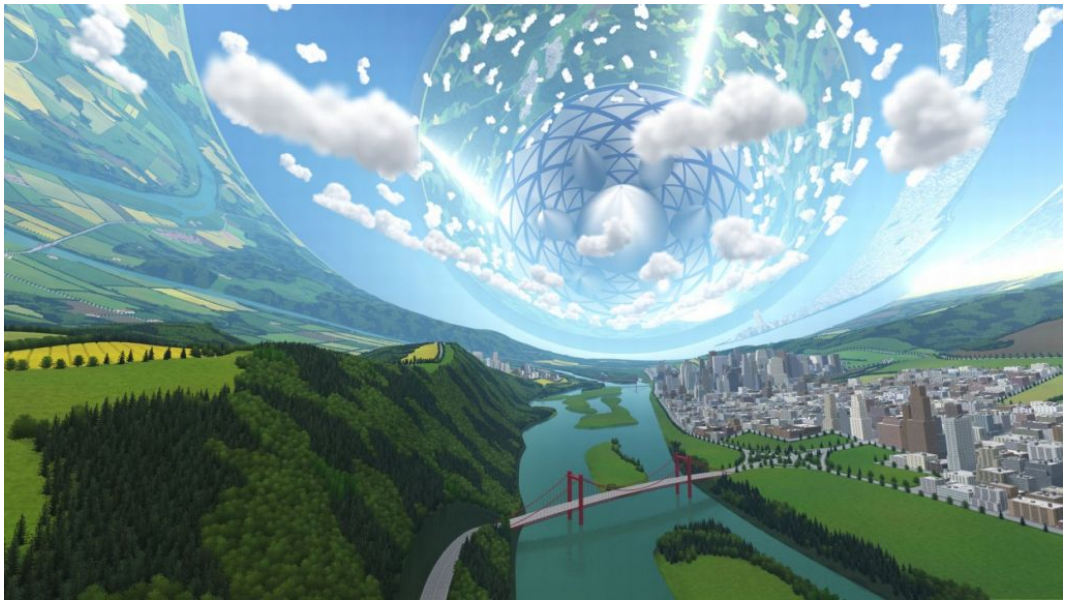
Modeling and Rendering Rama

Eric Bruneton

August 20, 2005



Contents

1 Introduction

Rama is a hollow cylindrical spacecraft invented by A.C. Clarke in his novel "Rendez vous with Rama" [1]. Its internal diameter is 16 km, and its external length is 50 km. Rama spins around its axis in order to simulate gravity at its surface. Figure 1 shows a schematic view of Rama, which can be divided in five zones: the south pole, the south plain, the cylindrical sea, the north plain and the north pole.

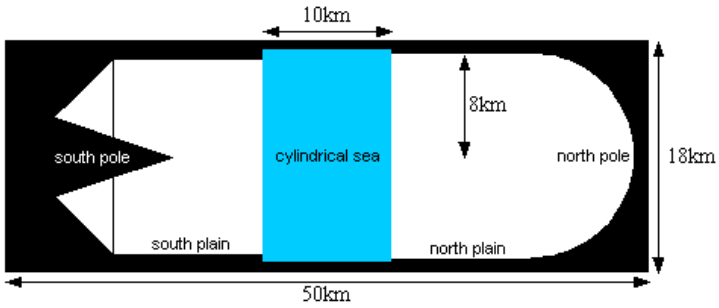


Figure 1: Schematic view of Rama

Soon after I read the book, I began to try to model and render 3D pictures of Rama. The first (see Figure 2) and second (see Figure 3) versions were modeled and rendered with custom tools implemented in Java, including a custom ray tracer.

Figure 2: Rama model, first version (1998-1999)

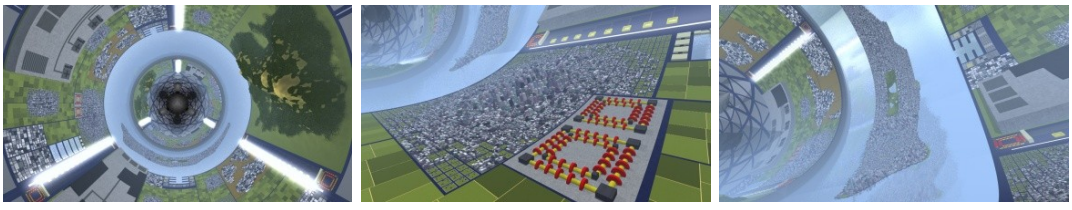


Figure 3: Rama model, second version (1999-2000)

I didn't use existing tools, because I though they would not be able to handle the very large scenes that I needed (the second version includes several millions of cones for the trees in the mountains, and half a million cylinder patches for the plains and for New-York - and, with a classical modeler, each patch would require four polygons and a cylinder surface patch). What is sure is that the model could not have been defined by hand with a classical graphical modeler, due to the very high number of objects to be modeled. It is less sure that the model could not have been rendered with existing tools, and with reasonnable performances (in fact the third

version is rendered with existing tools, but this required a different modeling strategy). Anyway, writing my own ray tracer was an enjoying exercise :-).

The second version was much more detailled than the first one, but I found it was still not giving any sense of the very huge size of the spacecraft. I though the main reason was the lack of natural and recognizable elements (vegetation, clouds, rivers, non rectangular shapes, ...). For the third version, started in 2004 and presented in this document, I therefore wanted to have a much more realistic and detailed model, which would not be limited to rectangular patches. I also wanted more detailed textures, realistic trees, clouds, ... It was then obvious that my limited ray tracer would not be able to render such a model efficiently. But it was also obvious that I could not use a classical modeler, due to the size of the model (it was already the case with the first two versions, and here I wanted a bigger model). So, as in the previous versions, I needed to write a program to generate the model, instead of drawing it manually with a graphical modeler.

But, at first sight, even this solution seemed unrealistic, due the level of details I wanted to achieve. For example, I wanted to model the plains with a resolution of 10 cm or less, in order to correctly represent the trenchs near roads. But, at this resolution, a height map of one hundred billion (10^{11}) pixels is necessary to cover the total surface of the two plains of Rama, i.e. 1000 km^2! The solution comes from the idea that, in order to compute an image from a given view point, it is not necessary to model the whole spacecraft with a very high level of details (for example, it is not necessary to model a house which is seen from a distance of 30 km with many details!).

So, at the end, I decided to wrote a program that could generate a model of Rama *adapted to any given view point* inside the spacecraft (which is harder to write than a program to generate the whole spacecraft at the maximum resolution: in both case the program must be able to generate a high resolution model of any portion of the spacecraft but, in the second case, it must also be able to generate lower resolution models).

2 Modeling

As said in introduction, the Rama model is *automatically generated*. More precisely, I wrote an ANSI C program that, given an arbitrary view point, generates a 3D model (in RenderMan [2, 3] format) of Rama, from a set of 2D maps, adapted to this view point. This section presents the 2D maps used to generate the models, and the process used to convert these 2D maps to 3D models (the models are rendered with 3Delight [4]).

Note: the overall shape of the spacescraft has been modeled manually (see Appendix A) with three cylinders (for the north plain, the cylindrical sea and the south plain), a half sphere for the north pole, a cone for the south pole, and seven cones for the horns.

2.1 Maps

Rama is divided in seven regions: three regions for the north plain (between the three north lights), three regions for the south plain, and one region for New-York. Each region is described with seven 2D maps (see Figures 4, 5 and 6):

- Height map: this map gives the altitude at each point of a region, at a ten meters resolution.
- River map: this vectorial map gives the shape of the rivers. Small rivers are modeled with a single open curve (made of a set of cubic Bezier curves) with a constant width. Big rivers are modeled with a closed curve, which allow their width to vary, and with optional additional closed curves, inside the main one, to model "islands".
- Road map: this vectorial map gives the shape of the roads. Each road is modeled with a set of cubic Bezier curves with a constant width. The roads are not independent curves: they are connected together in a graph.
- City map: this vectorial map gives the limits of the cities. Each city region is described with a closed curve (made of a set of cubic Bezier curves).
- Field map: this vectorial map gives the limits of the fields. For each *zone* implicitely defined by the road map (a zone is a maximal area enclosed between roads, that do not contain roads itself), the field map defines a graph, made of line segments, and which defines the field limits.
- Forest map: this vectorial map gives the shape of the forests. Each forest is modeled with a closed curve (made of a set of cubic Bezier curves).
- Bounds map: this vectorial map gives the bounds of the region. This map is made of a single polygon (which is a simple rectangle for all the regions except New-York).

The data for these maps come from free data that represent some real regions of the United States (which were carefully chosen to be not too flat and not too mountainous, and to include rivers, forests, and cities. The New-York region maps come, of course, from real Manhattan maps. Note also that the maps and heights have not been scaled in any way). More precisely, the height maps come from free satellite images of these regions [5]. The river and road maps comes from free vectorial maps of the same regions [6]. Unfortunately these free vectorial maps were made of polylines instead of smooth curves. It was therefore necessary to manually improve

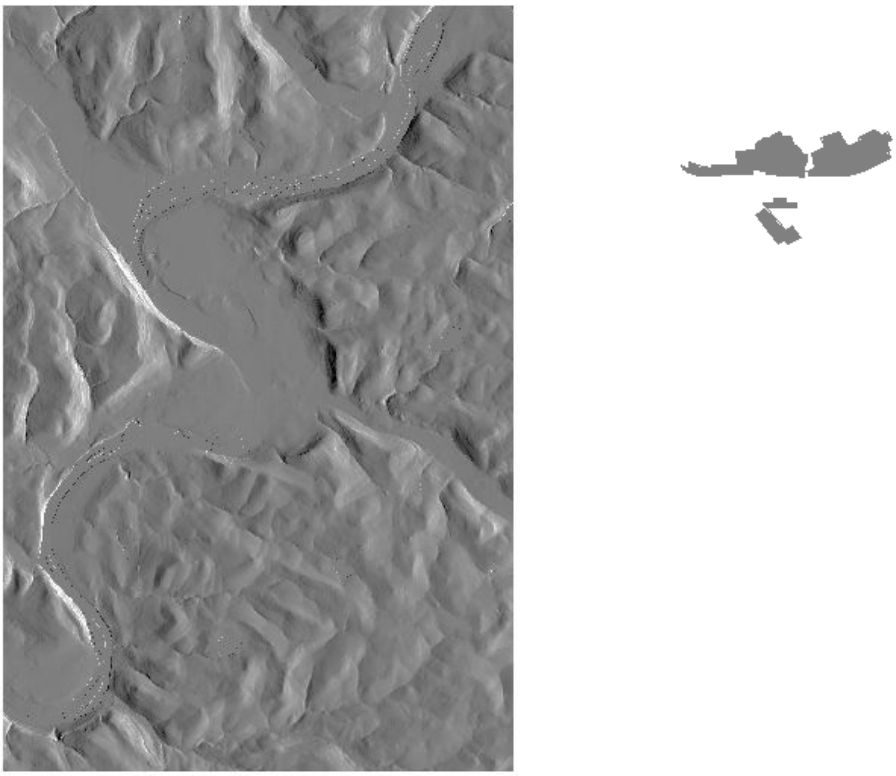


Figure 4: (Shaded) height map and city map of the first region

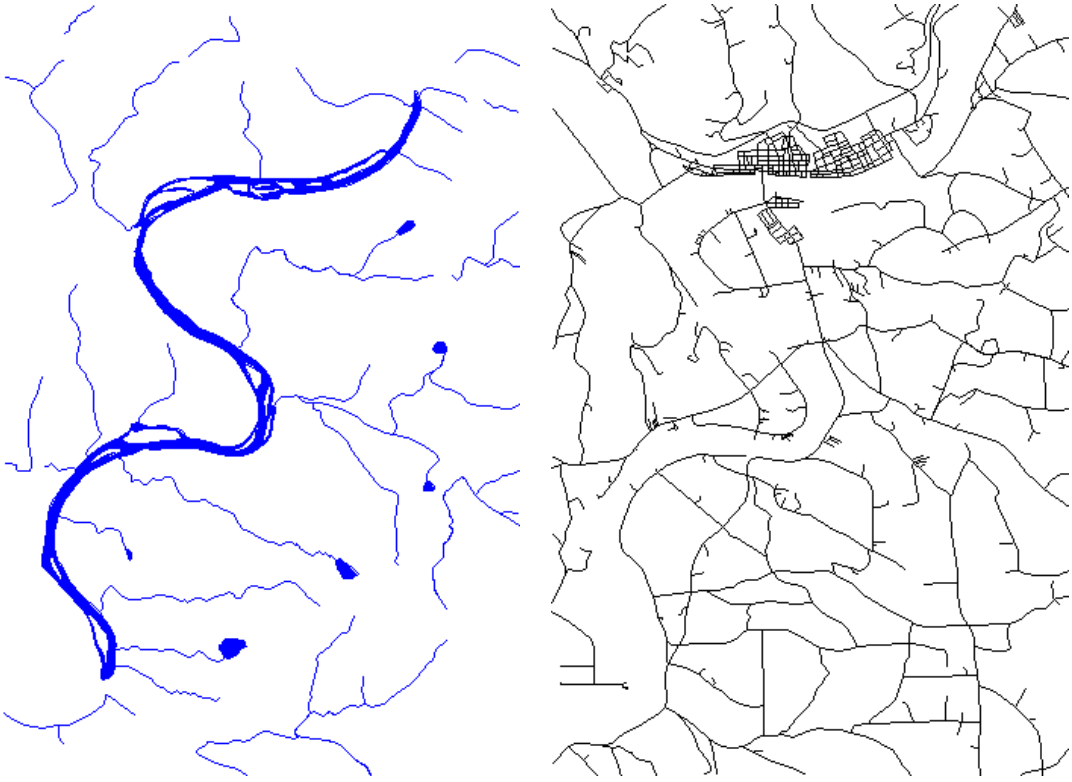


Figure 5: River and road maps of the first region

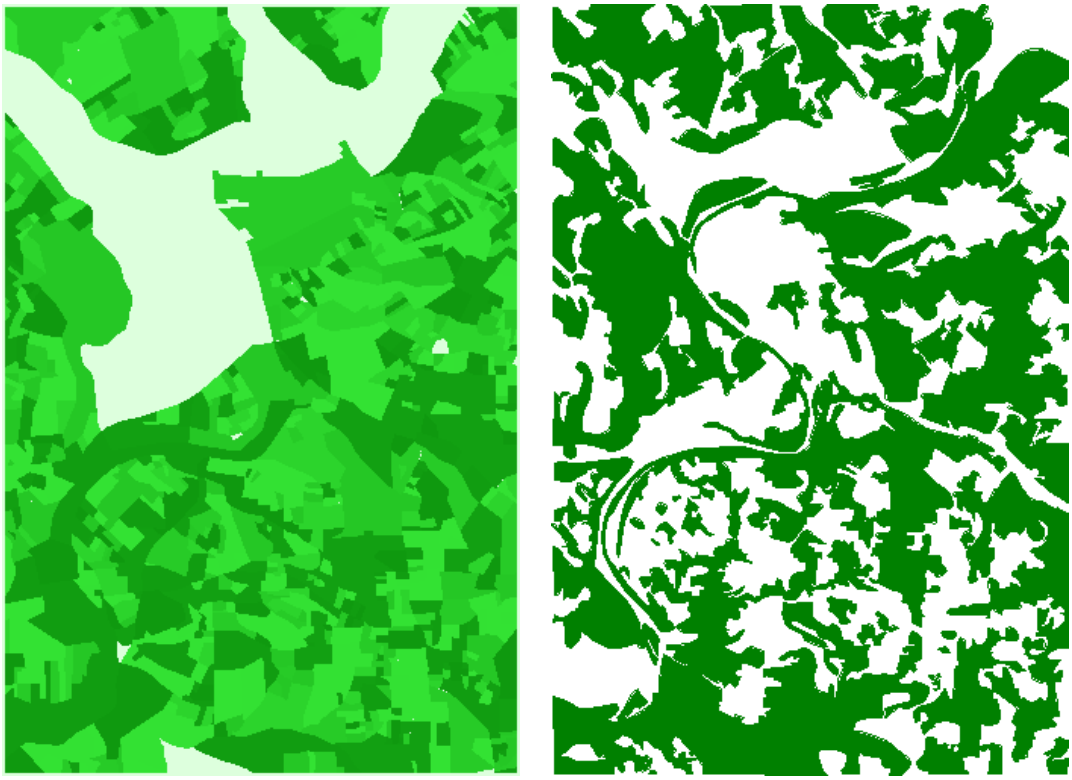


Figure 6: Field and forest maps of the first region

these maps, which was a tedious process (I did this with my own graphical editor, that I wrote in Java specifically for Rama). The other maps were created from scratch (with the help of satellite photographs of the previous regions [7]).

2.2 Terrain

For each region, the terrain model is generated as a right triangulated irregular mesh (see Figure 7), which means that each triangle in the mesh is a right triangle, and that triangles can have different sizes. The advantage of an irregular mesh, compared to a regular mesh, is that it is possible to represent small details near the camera without requiring to use small triangles everywhere (just use triangles whose size increases when their distance from the camera increases). In fact, using regular meshes would have been impossible in the case of Rama: in order to use 10 cm triangles near the camera, it would have been necessary to use 16 billion triangles for a single region! The algorithm used to generate the irregular meshes is the one described in [8]. The implementation is derived from the implementation of this algorithm in TopoVista [9].

The mesh is not directly generated from the height map. In fact it is generated from height values that are computed from the height map, the river map, the road map and the city map, in the following way (see Figure 8 and 9). The height h of a point P is:

- if P is inside a small river, h is the height of the nearest point on the river axis curve (zone a in the figure). The height of any point on the river axis curve is computed by interpolation from the rectified heights of some reference points on this axis (the rectification is used to ensure that the water level decreases from upstream to downstream. Since the height

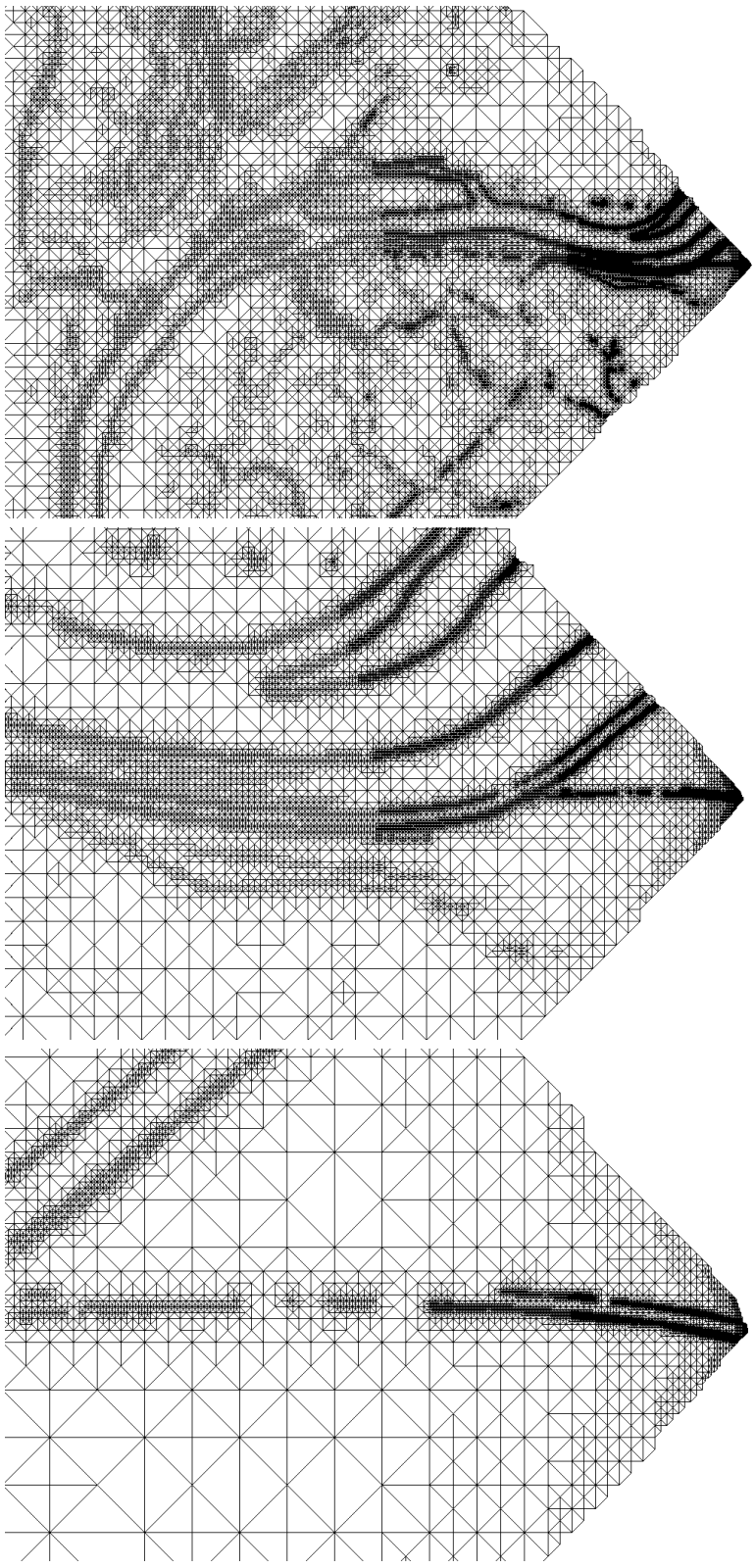


Figure 7: Successive zooms on a *single* right triangulated irregular mesh. This mesh has been generated for an observer at the extremity of the right corner. As can be seen the size of the triangles decreases when their distance to the observer decreases. It also decreases when necessary to represent relief details. Note also that triangles that can not be seen from the observer are not generated. This mesh is the one that has been generated for the images at the end of this section.

map data is not always accurate, and since the river curves are manually defined, the unrectified heights do not necessarily verify this constraint). This ensures that the river surface is flat, smooth, and that its height decreases from upstream to downstream. If P is inside a big river the process is similar, but a little more complicated.

- if P is near a river, h is the minimum of the height given by the height map and of a height computed from the distance of P to the river (zone b in the figure). This ensures that river borders are smooth.
- if P is inside a road, h is the height of the nearest point on the road axis curve (zone a in the figure). The height of any point on the road axis curve is computed by interpolation from the rectified heights of some reference points on this axis (the rectification is used to raise the road at bridge ends). This ensures that the road surface is flat and smooth.
- if P is near a road, there are two cases. If P is inside a city zone, h is the height of the road itself, plus ten centimeters (to model sidewalks). If P is not inside a city zone, h is the minimum of the height given by the height map and of a height computed from the distance of P to the road (to model trenchs - zone b in the figure).

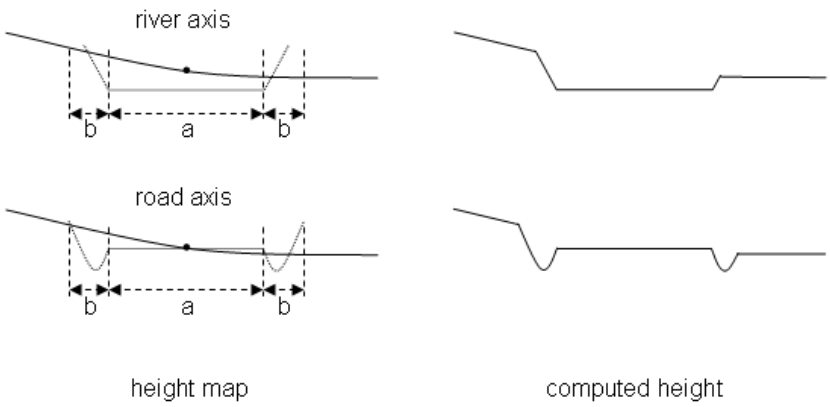


Figure 8: Heights computed from the height, river and road maps

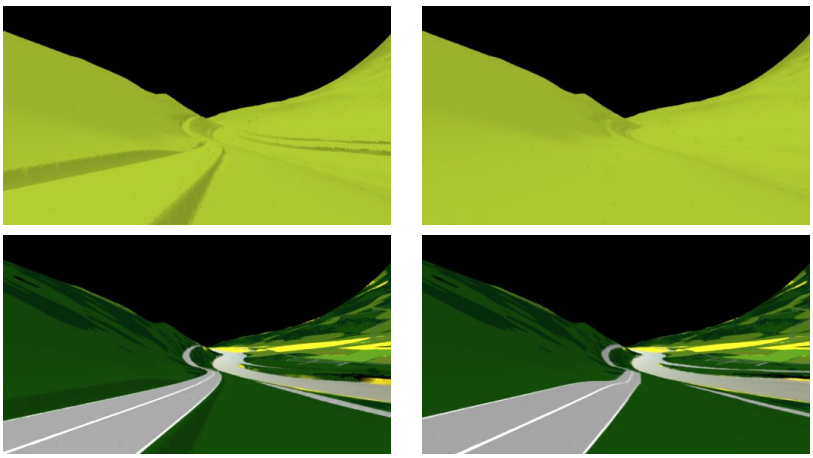


Figure 9: Rama with or without heights computed from roads and river maps

This process can give details that are much smaller than 10 meters, which is the resolution of

the height map. This is why it is sometimes necessary to use triangles as small as 10 cm, and why it is therefore necessary to use an irregular mesh to model the terrain.

Note: right triangulated irregular meshes cannot have arbitrary bounds. In the case of New-York, the generated mesh is post-processed in the following way: triangles that are not completely inside the region bounds are removed. Then a constrained Delaunay triangulation of the New-York bounds (a polygon) is performed (with the Triangle [10] program), with the remaining triangles as constraint.

2.3 Buildings

The buildings are generated from the road and city maps, in the following way, inspired from [11] and [12] (see Figure 11):

- the road graph is analyzed to find the zones enclosed by roads:
  - first, remove all roads that are not connected to other roads at their two ends.
  - then choose a road, follow it in one direction until a crossing is found. Then continue in the same direction on the rightmost road. Continue like this until you go back to your starting point. At this point a zone has been found.
  - choose another road and repeat the same process, until all roads have been examined, in all directions.
- the zones that are not in a city are removed.
- each zone is reduced to leave space for roads and sidewalks.
- each zone is subdivided in two blocks along its major axis, and so on recursively, until each block is smaller than a given size.
- the blocks that do not have direct access to a street are removed.
- each block is reduced to leave space between them.

Then a building is generated for each block with a recursive process. At the first step, a building is generated as a single prism. This prism is then sub divided into two prisms, by using a subdivision rule randomly chosen between a set of predefined rules (see Figure 10). These two prisms are themselves subdivided into two prisms, and so on recursively (see Figure 12). The implementation of this algorithm uses the General Polygon Clipper library [14] to compute polygon intersections, unions or differences.

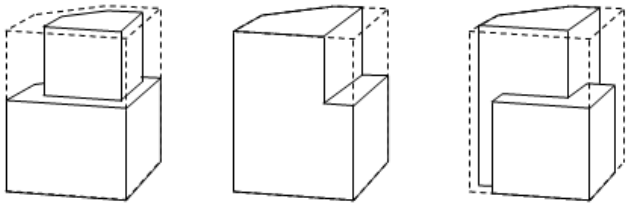


Figure 10: Some rules used to generate buildings

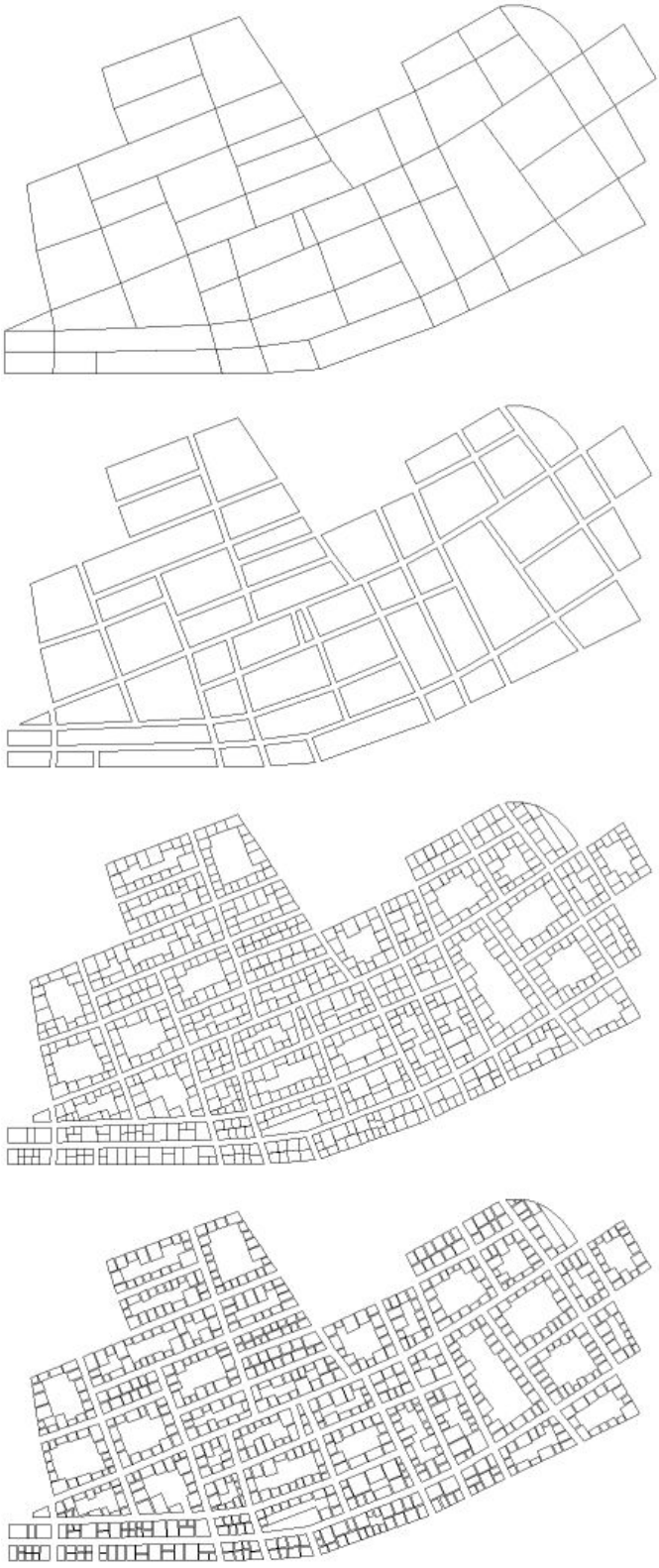


Figure 11: The different steps of the building blocks generation process

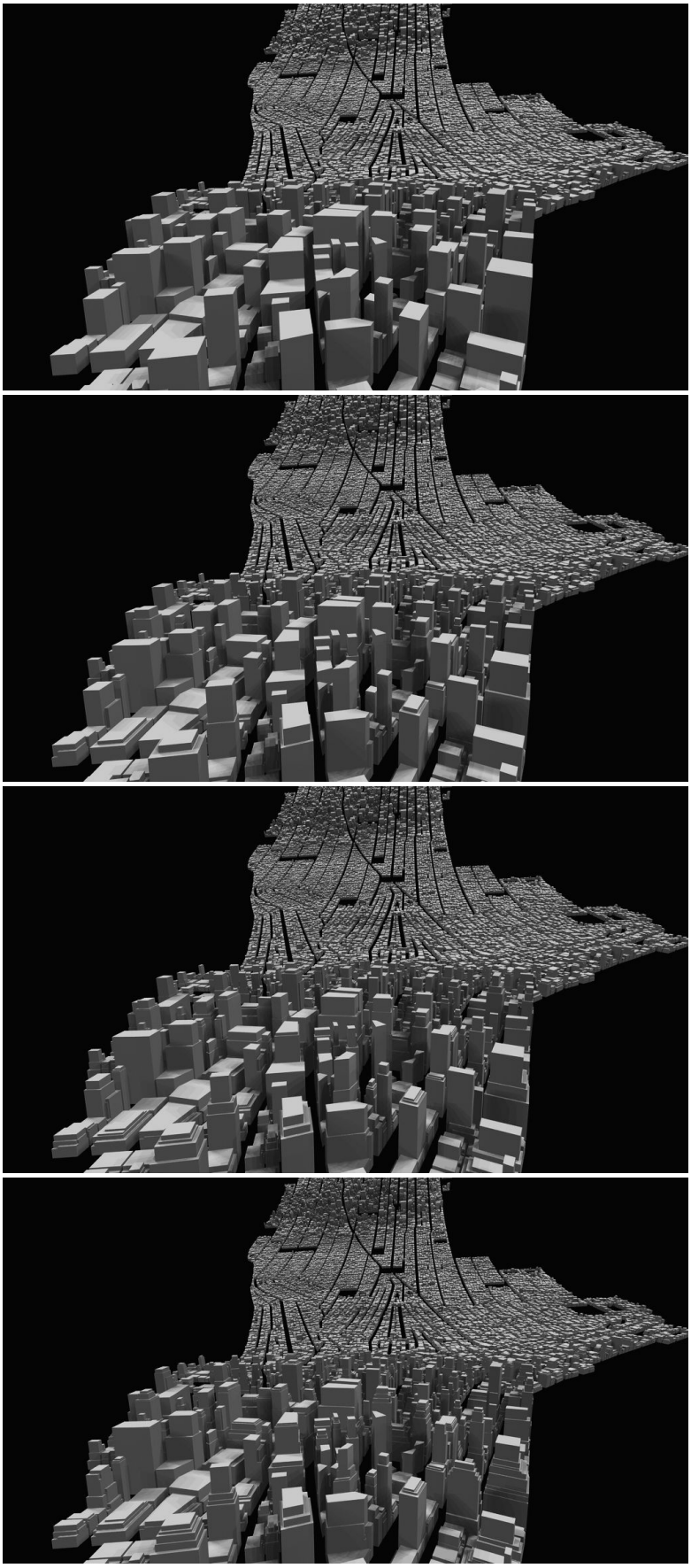


Figure 12: The different steps of the New-York buildings generation process

2.4 Vegetation

The vegetation in Rama is mainly made of trees in forests, in hedges between fields, and at road borders. The hedge trees are generated at the boundaries of fields, at more or less regular intervals. The road trees are generated at both sides of roads, at regular intervals, except in city areas. Finally the forest trees are generated in forest zones, at more or less regular intervals (measured on the ground, and not in projection).

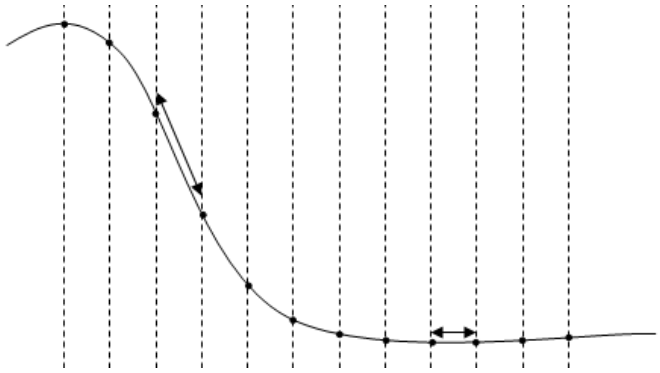


Figure 13: Regularly spaced trees in projection are not regularly spaced on the ground

Distant trees (i.e. trees which cover less than two or three pixels) are generated as points, with the `RiPoints` RenderMan primitive. Other trees are generated as procedural primitives. The procedural primitive works as follows. At initialization, some precomputed geometry files are loaded in memory: one file for the trunk, several files for level 1, 2 and 3 branches, and one file for leaves. Each file comes in two versions: a detailled version based on triangle meshes, and a simplified version based on points and lines primitives. Then, in order to generate a tree, some or all of this precomputed geometry is emitted to the renderer, based on the apparent size of the tree in the image, as described in section 5 in [16].

The precomputed geometry files are generated with a modified version of Arbaro [17], a Java program that implements the algorithms described in [16]. I translated the original program into C, and modified it to make it generate RIB files instead of POV-Ray files. Three tree models are used for Rama, including a pine tree model and a poplar model (see Figure 14).

Figure 14: The tree models used in Rama

2.5 Clouds

Clouds are generated by using a simplified version of the algorithm described in [18]. First, for each cloud, some spheres are generated, at random but next to each other, in order to define the global shape of the cloud (see Figure 15). Then smaller spheres are generated at the surface of the previous spheres, more or less of the same size, and more or less equally spaced. Finally the volume defined by these spheres is filled with particules, i.e. identical spheres, more or less regularly spaced inside all the volume. Only the particles are preserved, i.e. the spheres used in the first steps are discarded once the particles have been generated.

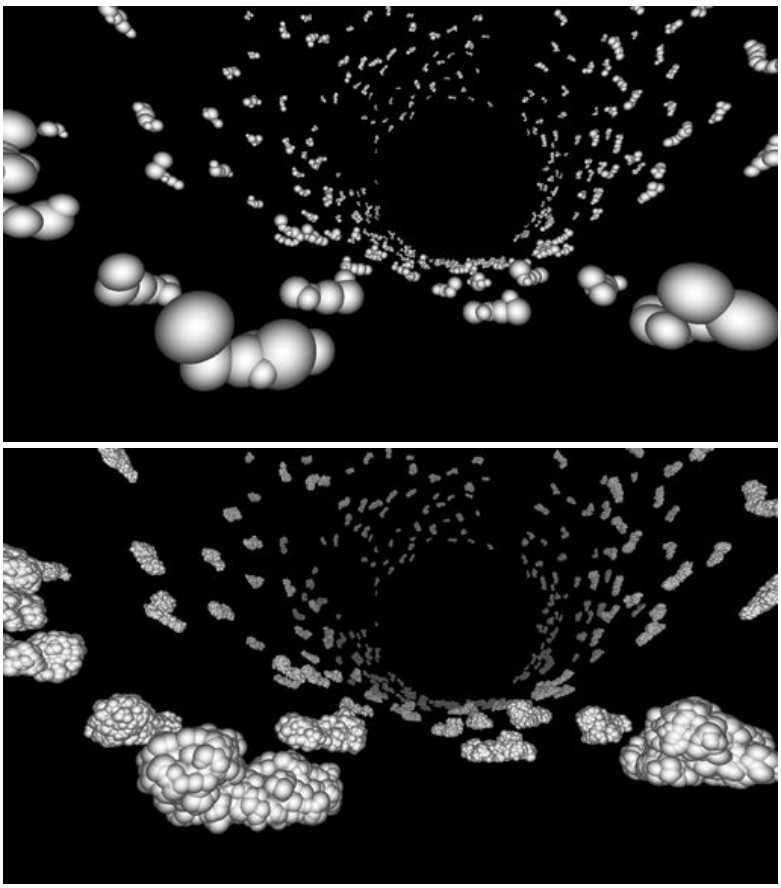


Figure 15: The different steps of the cloud generation process

3 Lighting

3.1 Light sources

The lighting of Rama is very special. Indeed there is not a single punctual light source, like outdoor scenes on Earth, but six linear light sources (see Figure 16) that can absolutely not be approximated by punctual lights, since they are ten kilometers long!

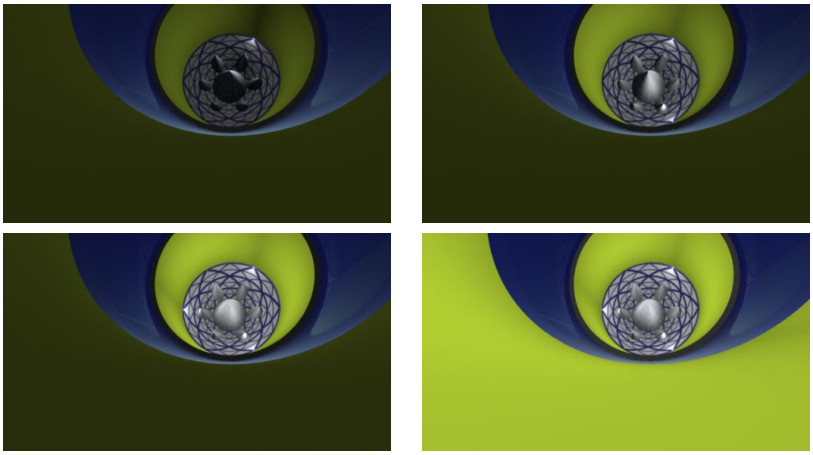


Figure 16: Rama with one, two, three and six linear lights on

In order to avoid using $6 * 10$ or $6 * 30$ punctual light sources to simulate these linear sources, I used an analytical model to compute the contribution of light sources. First, I supposed that the linear lights are made of millions of spot lights, so that the intensity of light decreases when the angle i to the light direction increases. To simplify computations I supposed that the light intensity of a single spot light in direction i is proportional to $cos(i)$ (and is also, of course, inversely proportional to the square of the distance to the source). Then I supposed a Lambertian model for diffuse reflection of surfaces, and I computed analytically the contribution of a linear light with this diffusion model.

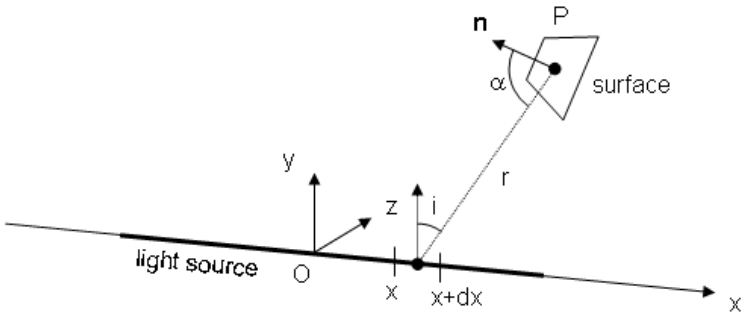


Figure 17: Coordinate system used for lighting computations

In a coordinate system where the light source is centered on the origin, and where the spots are placed on the x axis between $x = -L/2$ and $x = +L/2$, and directed along the y axis (see Figure 17), the light intensity of the spots between x and $x + \mathrm{d}x$ at a point P is given by

$\mathrm{d}i = k\mathrm{d}x \max(0, \cos(i))/r^2$, and the light diffused by the surface at P in any direction is given by $\mathrm{d}l = \mathrm{d}i \max(0, \cos(\alpha))$. This gives:

$$\begin{aligned}
\mathrm{d}i &= k\mathrm{d}x \max\left(0, \frac{p_y}{\sqrt{(p_x-x)^2+p_y^2+p_z^2}}\right) \frac{1}{(p_x-x)^2+p_y^2+p_z^2} \\
\mathrm{d}l &= \mathrm{d}i \max\left(0, \frac{(p_x-x)n_x + p_y n_y + p_z n_z}{\sqrt{(p_x-x)^2+p_y^2+p_z^2}}\right) \\
\mathrm{d}l &= k\mathrm{d}x \max(0, p_y) \frac{\max\left(0, (p_x-x)n_x + p_y n_y + p_z n_z\right)}{\left[(p_x-x)^2+p_y^2+p_z^2\right]^2} \\
l &= \int_{-L/2}^{L/2} \mathrm{d}l = k \max(0, p_y) \int_{-L/2}^{L/2} \frac{\max\left(0, (p_x-x)n_x + p_y n_y + p_z n_z\right)}{\left[(p_x-x)^2+p_y^2+p_z^2\right]^2} \mathrm{d}x \\
l &= k \max(0, p_y) \int_{p_x-L/2}^{p_x+L/2} \frac{\max\left(0, x n_x + p_y n_y + p_z n_z\right)}{\left[x^2+p_y^2+p_z^2\right]^2} \mathrm{d}x
\end{aligned}$$

The max function inside the integral can be removed by reducing the integration interval to x such that $xn_x > -(p_y n_y + p_z n_z)$:

$$l = k \max(0, p_y) \left[n_x \int_{\dots}^{\dots} \frac{x}{\left[x^2+p_y^2+p_z^2\right]^2} dx + (p_y n_y + p_z n_z) \int_{\dots}^{\dots} \frac{1}{\left[x^2+p_y^2+p_z^2\right]^2} \mathrm{d}x \right]$$

and, since the two above integrals can be computed analytically:

$$\int \frac{x}{(x^2+a^2)^2} \mathrm{d}x = -\frac{1}{2}\frac{1}{x^2+a^2} \quad , \quad \int \frac{1}{(x^2+a^2)^2} \mathrm{d}x = \frac{1}{2a^3}\left[\frac{ax}{x^2+a^2} + \arctan\frac{x}{a}\right]$$

it is possible to compute the whole light contribution analytically.

3.2 Athmosphere

The atmospheric scattering (i.e. the reflection and refraction of light on air particles, which is responsible for the blue color of sky - see Figure 18) gives two effects: 1) a photon emitted from an object towards the viewer may be deviated by an air particle between the object and the viewer and 2) a photon emitted from a light source in a different direction than the direction of the viewer, can be deviated towards the viewer by a particle between the viewer and the object ([20, 21]).

Figure 18: Rama with or without atmospheric scattering

If $f(x)$ is the light intensity at x between an object and an observer (see Figure 19), $f(x + \mathrm{d}x)$ is equal to $f(x)$, minus the light that is "absorbed" due to the first effect, plus the light that is "emitted" due to the second effect.

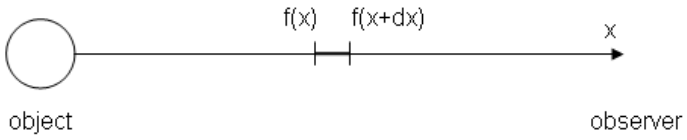


Figure 19: Coordinate system used for atmospheric scattering computations

The absorbed light is proportional to $f(x)$ and to the number of air particules between x and $x + \mathrm{d}x$, while the emitted light is proportional to the number of air particules between x and $x + \mathrm{d}x$. If $\tau(x)$ is the density of air particules at x, this gives:

$$\begin{aligned} f(x+\mathrm{d}x) &= f(x) - f(x)\tau(x)\mathrm{d}x + \tau(x)\mathrm{d}x \\ \frac{f'(x)}{1-f(x)} &= \tau(x) \\ \ln(1-f(x))' &= -\tau(x) \\ \ln(1-f(x)) &= -\int_0^x \tau(y)\mathrm{d}y + \ln(1-f(0)) \\ f(x) &= 1 - e^{-\int_0^x \tau(y)\mathrm{d}y} + f(0)e^{\int_0^x \tau(y)\mathrm{d}y} \end{aligned}$$

We must now compute τ, the density of air particules inside Rama. If we suppose that the atmosphere is at equilibrium and rotates at the same angular speed ω as the spacecraft itself, $\tau(P)$ only depends on r, the distance between P and the spacecraft rotation axis. A small atmospheric volume between r and $r + \mathrm{d}r$ is at equilibrium if the pressure difference between r and $r + \mathrm{d}r$ is equal to the weight of air inside the volume. This gives:

$$\begin{aligned} P(r+\mathrm{d}r)S - P(r)S &= \mu.S\mathrm{d}r.r\omega^2 \\ \mathrm{d}P/\mathrm{d}r &= \mu r\omega^2 \end{aligned}$$

where P is the air pressure and μ its volumetric mass ($r\omega^2$ is the centrifugal force at r). But μ is proportional to P. This comes from the perfect gaz equation, and from the definition of the volumetric mass μ, which is the mass $n.\nu$ (where n is the gaz quantity in mole, and ν the molar mass) divided by the volume V

$$PV = nRT \quad , \quad \mu = \frac{n\nu}{V}$$

If we suppose that T is constant, we get:

$$\begin{aligned} \frac{\mathrm{d}P}{\mathrm{d}r} &= rP\frac{\nu\omega^2}{RT} \\ P(r) &= P(0)e^{\frac{\nu\omega^2}{RT}\frac{r^2}{2}} \end{aligned}$$

Since P, μ and τ are proportional, the equation for μ and τ are similar. The atmospheric scattering effect along a ray $o + t.\mathbf{n}$ parameterized by t is therefore given by:

$$f(t) = 1 - e^{-\int_0^t \tau_0 e^{\frac{\nu\omega^2}{RT}\frac{r(t)^2}{2}}\mathrm{d}t} + f(0)e^{-\int_0^t \tau_0 e^{\frac{\nu\omega^2}{RT}\frac{r(t)^2}{2}}\mathrm{d}t}$$

Since the integrals cannot be computed analytically, the exponentials inside them are approximated with an expression of the form $a.r(t)^4 + b.r(t)^2 + c$, which gives a polynom of degree 4 in t when $r(t) = ||o + t.\mathbf{n}||$ is developped.

Note: with $\nu = 0.0289\ kg.mol^{-1}$ (the molar mass of an atmosphere composed of azote and oxygen in 80%-20% proportions, as on Earth), $\omega = 2\pi/180\ s^{-1}$ (which gives a centrifugal force on the Rama surface equivalent to the gravity on Earth), $T = 293\ K$ (a typical temperature on Earth), and with the constant of the perfect gaz equation $R = 8.314\ J.mol^{-1}$, we get $P(8000m)/P(0) = 1.588$. In other words the pressure, and therefore the air density, is normally quasi constant inside Rama (on Earth $P(5000m)/P(0) = 2$). However I did not use these values, because I found the atmospheric effect was more beautiful with higher pressure variations between the axis and the surface. I used $\nu = 1.0\ kg.mol^{-1}$, which gives $P(8000m)/P(0) = 5$.

3.3 Haloes

In the previous section I implicitely assumed, for the "emission" effect, that the intensity of the light coming from a direction $\mathbf{d}$ was independent of $\mathbf{d}$, which is only valid for ambient light. In order to compute the atmosperic scattering effect due to the six linear lights of Rama (see Figure 20), it is necessary to take $\mathbf{d}$ into account.

Figure 20: Rama with or without anisotropic atmospheric scattering

Lets consider one linear light, along the x axis, and pointed in direction $\mathbf{N}$ (with $N_x = 0$), and a point P on a ray between an object and an observer (see Figure 21).

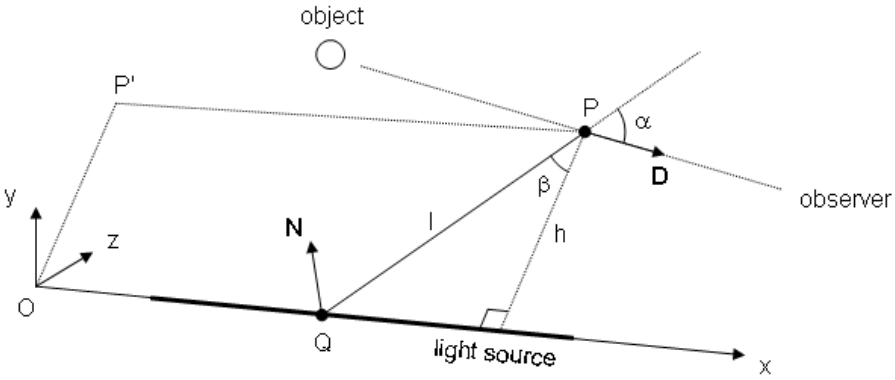


Figure 21: Coordinate system used to compute anisotropic scattering effects

At point P, the intensity of the light emitted from the spots at Q, between x and $x + \mathrm{d}x$, is $\mathrm{d}i = k\mathrm{d}x \max(0, \cos(\mathbf{n}, \mathbf{QP}))/l^2$. If a fraction $\phi(\cos(\alpha))$ of this light is deviated towards the

viewer, the light coming from Q and deviated towards the viewer at P is $\mathrm{d}s = \mathrm{d}i\phi(\cos(\alpha))$.

$$\begin{aligned}
\mathrm{d}s &= k\mathrm{d}x \max\left(0, \cos(\mathbf{n}, \mathbf{QP})\right) \frac{1}{l^2}\phi(\cos\alpha) \\
\mathrm{d}s &= k \max\left(0, \frac{\mathbf{n}.\mathbf{QP}}{l}\right) \frac{1}{l^2}\phi\left(\frac{\mathbf{D}.\mathbf{QP}}{l}\right)\mathrm{d}x \\
\mathrm{d}s &= k \max\left(0, \mathbf{n}.\mathbf{OP}'\right) \left(\frac{\cos\beta}{h}\right)^3 \phi\left(\frac{D_x.QP_x + \mathbf{D}.\mathbf{OP}'}{l}\right)\mathrm{d}x \\
\mathrm{d}s &= k \max\left(0, \mathbf{n}.\mathbf{OP}'\right) \left(\frac{\cos\beta}{h}\right)^3 \phi\left(D_x \sin\beta + \frac{\mathbf{D}.\mathbf{OP}'}{h}\cos\beta\right)\mathrm{d}x
\end{aligned}$$

lets define K, u and v as follows. With the relations between $x = Q_x$ and β

$$\begin{aligned}
K &= k\max(0, \mathbf{n}.\mathbf{OP}') \\
u &= \mathbf{D}.\mathbf{OP}'/h \\
v &= D_x \\
\tan\beta &= \frac{p_x - x}{h} \\
\frac{\mathrm{d}\beta}{\cos^2\beta} &= -\frac{\mathrm{d}x}{h}
\end{aligned}$$

we get

$$\begin{aligned}
\mathrm{d}s &= K\left(\frac{\cos\beta}{h}\right)^3 \phi(u\cos\beta + v\sin\beta)\frac{-h}{\cos^2\beta}\mathrm{d}\beta \\
\mathrm{d}s &= -\frac{K}{h^2}\cos\beta\phi(u\cos\beta + v\sin\beta)\mathrm{d}\beta
\end{aligned}$$

The light deviated at P towards the viewer is partially attenuated, due to the "ambient" atmospheric scattering. Moreover, ϕ is proportional to the number of air particules between P and $P + \mathrm{d}P$. We can then compute the light emitted at Q, deviated at P, and that reach the observer and, finally, the total scattering effect (we define t as the distance betwen P and the observer):

$$\begin{aligned}
\mathrm{d}l &= -\frac{K}{h^2}\cos\beta\tau_0 e^{\frac{\nu\omega^2}{RT}\frac{r(t)^2}{2}}\Phi(u\cos\beta + v\sin\beta)e^{-\int_0^t \tau_0 e^{\frac{\nu\omega^2}{RT}\frac{r(u)^2}{2}}\mathrm{d}u}\mathrm{d}t\mathrm{d}\beta \\
l &= -\int_0^T \frac{\tau_0 K}{h^2}\left(\int \cos\beta\Phi(u\cos\beta + v\sin\beta)\mathrm{d}\beta\right) e^{\frac{\nu\omega^2}{RT}\frac{r(t)^2}{2}} e^{-\int_0^t \tau_0 e^{\frac{\nu\omega^2}{RT}\frac{r(u)^2}{2}}\mathrm{d}u}\mathrm{d}t
\end{aligned}$$

This double integral cannot be computed analytically, and therefore requires a **lot** of numerical computations. Hopefully the inner integral can be precomputed, for various values of u, v and β, which can vary between -1 and 1, or -$\pi/2$ and $\pi/2$. This reduces the number of operations needed to compute the halo effect, but these computations still take a long time.

Note: for the phase function Φ I used $[(1 + \cos(\alpha))/2]^{20}$, which gives a strong forward scattering. In order to get more aesthetic results I also used a different density function than the air density function computed in section 3.2. In fact I used a function of the form $a.r^4 + b$ (this is possible, if the halo effect is seen as being due to the scattering of light due to water particles: the density function must then be the density of water particles, instead of the density of air particles). In summary, I used $\phi = (a.r^4 + b)\,[(1 + \cos(\alpha))/2]^{20}$.

3.4 Shadows

The preferred way to render shadows with RenderMan is to use shadow maps, i.e. depth maps computed from the view point of a light. Unfortunately these shadow maps are well adapted for spot lights, but not for distant lights or area lights (and the lights in Rama are *both* distant and non punctual). I therefore had to use the alternative method, i.e. ray traced shadows, which are much slower to compute, especially with area lights: with 6 linear lights, and at least 10 rays per light to get smooth shadows, the number of shadow rays is very high.

Figure 22: The Rama south pole with or without shadows

For a static scene, like Rama, the main disadvantage of using ray traced shadows, compared to shadow maps, is that shadows must be recomputed for each picture, while shadow maps are computed once and for all. A natural question was then: is it possible to compute some or all of the ray traced shadows once and for all?

The answer is yes for some shadows: the idea is to compute six maps, one per light, that give the percentage of light received at any given point at the surface of Rama. Since the size of these maps is limited by memory and disk space, their resolution is limited to something like 10 m (which gives 3200x5000 pixels for the plains and the cylindrical sea). They can therefore not accurately represent the shadows of objects whose size is less than 50 to 100 meters, like buildings or trees, but they can represent the shadows of clouds, of the south horns, or the shadows of hills or mountains.

For Rama the shadows are therefore computed in the following way:

- the terrain, horn and cloud shadows are ray traced, but computed once and for all in "ray traced shadow maps"
- the building shadows are ray traced, and recomputed for each picture
- the self shadowing of trees is simulated with shader tricks (there are far too many trees to ray trace these shadows)
- the shadows of trees on the terrain are not computed at all.

The ray traced shadow maps are computed in the following way:

- for the south pole shadows, a ray is cast for each point between $y = -8000\ m$ and $y = +8000\ m$, and $z = -8000\ m$ and $z = +8000\ m$, in the direction of the x axis, in order to find the position of the Rama surface at this position. Then many rays are cast from this computed position towards equally spaced points of one of the linear light. The percentage of rays that intersect an object gives the shadow intensity.

- for the south and north plain shadows, a ray is cast for each point between $x = -16000\ m$ and $x = +16000\ m$, in all radial directions, in order to find the position of the Rama surface for each x and each radial direction. Then many rays are cast from these computed positions towards equally spaced points of one of the linear light. The percentage of rays that intersect an object gives the shadow intensity.

In other words the south pole shadow maps are computed with a planar projection, while the south and north plain shadow maps are computed with a cylindrical projection. The result for one of the six lights is shown in Figure 23 (the right map shows, from left to right, the unrolled south plain, the unrolled cylindrical sea and the unrolled north plain; note that there are ten more maps for the five other lights).

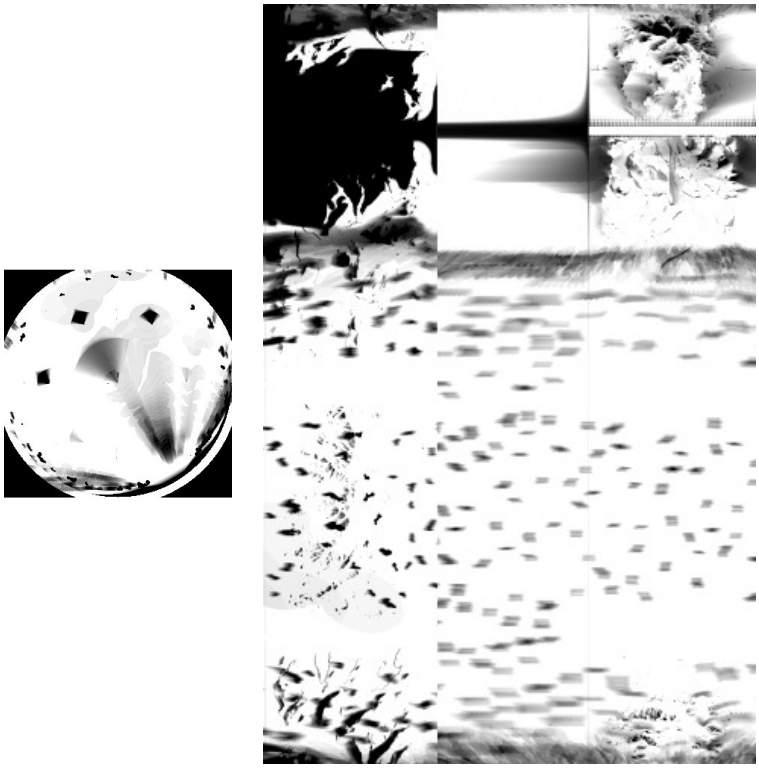


Figure 23: The ray traced shadow maps used for Rama

In order to compute the contribution of a linear light to the color of a surface at a given point P, while taking shadows into account, an approximation is used: instead of integrating the light contribution over all points of the light source that are visible from P, the integration is done over all points (visible or not), and the result is multiplied by the percentage of these points that are visible. This allows the integration to be performed analytically, as explained in section 3.1, and the percentage to be precomputed.

4 Texturing and shading

4.1 Terrain textures

The terrain textures are *automatically generated* from the 2D vectorial maps. The basic idea is to draw or fill the curves defined in these maps to obtain an image which is stored as a texture. More precisely, the terrain textures are made of several independent monochrome channels, each channel corresponding to a separate map (one channel corresponds to the roads, another to the rivers, another to the forests, and so on). The channels are shaded and composited together at render time, whith a Renderman shader.

While the texture channel for rivers or forests is easy to draw, the texture channel for roads is more complicated. In fact two channels are used: one for the road itself, and another for the marks on the roads. The first one is simple to draw, but the second one is more complicated, especially at road crossings (see Figures 24 and 25).

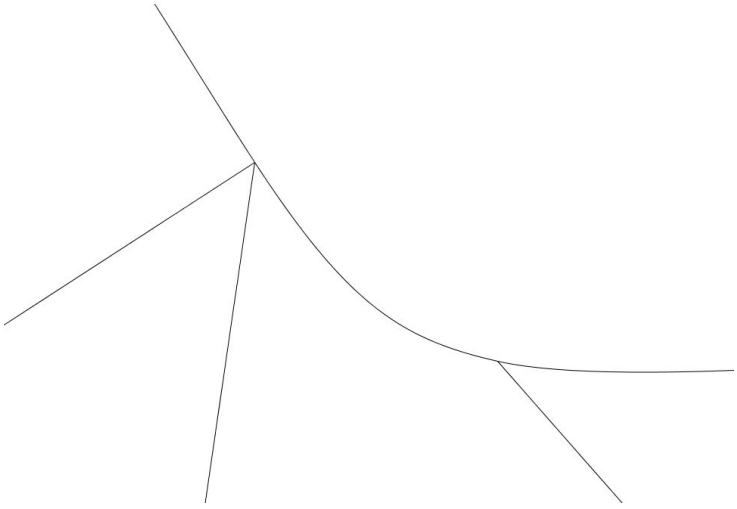


Figure 24: Road graph, made of cubic Bezier curves (with an associated width attribute)

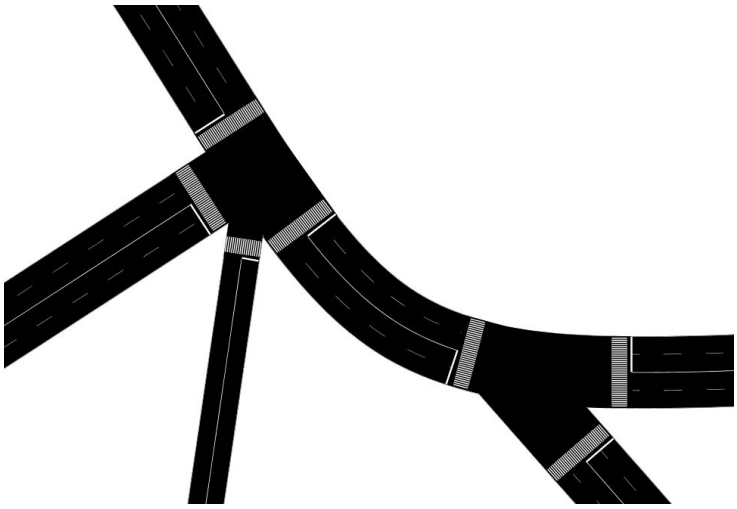


Figure 25: Road texture generated from the graph of Figure 24

Like for the terrain mesh, the terrain texture is generated with a variable level of detail: distant zones are textured with low resolution textures, while the zone near the camera is textured with a high resolution texture (see Figure 26). The resolution to be used is automatically computed from the image size, the camera aperture and the distance from the camera (each time the distance doubles, the texture resolution is divided by two).

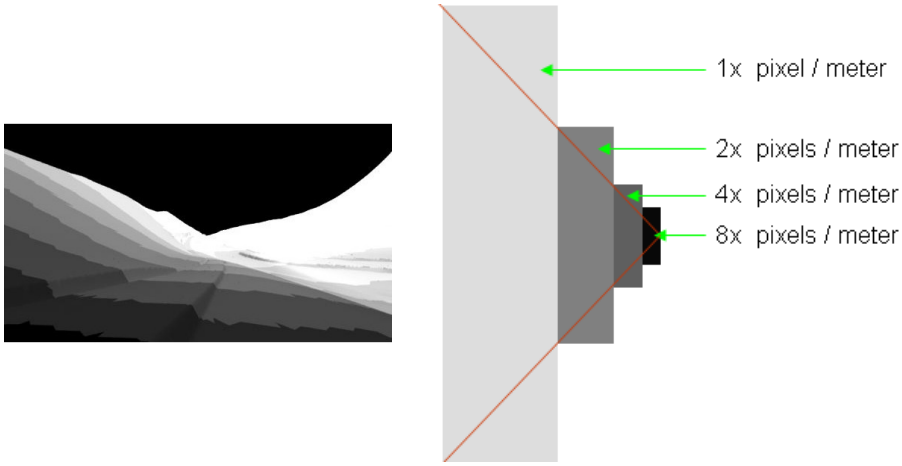


Figure 26: Terrains are textured with several textures at different resolutions

Note that since the 2D maps are vectorial, it is easy to get good images of them at any resolution (this is why I choose to use vectorial maps; this also explains the use of Bezier curves instead of polylines). The textures are drawn with the Libart library [15], which has native support for drawing cubic Bezier curves.

4.2 Terrain shader

As explained above, the terrain shader shades each layer and composites them together to get the final terrain color (see Figure 27).

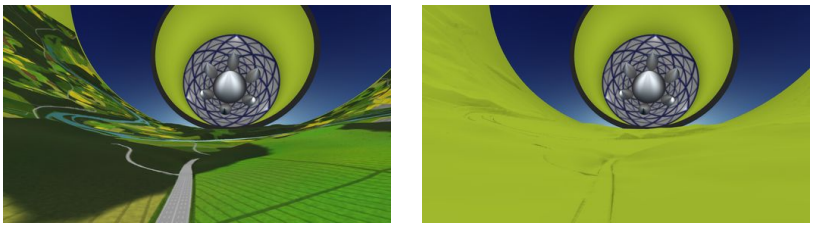


Figure 27: Rama with or without the terrain shader

Most layers are shaded with a very simple function using either a constant color or a simple color noise to introduce small variations in hue, saturation and luminosity:

```
color colorNoise (point P; float fw; uniform float freq, octaves) {
    return 1 + color vfBm(P*freq, fw*freq, octaves, 2, 1) * (0.025, 0.025, 0.1);
}
color colorProd (color C; color Cnoise) {
    return ctransform("hsl", "rgb", ctransform("hsl", C) * Cnoise);
}
```

```
color roadMarkColor (uniform float Ka, Kd; color Ca, Cd) {
    return ROAD_MARK_COLOR * (Ka * Ca + Kd * Cd);
}
color grassColor (uniform float Ka, Kd; color Ca, Cd, Cnoise) {
    return colorProd(GRASS_COLOR, Cnoise) * (Ka * Ca + Kd * Cd);
}
```

The most complex layer shaders are the field layer shader, which composites various field textures together, and the water layer shader, which uses perturbed normals and ray tracing to compute reflections:

```
color waterColor (
  uniform float Ka, Kd, Ks; color Ca, Cd; point P, Pw; vector V, Vw; float alpha)
{
    extern vector I;
    extern float time;
    point Q = P;
    point Qw = Pw;
    color C, Ct, Cr = 0;
    float Kr = 0;
    float Kt = 0;

    if (alpha > 0) {
        normal Ns = normal V;
        Qw = point (xcomp(Pw), 0, 0) + 8000 * Vw;
        Q = transform("world", "current", Qw);

        Q += 0.1 * ridgedNoise(Qw * 0.15, time, filterwidthp(Qw) * 0.15, 4, 2, 1) * Ns;
        Ns = normalize(calculatenormal(Q));

        fresnel(I, Ns, 1/1.33, Kr, Kt);

        Ct = WATER_COLOR * (Ka * Ca + Kd * Cd);
        Cr = trace(P, normalize(reflect(I, Ns)), 1);
        C = (1 - Kr) * Ct + Kr * Ks * Cr;
    }

    return C;
}
```

The terrain shader just calls the various layer shading functions and composes their results according to the layer opacities given by the texture maps:

```
color terrainColor (uniform float Ka, Kd, Ks; color Ca, Cd, C1, C2) {
    extern point P;
    extern normal N;

    // point in world coordinates, corresponding filterwidth
    point Pw = transform("world", P);
    float fw = filterwidthp(Pw);

    // vertical vector in world and current coordinates
    vector Vw = normalize(vector (0, -ycomp(Pw), -zcomp(Pw)));
    vector V  = vtransform("world", "current", Vw);

    // layer opacities
    float road     = comp(C1, 0);
    float roadMark = comp(C1, 1);
```

```
    float water    = comp(C1, 2);
    float forest   = comp(C2, 0);
    float field    = comp(C2, 1);
    float grass    = comp(C2, 2);

    // color noises
    color Cnoise1 = 1 + color vfBm(Pw*0.005, fw*0.005, 4, 2, 0.5) * (0.15, 0.2, 0.1);
    color Cnoise2 = 1 + color vfBm(Pw*0.25, fw*0.25, 6, 2, 1) * (0.025, 0.025, 0.1);

    // compositing
    color C = 0;
    float clip = 1;
    C += clip * roadMark * roadMarkColor(Ka, Kd, Ca, Cd);
    clip = clip * (1 - roadMark);
    C += clip * road * roadColor(Ka, Kd, Ca, Cd);
    clip = clip * (1 - road);
    C += clip * water * waterColor(Ka, Kd, Ks, Ca, Cd, P, Pw, V, Vw, clip * water);
    clip = clip * (1 - water);
    C += clip * forest * forestColor(Ka, Kd, Ca, Cd, Cnoise1 * Cnoise2);
    clip = clip * (1 - forest);
    C += clip * grass * grassColor(Ka, Kd, Ca, Cd, Cnoise1 * Cnoise2);
    clip = clip * (1 - grass);
    C += clip * fieldColor(Ka, Kd, Ca, Cd, Cnoise1, Pw, fw, field);

    return C;
}
```

4.3 Building shader

The building shader is based on a basic window pattern stored in a texture file:

```
color buildingWallColor (
  uniform float Ka, Kd, Ks; color Ca, Cd; color Cbuilding; normal Nn;
  float seed; float ss, tt; float u0, u1, v0, v1)
{
    extern point P;
    extern vector I;
    extern float du, dv;

    float fw = filterwidth(ss);

    // basic pattern
    float windowClip = float texture("window3.tdl", ss, 1 - tt);
```

In order to avoid truncated windows at wall boundaries, this pattern is modified based on the parametric coordinates `ss,tt` of the current point, and on the parametric coordinates `u0,u1,v0,v1` of the wall corners (which are computed during the building generation process, and stored as Renderman attributes):

```
    float um = ceil(u0/2);
    if (um - u0/2 < 0.5) {
        um += 1;
    }
    float uM = floor(u1/2);
    if (u1/2 - uM < 0.5) {
        uM -= 1;
    }
```

```
float boundsClip = filteredpulse(um, uM, ss, fw);
windowClip *= boundsClip;
```

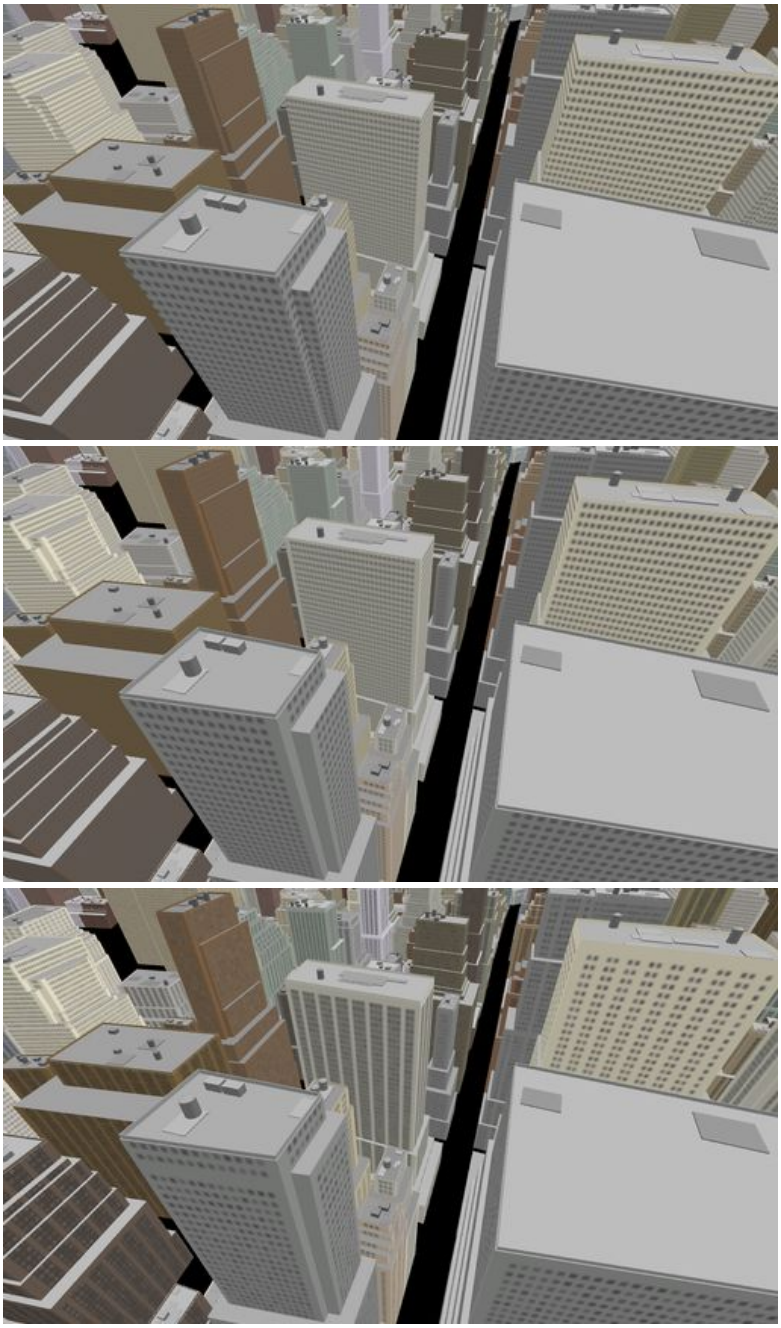


Figure 28: The building shader (top: basic, middle: no truncated windows, bottom: final)

In order to introduce variations between buildings, the pattern is modified based on the "identifier" or "seed" that was generated for each building and stored as a Renderman attribute. This seed is used to introduce vertical stripes without windows (with a variable period), and to add optional darker stripes around windows (see Figure 28). Finally, in order to introduce variations between the windows of a single building, a cellnoise is used:

```
// vertical stripes without windows
float stripesClip = 0;
```

```
    float period = mod(round(seed/79), 4);
    if (period > 0 && u1-u0 > 2*period) {
        stripesClip = filteredpulsetrain(1, period + 1, ss, fw);
        windowClip *= stripesClip;
    }

    // removes windows at the last floor (if at least 10 floors)
    float topClip = 1;
    if (v1 >= v0 + 30) {
        topClip *= 1 - step(v1/3 - 1, tt);
        windowClip *= topClip;
    }

    // optional vertical, darker stripes around windows
    float stripe = mod(seed, 10);
    if (stripe < 3) {
        stripesClip = 0;
    } else {
        stripesClip = stripesClip*boundsClip*topClip;
    }

    // ambient and diffuse light, layer colors
    color Cad = (Ka * Ca + Kd * Cd);
    color Cstripe = Cbuilding * 0.75;
    color Cwindow = 0.45 + 0.2 * float cellnoise(ss, tt);

    // compositing
    color C = 0;
    float clip = 1;
    C = clip * windowClip * Cwindow * Cad;
    clip = clip * (1 - windowClip);
    C += clip * stripesClip * Cstripe * Cad;
    clip = clip * (1 - stripesClip);
    C += clip * Cbuilding * Cad;

    return C;
}
```

4.4 Tree shader

The tree shader uses some tricks to simulate shadows and ambient occlusion. Shadows are simulated by multiplying the tree color by `1-I.N`, where `I` is the incident light vector, and `N` the normal to the overall tree shape (this normal is computed based on some predefined tree shapes which are hardcoded in the tree shader). The effect of this factor is to darken the interior of the tree contour, as if the tree was lit from behind. Ambient occlusion is simulated by multiplying the tree color by `z/zMax`, where `z` is the height of the shaded point, and `zMax` the height of the tree. The effect of this second factor is to darken the base of the tree (in a forest, due to ambient occlusion, there is less light at the ground level than at the top of trees):

```
surface tree (float Ka = 0.3, Kd = 0.7, Ks = 0; float treekind = -1) {
    normal Nf = faceforward(normalize(N), I);
    point Po = transform("object", P);
    vector Io = vtransform("object", I);
    float k = 1;

    if (treekind == 1) {
        float h = zcomp(Po) / 10;
        normal No = (xcomp(Po)*2.63, ycomp(Po)/2.63, zcomp(Po)*2.63);
```

```
        k = 1 - abs(normalize(Io).normalize(No));
        k = pow(k, 0.3) * pow(h, 3); // shadows factor * ambient occlusion factor
    }
    // ...

    k = sqrt(k < 0 ? 0 : k);
    Ci = Cs * (Ka * ambient() + Kd * k * (diffuse(Nf, 0) + 0.8*diffuse(-Nf, 0)));
}
```

Both factors increase the contrast between trees (see Figure 29).

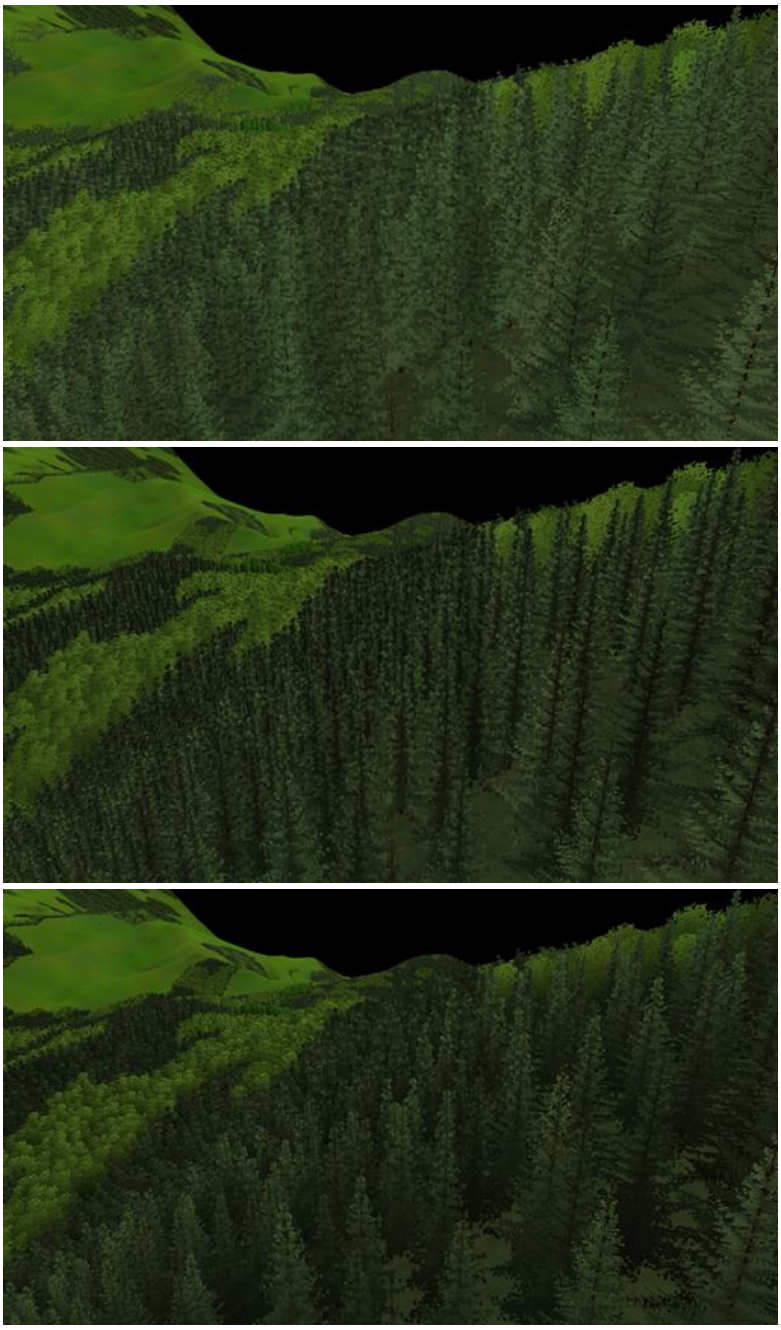


Figure 29: The tree shader (top: basic shader, middle: shadow factor, bottom: both factors)

4.5 Cloud shader

During the cloud generation process, after all could particules have been generated, a lighting computation phase is performed in order to compute the amount of light received by each particle, while taking into account multiple scattering effects. The implementation of this lighting computation phase is based on the algorithm described by S. Harris [19]. The result is a `lighting` attribute associated with each cloud particule. The cloud shader shades each particule with a transparency effect, and with a light intensity proportional to `lighting`.

```
surface cloud (float Ka = 0.5, Kd = 0.5, luminosity = 1, lighting = 1) {
    vector In = normalize(I);
    float cos = (N.In) / length(N);
    if (cos < 0) {
        point Pw = transform("world", P);
        float r = sqrt(ycomp(Pw)*ycomp(Pw) + zcomp(Pw)*zcomp(Pw));
        vector V = vector "world" (0, -ycomp(Pw)/r, -zcomp(Pw)/r); // vertical vector
        float ctheta = V.In;
        float rayleigh = 0.75 * (1 + ctheta*ctheta);
        Oi = 0.4 * hermite(sqrt(1 - cos*cos)); // hermite(u) = (2*u - 3)*u*u + 1
        Ci = (Ka + Kd * rayleigh * lighting * luminosity) * ambient() * Oi;
    } else {
        Oi = 0; Ci = 0;
    }
}
```

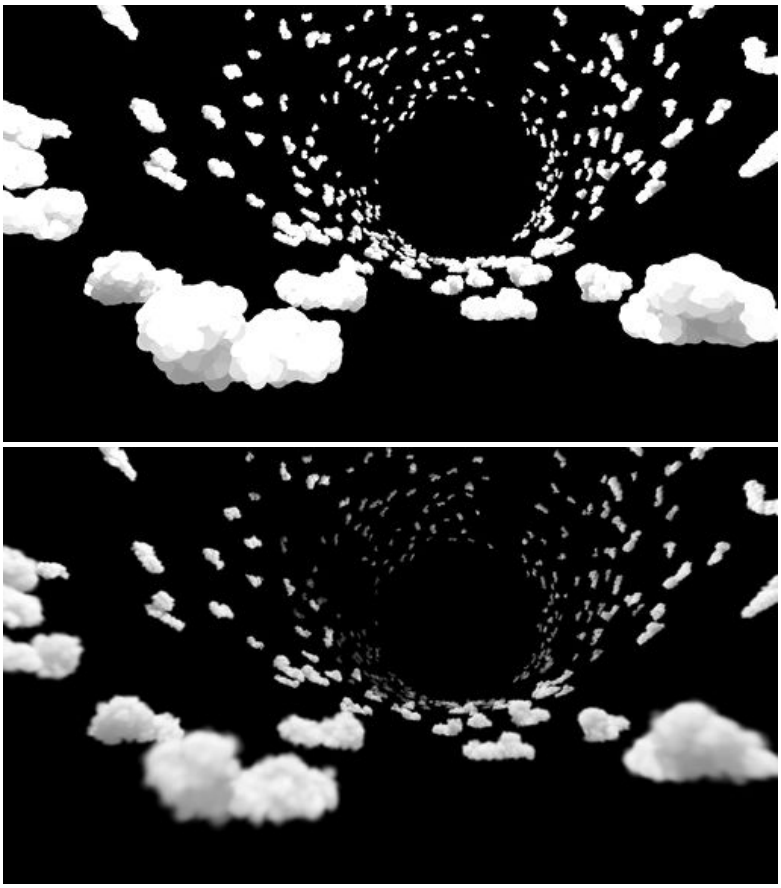


Figure 30: The cloud shader (top: no transparency, bottom: with transparency)

5 Perspectives

Many improvements are possible, both in the model generator and in the shaders. Here is a non exhaustive list:

- add a generator to generate wheat, mais, or grass blades in fields, with automatic handling of level of details.
- generate more realistic buildings, by using algorithms such as those described by Wonka et al. [13].
- generate cars on roads, human beings in cities, animals in fields, ...
- improve shaders to render the spacecraft (and especialy the cities) at night, or during winter or other seasons.

References

[1] A.C. Clarke, *Rendez vous with Rama*, 1973.

[2] *The RenderMan Interface*, Pixar, 2000.

[3] A.A. Apodaca, L. Gritz, *Advanced RenderMan*, Morgan Kaufman, 2000.

[4] *3Delight*.
`http://www.3delight.com`

[5] *ATDI*.
`http://www.atdi-us.com/SDTS_DL_b.htm`

[6] *Census 2000 Tiger Line Data*.
`http://www.esri.com/data/download/census2000_tigerline/`

[7] *TerraServer*.
`http://terraserver.microsoft.com/`

[8] W. Evans, D. Kirkpatrick, G. Townsend, *Right Triangulated Irregular Networks*, Algorithmica: Special Issue on Algorithms for Geographical Information, 30(2), p. 264-286, 2001.

[9] W. Evans, G. Townsend, *Topovista*.
`http://www.cs.arizona.edu/topovista/`

[10] J.R. Shewchuk, *Triangle*.
`http://www-2.cs.cmu.edu/~quake/triangle.html`

[11] Y.I.H. Parish, P. Muller, *Procedural Modeling of Cities*, SIGGRAPH, 2001.

[12] S. Greuter, J. Parker, N. Stewart, G. Leach, *Real-time procedural generation of 'pseudo infinite' cities*, GRAPHITE, 2003.

[13] P. Wonka, M. Wimmer, F. Sillon, W. Ribarsky, *Instant Architecture*, SIGGRAPH, 2003.

[14] A. Murta, *General Polygon Clipper*.
`http://www.cs.man.ac.uk/~toby/alan/software/`

[15] R. Levien, *Libart*.
`http://www.levien.com/libart/`

[16] J. Weber, J. Penn, *Creation and Rendering of Realistic Trees*, SIGGRAPH, 1995.

[17] W. Diestel, *Arbaro*.
`http://arbaro.sourceforge.net/`

[18] A. Bouthors, F. Neyret, *Modeling clouds shape*, EUROGRAPHICS, 2004.

[19] M. J. Harris, A. Lastra, *Real-Time Cloud Rendering*, EUROGRAPHICS, 2001.

[20] D.S. Ebert, *Texturing and Modeling: a procedural approach*, Morgan Kaufmann, 2003.

[21] A. J. Preetham, *A Practical Analytic Model for Daylight*, International Conference on Computer Graphics and Interactive Techniques archive, p. 91-100, 1999.

A rama.rib

```
AttributeBegin
  Color 0.415 0.48 0.117
  Surface "basic"
  Rotate 90 0 1 0

  # north plain (coordinates are in meters)
  AttributeBegin
    Surface "northc"
    Cylinder 8000 5000 16000 360
  AttributeEnd

  # south plain
  AttributeBegin
    Surface "southc"
    Cylinder 7500 -5000 -16500 360
  AttributeEnd

  # cylindrical sea
  AttributeBegin
    Surface "sea"
    Cylinder 8010 -5000 5000 360
  AttributeEnd

  # south cylindrical sea border
  AttributeBegin
    Color 0.3 0.3 0.3

    Translate 0 0 -5000
    NuPatch
      9 3 [0 0 0 0.25 0.25 0.5 0.5 0.75 0.75 1 1 1 ] 0 1
      2 2 [0 0 1 1 ] 0 1
      "Pw" [
      7500  0     0 1
      5303  5303  0 0.707107
      0     7500  0 1
      -5303 5303  0 0.707107
      -7500 0     0 1
      -5303 -5303 0 0.707107
      0     -7500 0 1
      5303  -5303 0 0.707107
      7500  0     0 1
      8010  0     0 1
      5664  5664  0 0.707107
      0     8010  0 1
      -5664 5664  0 0.707107
      -8010 0     0 1
      -5664 -5664 0 0.707107
      0     -8010 0 1
      5664  -5664 0 0.707107
      8010  0     0 1
      ]
  AttributeEnd

  # north cylindrical sea border
  AttributeBegin
```

```
    Color 0.3 0.3 0.3

    Translate 0 0 5000
    NuPatch
      9 3 [0 0 0 0.25 0.25 0.5 0.5 0.75 0.75 1 1 1 ] 0 1
      2 2 [0 0 1 1 ] 0 1
      "Pw" [
      8000  0     0 1
      5656  5656  0 0.707107
      0     8000  0 1
      -5656 5656  0 0.707107
      -8000 0     0 1
      -5656 -5656 0 0.707107
      0     -8000 0 1
      5656  -5656 0 0.707107
      8000  0     0 1
      8010  0     0 1
      5664  5664  0 0.707107
      0     8010  0 1
      -5664 5664  0 0.707107
      -8010 0     0 1
      -5664 -5664 0 0.707107
      0     -8010 0 1
      5664  -5664 0 0.707107
      8010  0     0 1
      ]
  AttributeEnd

  # north pole
  AttributeBegin
    Color 0.4 0.4 0.4
    Surface "northp"

    Rotate -90 0 0 1
    Translate 0 0 16000
    Sphere 8000 0 8000 360
  AttributeEnd

  # south pole
  AttributeBegin
    Color 0.4 0.4 0.4
    Surface "southp"

    Translate 0 0 -16500
    Rotate 180 1 0 0
    Cone 7500 7500 360
  AttributeEnd
AttributeEnd

# horns
AttributeBegin
  Color 0.4 0.4 0.4
  Surface "horn"
  Rotate 90 0 1 0

  AttributeBegin
    Translate 0 0 -24000
    Cone 12000 4000 360
```

```
  AttributeEnd
  AttributeBegin
    Translate 0 0 -21000
    Rotate 30 0 0 1
    AttributeBegin
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
    AttributeBegin
      Rotate 60 0 0 1
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
    AttributeBegin
      Rotate 120 0 0 1
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
    AttributeBegin
      Rotate 180 0 0 1
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
    AttributeBegin
      Rotate 240 0 0 1
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
    AttributeBegin
      Rotate 300 0 0 1
      Translate 4500 0 0
      Cone 5000 1500 360
    AttributeEnd
  AttributeEnd
AttributeEnd
```