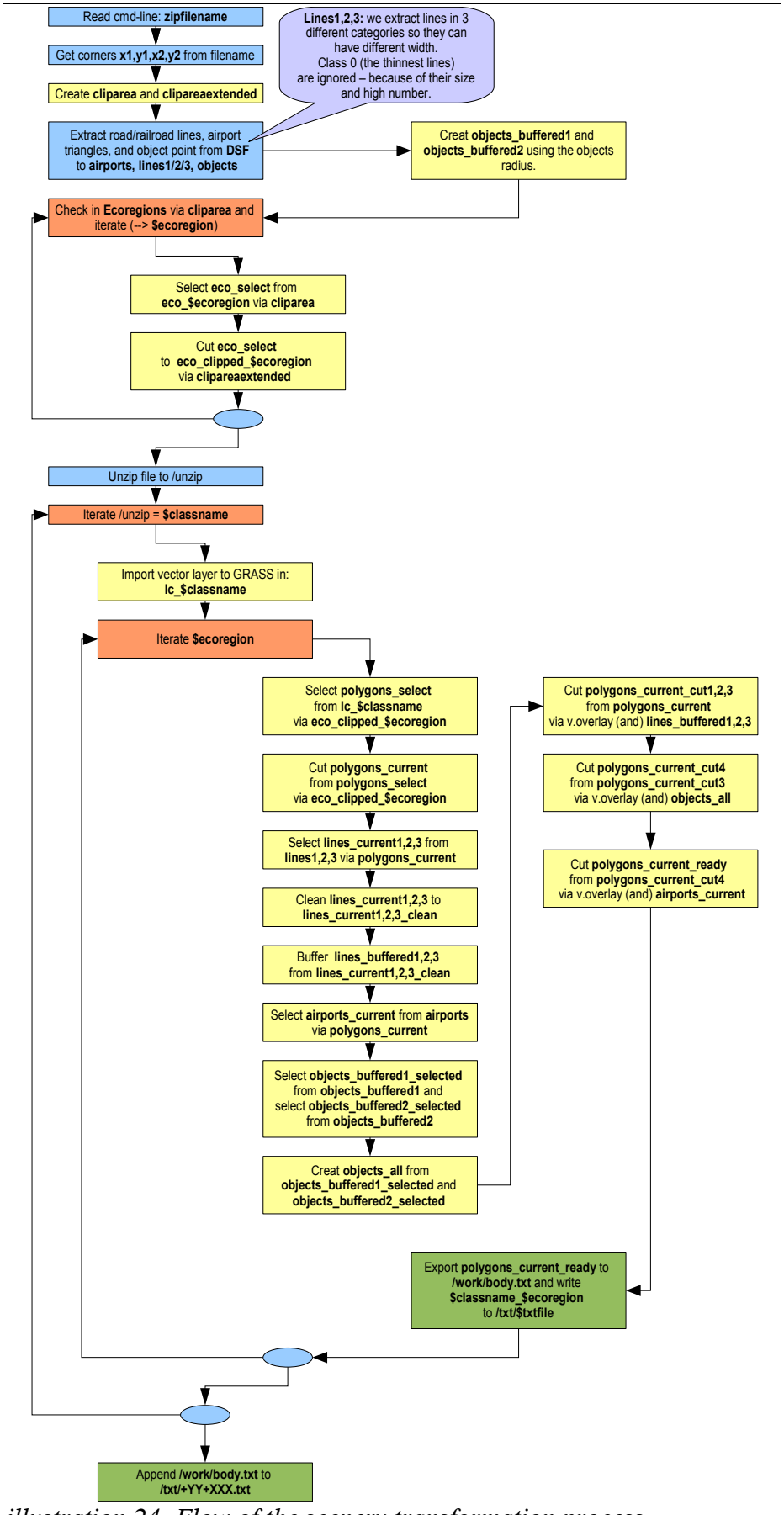


illustration 24: Flow of the scenery transformation process

4 Visual artwork creation

This is the final task in the forestation project. It is very different from the rest, as it represents less of an algorithmic, straight forward process but rather the creative and often more tedious work. It consist of two main building blocks, which have to be addressed:

1. Creating a versatile collection of tree textures.
2. Defining the forest definition files (.for) which build the bridge between pure scenery polygons and textures. They tell the simulator, which tree textures to use, with what distribution, size variations etc.

For creating the large number of forest definition files, we have created a document with OpenOffice Calc, where we – or rather Albert Laubi, the artwork specialist in the team – can easily enter all the tree data, and the composition of the forests. For the "rapid prototyping" the Calc document also has a built in macro (programmed in OpenOffice Basic), that creates all the final .for files with one click.

The following words are from Albert Laubi (alias xflyer) himself:

> USA-Forests
>
> You have seen now, how Andras prepared the soil. And my task was to bring this data adequately to your eyes with planting the corresponding trees from the Adirondack regions to the Everglades swamps, from the Chihuahuan semi desert to the Sierran Redwood (and Sequoia) forests up to the Cascades Wilderness.
>
> This issue kept me busy for months. I had to search for biogeographical information, descriptions and pictures. The trees I created then, mostly represent a certain species and should be recognizable - at least when flying extremely low ;-) The ifinite variety of tree shapes in reality has to be limited here of course to one or two.
>
> As each forest has its distinct composition of different species, I tried my best to make a reasonable approach. But since every definition covers wide areas and the tree-planting algorithm has its own rules, the result still is subject to much generalization.
>
> However I find it fascinating, how sleeping numbers of huge data bases thus can come alive in visualization!
>
> This custom made plant cover of the USA now consists of 38 given ecoregions with 351 individually designed forest definitions and 5735 assignments out of 193 different trees* and shrubs :-)
>
> Combined with stunning exact forest outlines, this extraordinary scenery gives the USA a very plausible look and to you the joy of a superbe view from the cockpit.
>
> *Many thanks to the ones placing their trees at my disposal: Sergio Santagada (X-Planes artwork creator) from whose collection 42 plants come of, 11 are from Andras Fabian and 9 from Geoff Legg.
>
> Albert Laubi (xflyer)

5 List of illustrations

illustration 1: before and after adding forests to X-Plane....3
illustration 2: canopy example....4
illustration 3: landcover example....4
illustration 4: Bayley's Ecoregions as used in the project....5
illustration 5: example for extracted line features and airport polygons....7
illustration 6: how vectorization makes polygons out of raster data....8
illustration 7: (real) example for the coordinate transformation....9
illustration 8: our example canopy area....11
illustration 9: our example landcover are....11
illustration 10: the landcover dataset after the reclassification....13
illustration 11: the landcover after finished filtering steps (landcover_finished)....14
illustration 12: How the 9 layers are split up....14
illustration 13: Transformations between geographical and pixel coordinate space....16
illustration 14: cutting up the polygons....17
illustration 15: flow of the vectorization process....18
illustration 16: Example for extracted line features and airport areas....21
illustration 17: Example of selecting the ecoregions....22
illustration 18: Process of splitting up landclasses to ecoregions....23
illustration 19: Selecting the line segments, which cross a forest polygon (dark yellow line)....24
illustration 20: Buffering the selected road segment (dark read is the area of the new road polygon)....25
illustration 21: Forest polygons before cutting with line features and airport (but already overalyed as "preview")....26
illustration 22: Forest polygons after cutting with line features and airport 27
illustration 23: Logical structure of a forest overlay DSF file....28
illustration 24: Flow of the scenery transformation process....30