A software architecture for multi-LOD planetary surface meshes

C.J. Plooy

Version 2: last update 18-01-2009

Goal

Make a piece of software for rendering planetary surfaces:

- The surface has to look realistic from all viewpoints, both far away (e.g. in a high orbit in space) as close to the surface (let's say a minimum altitude of 1 meter)
- Rendering has to be done at real-time frame rates on normal consumer hardware. It needs to be suitable for use in (space)flight simulators.

Problems and suggested solutions

- Realistic rendering at low altitude requires fine details, but rendering an entire planet in full detail is impossible in real-time and with normal amounts of memory
  - Make a system that continuously switches between different Levels Of Detail. Surfaces far away can be frawn with few faces and coarse textures, while surface close to the camera will have a fine-grained mesh with highly detailed textures.
- Fine detail data is often not available for an entire planetary surface, or it would require too long download times, or too much disk space
  - Append the coarser levels of detail with procedurally generated details, whenever fine details are not available as data files.
- Procedurally generated landscape often don't look completely realistic
  - Then make them more advanced. Hidden layers can be used to contain geological data, which can e.g. be used to modify the parameters of the heigtmap generating algorithms.

Subsystems

The system will be divided in three subsystems, each solving its own part of the problem. These subsystems are:

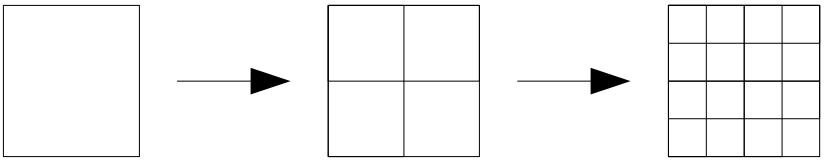
- The datasource subsystem. This subsystem loads data from disk, and procedurally generates missing data. Slow operations (like loading or generating textures) will be executed asynchronously in a separate thread. The other subsystems don't have to worry about this: they simply ask for data, and they get it (or sometimes they don't).
- The LOD subsystem. This subsystem gets the camera position, and continuously optimizes the surface so that the level of detail is always acceptable. It sends requests for new data to the datasource subsystem, and it updates the meshes once new data is available.
- The render subsystem. This subsystem gets meshes and texture data, checks them for recent changes, and if necessary, updates data in video memory. Then, it makes sure the planet gets rendered from the correct camera position. Ideally, this would be the only graphics-API specific subsystem, but for performance reasons it could be necessary to make exceptions to this rule.

Textures

Textures are always rectangular (or square of course), and 3D APIs usually require them to have resolutions that are powers of two. There usually is a videocard-dependent maximum resolution for textures.

To save disk space, it is common to store textures in a compressed image format. The downside of this is that we don't have random access to pixels: even if we only need a single pixel value, we need to load the entire texture and decompress it. If the finer details were stored in a single texture file, loading such a texture would require way too much time and memory. We would have to load the texture for the entire planet, while we only need the small part around the camera position. Therefore, fine-detail textures will be split up into tiles, and each tile will be stored in a separate file. This way, only the tile textures corresponding to the surface around the camera need to be loaded. Another advantage of tiling is that it allows to make a surface with an effective texture resolution that is higher than the maximum per-texture resolution of the video card.

At finer LODs, the surface needs to be composed of more tiles to get the same effect. The following scheme shows a straightforward way for texture subdivision:



Each square in this scheme is covered by its own texture. If we give each texture the same resolution, then going to a finer LOD will give you a 2*2 increase in effective resolution.

The projection method

Because of the rectangular shape of textures, the texture tiling necessarily needs to have a rectangular shape. Unfortunately, there is no perfect way to split up a sphere-like surface into rectangular shapes: there will always be some kind of distortions.

In the choice of the projection, it is convenient to use a cylindrical projection where the edges between tiles follow lines of constant longitude or constant latitude.

The projection that is most commonly used in computer graphics is the plate carrée projection. In this projection, there is a linear relationship between map coordinates and degrees longnitude and latitude. The first LOD could be four textures, with the following ranges:

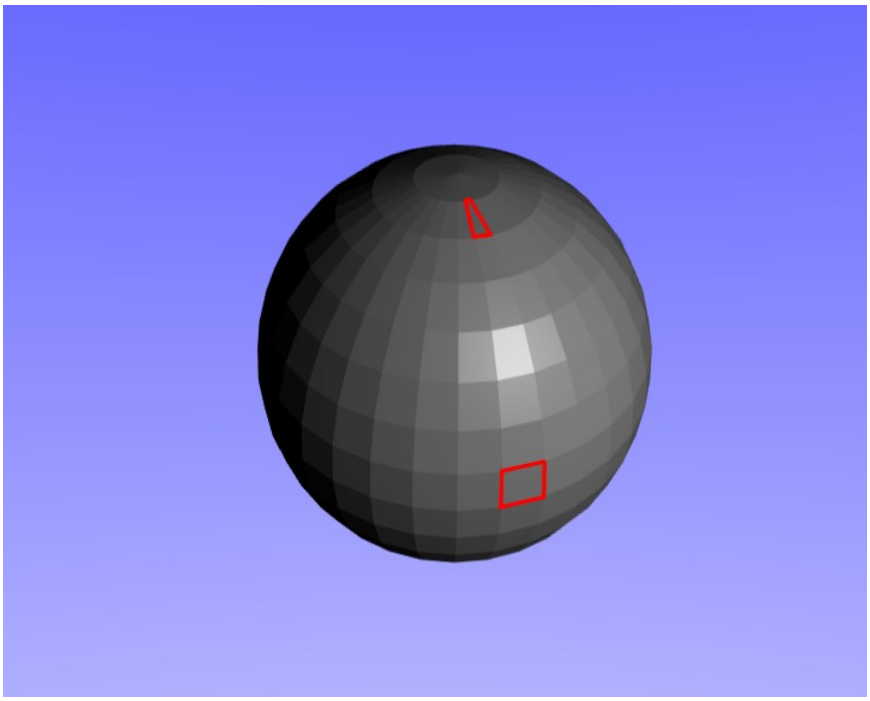
- lon:-180..0, lat: 0.. 90
- lon: 0..180, lat: 0.. 90
- lon:-180..0, lat:-90..0
- lon: 0..180, lat:-90..0

On each subdivision, the latitude and longnitude range is split up into two parts of equal angular size. This allows for easy calculation of the 3D coordinates of the mesh points, but it has the disadvantage that tiles close to the poles have a very non-square length/width ratio. This has the following consequences:

- For realistic effects, landscape generation algorithms have to take into account that their pixels are largely non-square. This leads to increased complexity of the landscape generators.
- When only the kind of subdivision is allowed that simultaneously splits a tile both horizontally

and vertically, the subdivision algorithm will not handle extremely non-square tiles well. It is incapable of generating more square-like tiles on the higher LOD-level, and the tiles will either be too large in one direction, or too small in the other direction, or both. Also, extremely non-square tesselation is visually more distracting than square tesselation. Supporting multiple types of subdivision adds a huge amount of complexity.

- In fact, there are extreme cases for which it is necessary to limit the subdivision based on the *smallest* side of a tile, to avoid unlimited subdivision, and the resulting crash of the application. As a result, subdivision can not take place in the polar regions beyond a certain level.



Afbeelding 1: With the plate carrée projection, tiles close to the equator have a near-square shape, but close to the poles they are extremely stretched.

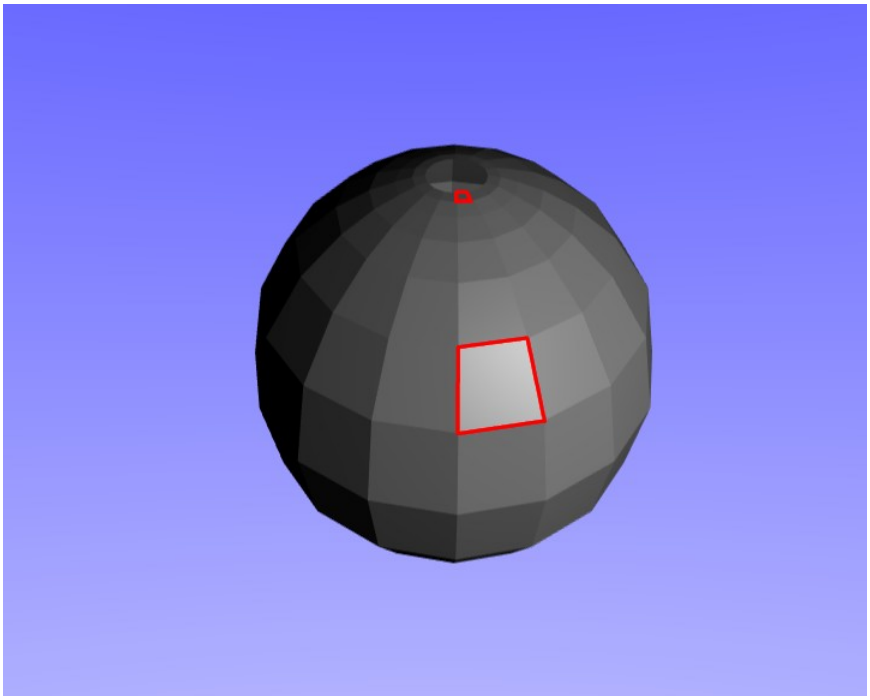
The result is that the plate carrée projection can not be used without either adding a lot of complexity, or adding a lot of artifacts. Because simplicity was the reason to use plate carrée in the first place, and this goal was not reached in the end, another projection was considered as well: the *Mercator* projection.

In the Mercator projection, horizontal and vertical lines in the image also correspond to lines of constant latitude and longnitude. In fact, the relationship between the horizontal map coordinate and degrees longnitude is the same as in plate carrée. The difference is that in the Mercator projection, when you move away from the equator, the distance between constant latitude lines will increase. Close to the poles, the map is “stretched” vertically. This stretching is so extreme that the vertical map coordinates of the poles are infinite.

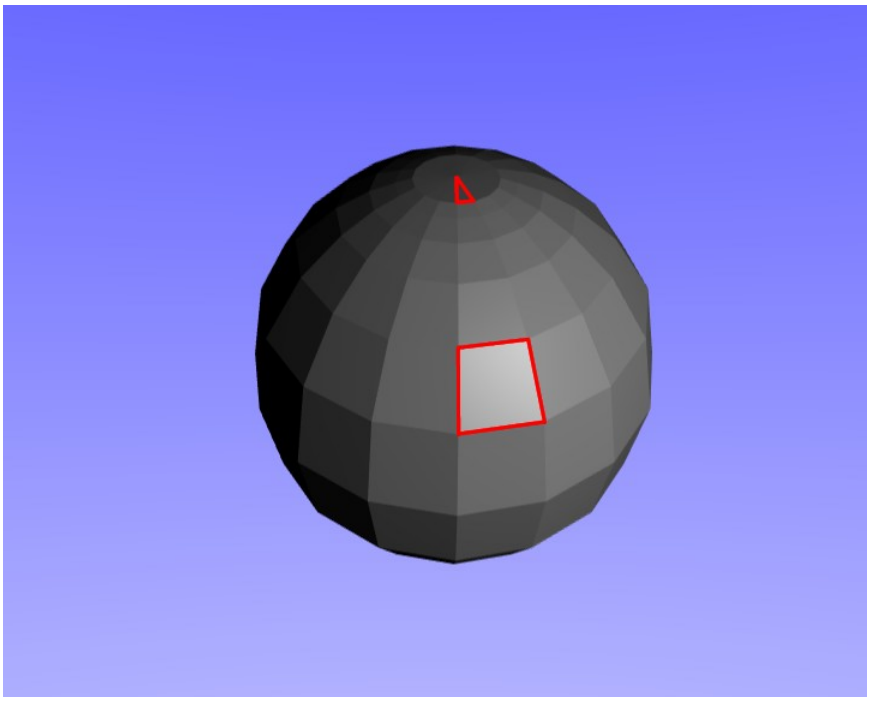
At first, this sounds like it only makes problems worse, but the advatage of the Mercator projection is that on every location of the map (except for the poles) the tiles will approach a square shape, when the LOD is high enough. This takes away all the disadvantages of non-square tiles, but in return we need to

solve the problems related to the Mercator projection. One problem is that, within a tile, pixels on the equator-side are physically larger then the pixels on the pole-side, and landscape generation algorithms may need to compensate for this. This is not a major concern, and maybe it is not needed at all, because the landscape generation algorithms typically operate at a high-LOD level, where these distortions are minimal. On the low-LOD level, the problem is that map data is usually available in the plate carrée projection, so it needs to be reprojected before it can be used. The last problem is the infinite vertical size of the maps, and the fact that the poles themselves are actually not present on any map.

While the poles are not present on the map, they need to be displayed somehow. The vertices that form the polar end caps of the planet mesh need to have *some* non-infinite texture coordinates. A possible solution could be to stretch the tiles around the poles, so that they completely cover the missing part of the polar region. Obviously this will cause some distortion, but the distortion will reduce when the viewer comes closer and the LOD increases. Hopefully, this will make the distortion not too obvious.



Afbeelding 2: With the Mercator projection, all tiles have a near-square shape, but close to the poles there is a hole.



Afbeelding 3: Mercator projection with fixed polar caps. Only the stretched tiles near the poles are highly non-square.

Meshes

Meshes consist of a list of vertices, connected by faces, usually triangles. Textures are projected on faces by assigning texture coordinates to the faces' vertices. When texture coordinates exceed the range [0, 1], the texture will either be wrapped (resulting in a repeating pattern), or clamped (resulting in the rest of the face being filled with the texture's edge color). Neither effect is desired for a mesh with tiled textures as described in the previous section. As it is impossible (at least without complicated pixel shader tricks) to let one texture start where the other one ends (within a single vertex), there need to be mesh edges on the same places where the texture tiling has its edges. Therefore, the mesh needs to follow the same rectangular subdivision scheme as the textures.

Every quad will be associated to only a single texture tile, and all quads associated to a certain texture tile will usually share vertices. Also, texture context switches can be expensive operations in some 3D engines. Therefore, it makes sense to organize the mesh in groups, where each group contains the quads associated to a single texture tile.

Like the level of detail of the textures, the tesselation inside a group can also be adapted continuously to the camera position. However, this can be seen as a secondary feature, and its implementation does not need to have a high priority. A fixed tesselation level for a group will already give a continuous LOD adaptation, caused by the LOD adaptation of the texture tiles. The tesselation at the edges of the group needs to be adjusted to the tesselation of neigboring groups.

The datasource subsystem

On the interface to the other subsystems, the datasource subsystem allows requests for data. For each texture tile, the following data will be available:

- A surface altitude map. This will be used for positioning the vertices when the mesh of the tile is generated and/or updated. It can also be used for collision detection in a physics engine. Also, it can be used to determine the "roughness" of the terrain at a certain location. Very smooth terrains need fewer mesh subdivisions than rough terrain parts.
- An internal representation of the terrain. This may contain, for instance, high-accuracy versions of the texture and height data, and 'hidden' layers, e.g. containing geological information. This is used when a higher LOD level needs to be generated procedurally.
- Multiple textures, for instance for surface color, shininess, night color and bumpmapping. The textures may already be stored in a format used by the 3D API, through the use of derived classes for the texture objects.

Other subsystems can place requests on this subsystem to load certain texture tiles. A certain tile can only be generated or loaded when its area and its neighboring areas are already loaded at exactly one lower LOD. The datasource subsystem will make sure the dependencies are loaded first, before the requested tiles are loaded. The dependencies exist because the terrain generation algorithms need to be able to create a higher LOD based on the maps of a lower LOD.

Internally, much of the data acquisition (file loading, procedural generation) happens in a separate thread. The datasource subsystem knows what data to load based on recent requests on its interface to the other subsystems. There will also be a mechanism for unloading data that is no longer needed.

The LOD subsystem

This subsystem will determine, based on the camera position, which texture tiles need to be loaded, and which can be unloaded. It will also update the meshes when new data is available, or when meshes are dynamically updated based on the camera position.

The render subsystem

This will probably the simplest of all subsystems. It will do all the important communication with the video card, and do the actual rendering. The rendering subsystem will need to be adapted / rewritten for each application in which the landscape rendering is used, to integrate the landscape rendering with the rendering of other parts of the scene. The render substystem will need to have some way to determine whether meshes or textures have changed since the last frame. Parts that haven't changed don't need to be sent to the video card again on each frame.

Because of the wide range of depths in many landscape scenes, the render subsystem probably needs to perform multiple passes with multiple Z-buffer settings. It should be able to determine for each mesh group in which depth ranges it should be rendered.

While double precision floating point numbers are accurate enough to express even the most detailed vertex positions w.r.t. The center of the planet, this is not the case for single precision floats, and most video card drivers work in single precision. Therefore, vertex position probably first need to be

expressed relative to the camera position, before being given to the 3D API.

Ideally, the render subsystem would be the only part with specific calls to a 3D API like Direct3D or OpenGL. However, it is possible that this design could involve too many copies and conversions. If this kind of things become performance bottlenecks, an alternative approach can be used on these places, by wrapping 3D API functions with API-independent classes, which can be used in the other subsystems. For instance, a texture class (encapsulating a graphics API texture object) can be used in the entire pipeline from file loading in the datasource subsystem to the final rendering, instead of a generic texture representation in a simple pixel array. This way, the intermediate algorithms can still be implemented in a graphics-API- and platform-independent way, without hurting performance too much.