Slartibartfast - code documentation

Martin Kahoun

May 14, 2010

Contents

List of Figures

1 Introduction

Slartibartfast is cross-platform procedural planetary body generator written in `C++` using `OpenGL`. It's a demonstration of what can be done with several numbers, Perlin noise and simple fractal function. It basically operates in a way that user supplies planet parameters such as radius, maximum height, random seed plus several others that influence output of the Perlin noise and the program will generate a planetary body. Rotating with the planet, zooming and changing the way how it's rendered is available. Rendering ranges from simple wireframe up to texture blending provided by `GLSL` shaders.

Slartibartfast allows saving any generated planet as a simple text file containing parameters used for its generation, so any planet can re-generated later. Apart from this, *Slartibartfast* also provides a functionality for exporting the planet into .obj model format for further usage outside this demo.

All source files are located within the `src` directory and all headers are located in `include` directory. Each file corresponds with class of the same name. Some classes use only header files, but most of them have larger methods defined separately in source files.

2 Work flow

In this section we'll take general tour around the code and show what they are doing. Implementation details are listed in the next section. Figure 1 shows us basic work flow of the program. The main function obtains an instance of `Core` class and calls its `Init()` method. If everything is set up correctly the control goes into `Run()` method that implements our main loop. This method also makes sure that all frames are calculated in fixed time steps and thus the execution should appear at the same speed on all machines that can run this program. When the control returns to `main()` we hand it over to `ShutDown()` which will clean all dynamic objects. Then the program terminates.

2.1 Initialization

First thing we do is calling `loadConfig()` method that tries to open and parse configuration file. Whether this method fails or not doesn't bother us, because either way we'll have valid configuration parameters (in the case non-existent/corrupted configuration file or errors in the file all or wrong options are set to default and then written back before the application exits).

Then we need to initialize the `Allegro` library. This is done through some standard API functions. We prepare keyboard and mouse input. At this point we try to set up graphics window. If this fails initialization is terminated immediately and program then exits. Notion of what exactly went wrong should be indicated inside the log file. From this point onward

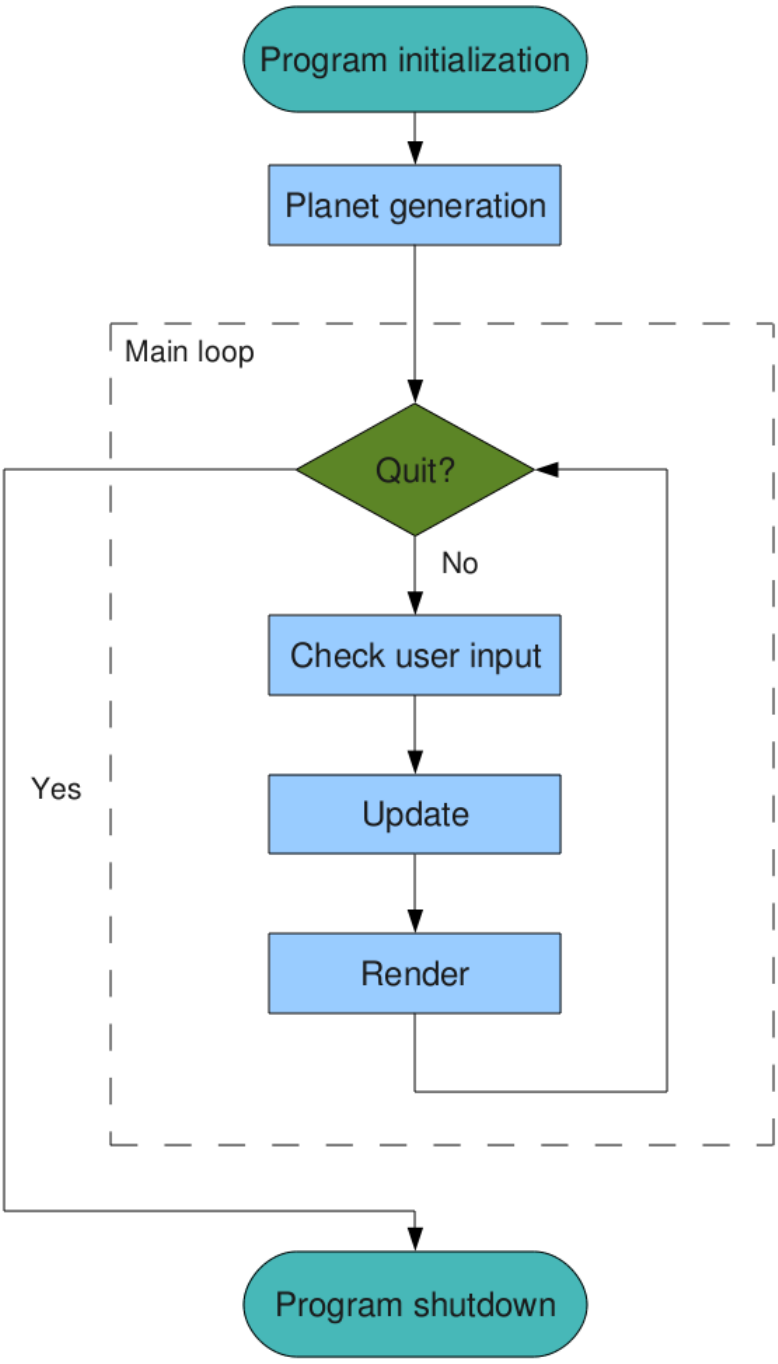

Figure 1: Flowchart

we should be looking into black window have configured projections matrices, lights and vertex buffer should be ready to use (it loads textures and compiles shaders – if a failure occurs it logs it, but doesn't terminate).

Next on the list is planet initialization. We create initial planet with predefined stats and randomly selected seed. More about planet initialization in section 3.8. This is the longest process of the initialization phase and its length depends on how large cube map texture has been asked for. User can notice lag, but shouldn't be worried.In the end we prepare cameras and console. If everything went smooth we are about to enter the main loop.

2.2 Planet generation

This is the most lengthy process of the initialization phase. It's length depends on how big cube map texture needs to be generated. During planet generation we initialize world generator that is implementing Perlin noise. Then we proceed to generate six textures to serve as faces of unit cube used for cube mapping. Time spent here is $O(6 * n^2)$ where n is texture size.

2.3 Main loop

Upon entry into the main loop we call `Allegro` function to initialize timer. It sets another thread that periodically calls small user defined function 60 times per second. Inside this function we increment two helper variables. One for frame synchronization and one for FPS counter.

Inside this loop we check for FPS counter helper variable and if it has been incremented 60 times we update our FPS counter which is implemented as simple circle buffer. Each second we count how many frames have been rendered and calculate an average from it. We also check whether the inner state hasn't changed to exit.

Then there's a nested loop testing against volatile variable that is incremented by our first timer. Until the variable is larger than 0 it calculates new frames. Upon breaking from the nested loop we do all rendering and we do it until the control variable is again larger 0. After each cycle inside the nested loop we decrement the control variable and check for being larger than 1 – that would mean our code takes longer than $1/60s$ to execute and we need to drop frames.

When the rendering is done we give back some time to CPU if the user hasn't specified he wants the maximal performance. This whole method of loop update is for example default in `XNA Game Framework for .NET`.

2.3.1 User input

First thing to do in the main loop is to check for general user input i.e. control keys. We basically check the state of the keys we are interested in and execute the logic behind them – like swapping the camera, adjusting

`OpenGL` parameters, switching to console or toggling the flag to quit the application.

2.3.2 Update

All main logic is executed here. Depending on the inner state of the application we either update planet object, camera and lighting or we update our system console. During this phase the `VertexBuffer` class is filled with newly created vertices and old vertices are removed from it.

Should the user issued planet loading or generating completely new planet, we stop execution, return back to the planet generation phase, wait until it ends and then resume execution with newly created planetary body.

2.3.3 Render

We use visitor pattern to render the planetary object. So what we do is to call planet render method with appropriate visitor to fill `VertexBuffer` with triangles to be rendered. Then we push data to the GPU – because of the nature of our data that change from frame to frame, we can't use VBO[1, p. 93] hence we use only vertex and index arrays, but we hide them inside `VertexBuffer` class – and then we render them.

3 Class overview

Slartibartfast is implemented in the OO fashion. Therefore it is divided into several objects that interact with each other as shown in figure 2. Lines between classes show logical interactions between them, inheritance and instantiation. For more details skip directly to appropriate subsection.

All blue classes are singletons and serve mainly as utility classes for this demo i.e. they implement setting up graphics window, user control and low level rendering (physical vertex pushing through the pipeline). All light-green classes are abstract interfaces from which we derived actual implementation of these components. Green classes are all implemented.

All green classes except the `Camera` class compose the main kernel of the demo – the actual algorithm for generating a planetary body. Should anybody decide to use this algorithm in their work all he needs to do is taking these classes and importing them into their project. Small changes will be necessary as implied by the diagram. These include:

- Method for creating and removing vertices – right now this is done via `VertexBuffer` class from `ROAM_Triangle` objects. Vertex is represented by structure not shown on the diagram, it encompasses things like position, normal, colour etc.

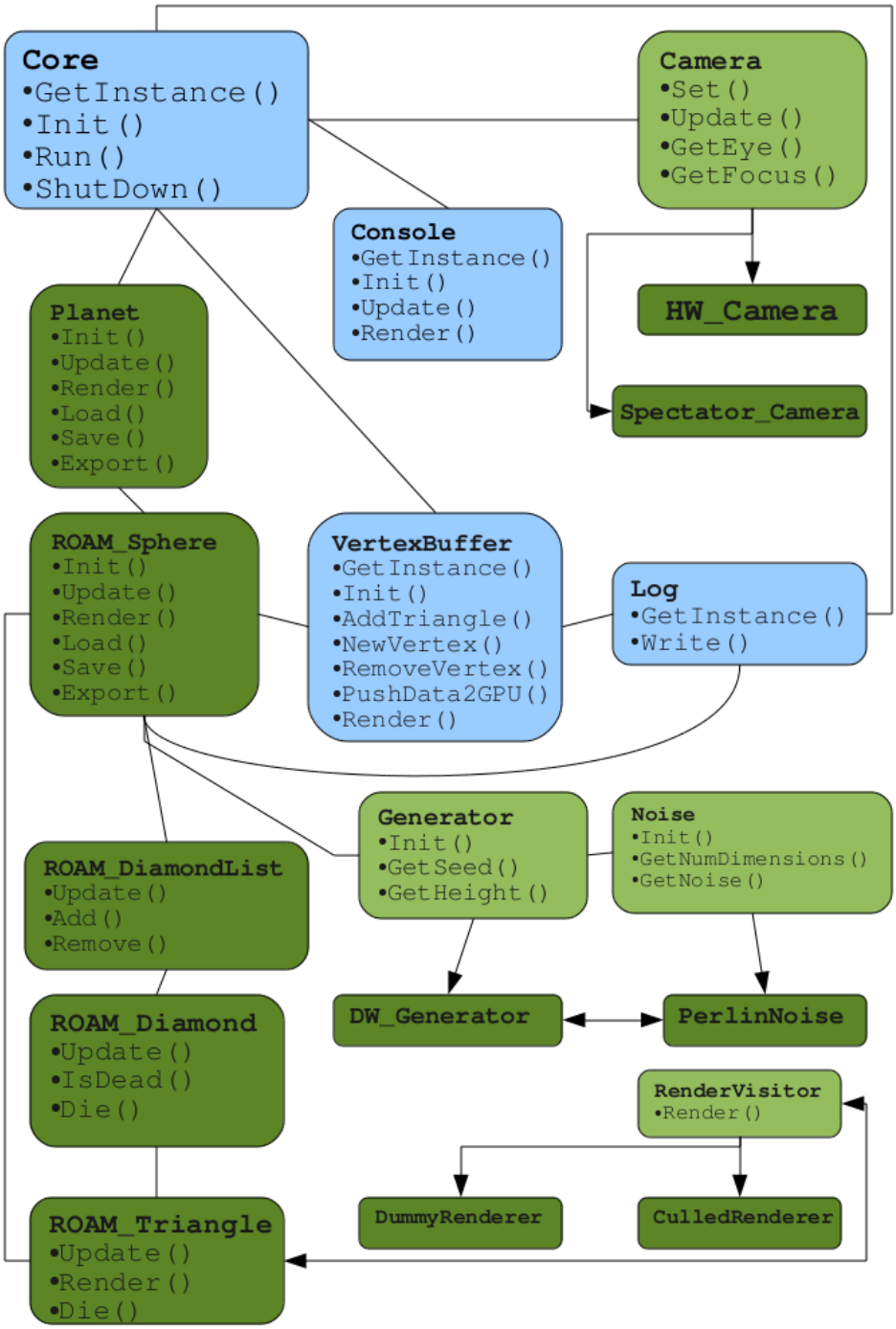

Figure 2: Class diagram

- Physical rendering of the planet – this is actually very easy since virtually everything needed to achieve this, is deriving new class from `RenderVisitor` and writing new code into its `Render()` method. Render visitors are instantiated from the `Planet` class. Then are passed to `ROAM_Sphere.Render()` method (not shown in the diagram) and propagated all the way down to the `ROAM_Triangle.Render()` method which serves as acceptor.

- Passing through camera position – right now `Core` class passes down active camera object so the algorithm knows where is the eye and where is it looking. It uses these information to decide where it should increase visible details and where it isn't necessary.

- Omitting or changing log info – currently

Remainder of this section describes all important classes and their inner logic. Most of the time the inner workings will be described in general fashion since the implementation details should be obvious from the code and in-code comments, however the core algorithm and data structures will be described in-depth.

3.1 Core

`Core` class encompasses most of the boring yet necessary code. It is a singleton that provides interface for initializing, running and shutting down the graphics engine. If a strict OO design would be followed it would also provide interface for controlling the planetary objects. But being this a demo application a decision to omit such proxy methods has been taken and the class is friend with the `Console` class. That means that the console can access all data members.

Class declaration is inside `core.h` header along with some data types definitions. These consist of aggregate types encompassing program configuration and lighting. `Core` implements small finite state machine that decides what to do. Possible states are:

- Init – default state during which we initialize everything. Should this state occur inside the main loop it is perceived as an error that is reported to the system log and then the application terminates.

- Run – this is the state in which we update planet level of detail and camera position.

- Console – in this state all keyboard input is used to type in commands in order to control the generator.

- Exit – this state is issued upon pressing the key `'Esc'` or typing `"exit"` into the console. When this state is encountered we terminate the main loop and clear all dynamically created data from memory.

Everything pertaining to inner workings has been covered in section 2 and there's no need to go through it once again. There are no fancy tricks present in this class, just mundane code that should be well readable and understandable, because nearly all specific tasks have their own private method. And these methods are called from the public interface.

3.1.1 Configuration file

All available options for the configuration file are listed as an appendix in the user manual. Format of the configuration file is quite simple. Any line beginning with `#` will be treated as a comment. All other lines are either blank or in the format of `option = value;`

The parser omits all blank lines and comments. Other lines are split by `' '`, `'='`, `';'` and then the parser matches first word on the line against all known options. When a match is found it tries to parse the argument and save the value into the appropriate variable. If an invalid value or syntax error is encountered it is written to the log and the parsing resumes.

3.2 Console

Contains basic command line console implementation. So far only typing in characters, deleting them and command execution is implemented. It can't do much but it does its work i.e. user is able to generate new planets, load them and save them. `Update()` method reads characters from the keyboard and writes them to the input buffer. When key `'Enter'` is pressed sends the buffer to parser function and output lines and then clears it.

The parser splits the line by spaces and tries to match first word against known commands. When a match is encountered appropriate execution method is called with the command and arguments (all words found during string split operation). All commands are composed of one word (if more words are needed `'_'` should be used instead of space) – a combination of alphanumeric characters and other characters except `'~'` (tilde). However it is discouraged to use really weird combinations. All arguments are space separated – this disallows filenames or paths with spaces.

The console also doesn't support autocompletiton nor file system interaction. It also won't give any warning if the user is about to overwrite already existing file. So careful approach must be taken. It is encouraged to use working directory as place for any saved planets. Complete list of commands is available in the user manual.

3.3 Log

This is a small utility class that provides interface to write messages into a log file. It is implemented as a singleton. The class provides a method `KeepLog(bool)` to toggle logging on and off depending on the input value. When the log should be kept a log file (`log.txt`) is opened in the append mode and current date and time are written to it.

When something should be logged, the class provides `Write(string)` method. It takes it's argument and appends it into the log file verbatim. It also flushes the output, because we keep the file open until the program terminates – be it expectedly or unexpectedly.

3.4 VertexBuffer

As mentioned in section 2.3.3, `VertexBuffer` class hides from the user implementation of `OpenGL` vertex arrays. It contains array of 65535 vertices (a vertex is a aggregate structure defined in `vertex.h`) and provides an interface to create or remove vertex. Structures described later require shared vertices and having them stored in one place in a way that allows usage of vertex arrays proved to be more elegant than having them scattered all around memory inside classes with reference counters. If nothing else it is much less error prone.

Vertex structure holds information about its position, colour, texture coordinates and normal. `VertexBuffer` can use all four for rendering through specifying vertex, colour, texture coordinates and normal arrays – these can be switched on and of through `Enable*Array` and `Disable*Array` methods.

During program update phase new vertices can be added into the buffer and old ones can disappear. To keep track of free space inside the buffer we are holding an index at a position after the last vertex plus we store non-continuous free space between beginning and the highest peak in priority queue. When we add new vertex, we first look into the priority queue if there's a space before the high peak. If there is, we pop it from the queue and store the vertex there. This operation is $O(log\ n)$ – we use STL's container `priority_queue` that constructs heap above supplied vector of numbers. If there's no free space we move the high peak one index to the right. When we run out of space we return `V_BUF_SIZE` constant that denotes size of the buffer (65535 vertices in the current implementation).

Note that during adding or removing vertices there's no memory consumption since the memory for them is allocated when the class is created. All we do is updating vertex counter and free space information. Upon successful creation of a vertex (which can be created either blank or initialized – depending on which overloaded `NewVertex()` method is used) an index into the buffer is provided. Access to stored vertices is made possible through `GetVertex(size_t)` method that returns reference to the vertex structure

stored under supplied index. Because speed is crucial no check whether we are in the buffer boundary is done so it is programmer's responsibility to supply valid index. Result of obtaining vertex from position marked as free space is undefined – valid reference is returned, but data can contain anything.

In order to render anything polygons must be constructed from the vertex array. This is done via `AddTriangle()` method that takes three indices into the vertex array and stores them in order into the index array (which is 4 times bigger than vertex array). A care must be taken to group vertices in correct order, otherwise `OpenGL` will think it is a back face and will not render the polygon. `PushData2GPU()` method must be called before rendering. It takes all enabled arrays and copies them into appropriate arrays on the GPU unit. `Render()` then takes indices in the index buffer between 0 and 3 times the number of triangles and renders them using specified method. These are:

- Wireframe – all triangles are rendered only as edge lines.
- Culled wireframe – same as above, except only those triangles facing the camera are rendered. Good when you want to see the terrain without the back of the sphere.
- Gray scale – planet is rendered as textured solid. Height map texture is cube-mapped to the surface.
- Textured – custom vertex and fragment shader is used instead of fixed rendering pipeline to produce mesh with terrain texture and oceans.

After calling `Render()` method again an index buffer must be cleared to make space for new frame data. When new planet is generated method called `Purge()` is used to get rid of all vertices in the buffer – it resets free space pointer to the beginning and clears the priority queue.

`DumpObjData()` method can be used to "render" contents of the vertex buffer into external file in the `.obj` format, which is Wavefront's text format for representing meshes.

3.4.1 Shaders

As mentioned above *Slartibartfast* uses `GLSL 1.1` shaders to render planets with terrain textures. These shaders are defined in `vertex_shader.shd` and `fragment_shader.shd` which should be distributed along the executable. They are loaded from the `Init()` method. Any errors are reported and shaders are then not used for rendering. If everything runs smoothly and shaders are compiled they are then used to replace `OpenGL` fixed pipeline to render the planet.

Vertex shader mainly transforms vertices according to the projection matrix. All vertices that are under the ocean level, are moved to it, so water

surface is created. Fragment shader then uses information about the fragment obtained from the cube map texture and parameters specified from the program to determine final colour of the fragment. The value is calculated from the height, weather zone is also accounted for. All fragments that are under the water surface are textured as water, all others are textured with a terrain texture.

Texture coordinates are calculated with a technique called "tri-planar texturing". It uses three planar projections of 3D texture coordinate and blends between them in order to use the projection that is the closest to the direction current fragment is facing. This technique however requires three times more texture fetches than standard techniques, on the other hand it produces nice results. The technique is described in [5, chapter 1]. Fragment shader also calculates per-pixel lighting.

3.5 Camera

This class holds and updates the view port. It provides abstract interface for user defined camera controls. The `Update()` method is used for obtaining user input to control the camera and updating its state. Camera is defined by position of the eye, focal point and three angles: roll, pitch and yaw. Interface for obtaining these values is provided as well as two basic prototypes for setting up the camera. These values are used in rendering as arguments for `gluLookAt()` function.

3.5.1 Rotating camera

Default camera (called "homeworld") has its focal point fixed to the point of origin and rotates around it. It also allows user to zoom in and out.

3.5.2 Spectator camera

Another type of camera known mainly from FPS games. It allows user to freely move and look around. There are no collisions with the planet and the up vector always points in the direction of z axis.

3.6 RenderVisitor

In order to allow flexibility a visitor pattern has been used for rendering planet objects. Planet's `Render()` method propagates supplied visitor down to the triangles. `ROAM_Triangle.Render()` serves as an acceptor for any class derived from `RenderVisitor`. Visitor's own `Render()` method is basically a visit method that has access to the public interface of supplied `ROAM_Triangle` object. Thus the method for rendering is separated from the implementation.

Currently all visitors use `VertexBuffer.AddTriangle()` for specifying polygons to be rendered. It is also possible to use immediate mode inside the visitor and thus removing vertex buffer from play (however `VertexBuffer` class will still be needed to provide vertex information).

3.6.1 DummyRenderer

This visitor takes the triangle and sends it to vertex buffer for rendering. It is used during wireframe and textured rendering.

3.6.2 CulledRenderer

This visitor checks whether the triangle is facing towards the camera or not by calculating dot product between one of its vertices and polygon normal. When sharp angle is detected the polygon is send for rendering. Otherwise it is discarded.

3.7 Planet

`Planet` class serves a shell for harboring any algorithm used for planet generation. By swapping the private sphere object for anything else, one can implement his own algorithm instead of using current implementation. It provides public interface for all allowed operations upon the planetary body through proxy methods. It also holds all planet's parameters (contained within a structure defined in `define.h`). These are:

- Radius
- Max height – defines maximal/minimal height of the terrain measured from the radius. Care should be taken when specifying it, high values can produce non-pleasing outputs.
- Ocean level – defined as a radius below which we deem the terrain as ocean floor. Visible in textured rendering via shaders.
- Angular velocity – defines rotation of the planet (in the current implementation rotation of the light source around the planet).
- Seed – random number seed for the noise object inside planetary generator.
- Weight factor – explained in section 3.13.1
- Exponent – explained in section 3.13.1

3.7.1 Save file format

Planetary data can be stored for further re-generation in a text file on disk. Format of such files is simple and is the same as format of configuration files with the exception of different settings. Any line beginning with `#` will be treated as a comment. All other lines are either blank or in the format of `option = value;`

The parser omits all blank lines and comments. Other lines are split by `' '`, `'='`, `';'` and then the parser matches first word on the line against all known parameters. When a match is found it tries to parse the argument and store the value into the appropriate variable. If an invalid value or syntax error is encountered it is written to the log and the parsing fails.

First non-blank, non-comment line must contain `version = V_STRING;` where `V_STRING` denotes version of the save file. Basically each time some parameters are removed from the file structure or added to it, the version should change, so the parser will reject incompatible files (either the result would be incomplete or weird values would be set as parameters). The parser also controls whether all expected parameters were present or not.

All parameters needed for successful saving and loading are contained within `planet_info_t` structure defined in `define.h`. All parameter strings and version string are defined in the beginning of `planet.h`. For detailed parameter explanation see either appropriate section in the user documentation or appropriate sections in this document (namely section 3.7 and section 3.13.1

3.8 ROAM Sphere

This class is a heart of the ROAM algorithm. ROAM stands for Real-time Optimally Adapting Meshes and it is an algorithm for displaying terrain in various level of detail dependent on the camera distance i.e. distant areas are rendered with less details while near areas are more detailed. You can read more about ROAM algorithm in the official paper[3].

It basically treats the terrain as a single quad formed by two triangles. Some error metric is used to determine when the visual error is big enough to increase level of detail. This is achieved by splitting the triangles by set of predefined rules to prevent cracks occurring in the mesh. When the details aren't needed we merge triangles back to larger ones. The rules are simple – splitting is allowed only when the neighbour along the longest edge is the same size as the triangle we are about to split as shown in figure 3. Splitting triangle T requires introduction of new vertices and edges (shown in red).

Merging requires triangles to be merged to form a diamond i.e. four triangles sharing a vertex lying by their right angle. Every split operation creates 1 diamond and destroys between 0-2 diamonds while merge does the opposite. A care must be taken when operating on triangles by the edges

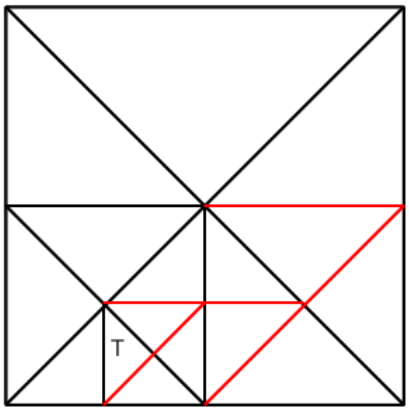

Figure 3: Split diagram

of the quad to prevent memory leaks. However our implementation doesn't need that.

This idea was presented by Sean O'Neil in [4]. He proposed to create a cube of six ROAM quads to form a sphere. What he does when he splits a triangle and introduces new vertex is, instead of just looking up its height, moving it to the surface of the sphere (determined by its radius) and then adding up vertex height obtained from the noise generator. It is this implementation that is used in *Slartibartfast*.

3.8.1 Initialization

What we do is creating 12 triangles – each two composing one face of a cube – and then letting them subdivide. These 12 triangles represent roots of a tree structure generated by splitting operations. This approach literally creates a sphere from a cube.

We create the cube inside private method `buildCube()` which is called from `Init()`. First step is creating 8 primary vertices. Then we create 12 triangles and assign primary vertices to them. Figure 4 shows layout of the vertices and triangles with respect to the code (black numbers denote vertices, red numbers denote triangles and red lines indicate neighbour relationships – apart from trivial along share edges).

When the cube is ready we generate the height-map texture that is cube-mapped to the sphere in gray-scale and textured render modes. We generate each plane (there are 6 in total) separately inside two loops making $n \cdot n$ samples, where n is the size of the texture specified inside the configuration file.

Before the loop we set six initial vectors pointing towards vertex to which

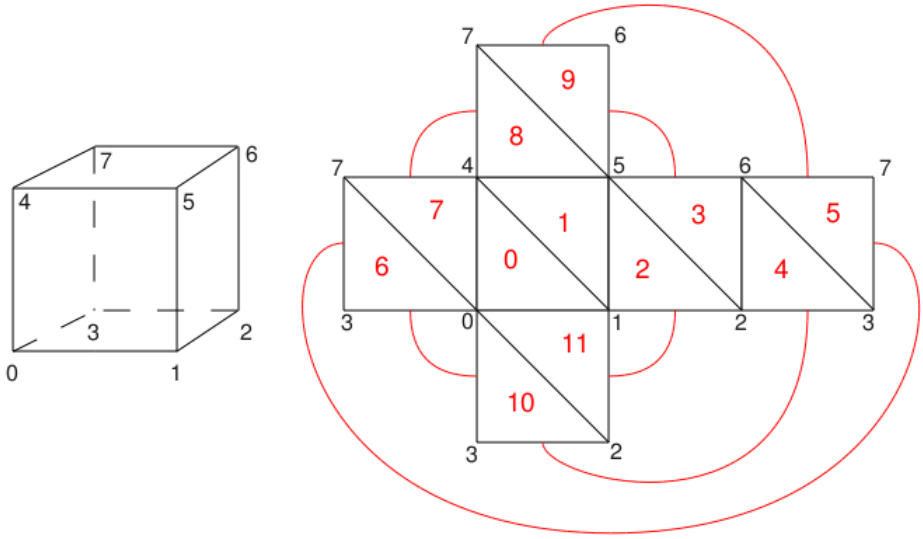

Figure 4: Triangle layout

the top-left corner of the texture is mapped. Then we calculate the increment by dividing length of the cube edge by n. This way we get non-linear probing rays distribution along the given wedge (from the origin towards each face respectively). Probing rays near the edges will be more dense than in the middle of the face, which can help to prevent distortions and artifacts along the texture seams.

We then loop $n \cdot n$ times and gradually increment initial vectors along which we cast probing ray. I.e. we read the height in the given direction and given maximal height for the planet we calculate hue of gray and write it to the texture. At the end we set all six faces as appropriate cube map textures.

3.8.2 Update

Each time an `Update()` method is called call recursive update on all 12 triangles and then update the diamond list each sphere possesses.

3.8.3 Render

There are two possible ways to render the sphere in our implementation since we hold the whole tree structure. First way is via standard recursion, where we traverse the tree until a leaf is found which then calls `Render()` method of the supplied visitor with the leaf as the argument.

The other way is non-recursive, which is achieved by holding a list of leaves obtained during update phase. Then we call `Render()` method of the supplied visitor on all elements of this list. Which method will be used can be switched at compile time via defining or un-defining preprocessing constant `CFG_USE_RECURSIVE_RENDERING`.

3.9 ROAM Triangle

Triangle represents basic element of the ROAM algorithm. Each triangle is right and implements public methods for rendering and updating. Triangles form recursive tree structure with leaves being current triangles to be rendered while the inside nodes serve as cache for merging.

3.9.1 Update

`Update()` method uses DFS algorithm to get the leaves. Once in the leaf we calculate visible error for the triangle and compare it with error threshold supplied via parameter. If the error is larger we initiate split operation. During DFS phase we also check for triangles that have been marked as dead and delete them.

There is one overload of the `Update()` method. This one doesn't take camera object pointer nor error threshold. Instead it uses maximal depth of recursion and current depth and until the current depth is less than the maximum it initiates split operation. Therefore we create uniform triangle distribution of triangles along the sphere thus same level of detail everywhere. This method is used for exporting planetary data into the `.obj` file format.

3.9.2 Split

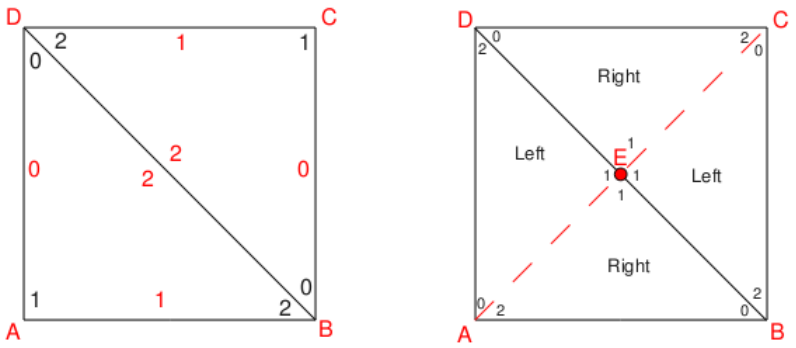

Figure 5: Triangle schema

Figure 5 shows how the situation looks before and after splitting. Black numbers denote local index of vertices, red numbers denote edge indices. We can see that both triangles on the left side of the figure share 2 vertices (B and D), but have them listed under different local indices. This is because we need to keep same vertex-edge orientation for all triangles. Each triangle has three element array containing indices to `VertexBuffer` (red letters in the picture would correspond to these). Each triangle also contains three element array of pointers towards neighbours along all edges. Both arrays are indexed locally as shown in the figure.

Now, when we split this triangle first thing we have to check is whether we are part of a square. A square is composed from and only from two triangles sharing the longest edge (as shown in the figure). For this `isSquarePart()` method is implemented. It basically does this (in order to be as fast as possible we don't check for null or invalid pointers):

if $neighbour[2] -> neighbour[2] = this$ **then**
  return *true*
else
  return *false*
end if

If we are not part of a square, we must first call `split()` method on the neighbour along the longest edge and only then we can proceed. By splitting we introduce new vertex E, so we ask `VertexBuffer` to give us its index. If there is no space left we quit immediately. We assign position (precalculated, when the triangle was created), texture coordinates, normal (actually normal is calculated at the end, when we have all four new triangles ready, so we can calculate a normal as average of normals of triangles that share the new vertex).

Then we allocate space for both children of the current triangle and the opposite (neighbour along the longest edge and part of the square). We set parents to these children and assign vertices to them (how we do that is shown in the figure 5 in the right). Then we update neighbourhood of all four new triangles – so each triangle has valid pointers towards its neighbours along all edges.

An important step is to check whether or not we (or the the opposite triangle) weren't part of the diamond. If so, we need to mark these as dead. We then add new diamond to the list. Last step depends on usage of recursive rendering or not. If we do not use it, then we add four new triangles to the list of leaves and remove both parent triangles from the same list.

3.9.3 Render

As mentioned in section 3.6 we use visitor pattern for rendering. `Render()` method serves an acceptor and takes one argument of `RenderVisitor` type. It then passes current instance into the visitor's `Render()` method.

3.10 ROAM Diamond

`ROAM_Diamond` is a meta object. Each triangle has pointer to one, because each triangle can be part of exactly one diamond, or doesn't need to be (in that case the pointer leads to null). Diamonds are implemented to take the burden of merging triangles together. Instead of checking the condition inside triangle update method we do it from diamond – it would require calling `ROAM_Triangle.isDiamondPart()` more times than it is necessary.

When we get diamond object we already know that it was either destroyed by split operation or that it is still alive and don't need to check more things (to check whether a triangle is a part we just run in circle along one of the short edges, if we get back to original triangle we form a diamond).

Diamond update method is very similar to the one of a triangle. We calculate visible error assuming a merged diamond, compare it with supplied merge threshold and if such metric is less or equal the threshold, we merge the diamond. Merge operation is exactly opposite to split, so figure 5 applies as well, except this time left side is post-merge and right side is pre-merge.

3.10.1 Merge

When a merge is invoked we take parents of all four triangles (two of them share one parent, the other share another parent), update their neighbourhood using information from the diamond. Then we take optional step of removing two leaves from the leaf list and adding one new (the parent becomes a leaf again). Last step is to check whether the parent forms a diamond, if yes we add it to the list. This is done twice – once for each parent.

Last three steps of merging yield recalculation of vertex normals, removing vertex E from the buffer and marking four merged triangles as dead, so they can be removed in the next triangle update cycle (see section 3.9.1). We also mark the current triangle as dead, so it will be removed in the next update cycle of diamond list (see section 3.11).

3.11 ROAM DiamondList

This class implements linked list of all diamonds created through split and merge operations and provides interface for adding new diamonds. It also implements `Update()` method, which is called once per main loop cycle and runs through all diamonds in the list. If a diamond marked as dead is found it is promptly removed from the list, otherwise `ROAM_Diamond.Update()` method is called on it.

3.12 Noise

This is an abstract interface for implementing n-dimensional persistent noise, in practice we use only 3 dimensions, however the number can be increased.

By persistent we mean that the output of the noise function will be the same given the same input. The class declares virtual method to initialize such noise generator with a seed value and provides three pure virtual methods for one, two and three dimensional noise.

3.12.1 Perlin noise

More on the Perlin noise can be found in [2]. In short we use 3-dimensional lattice of random numbers to generate the output. Up to three numbers are passed as its input (if less than three numbers are passed, the rest is padded with zeroes).

We take integral and fractional parts of these numbers. Integral parts are then used as indices to the lattice. A single index is constructed from all three indices and a three dimensional vector is taken from pre-calculated array. All elements are multiplied by fractional parts respectively and a single value obtained as sum of all elements is returned.

For one dimensional noise we take two random numbers from the lattice and interpolate between them (current implementation uses linear interpolation). To obtain one we use first element of the supplied vector, to obtain the other we add 1 to the first element of the vector.

Two dimensional noise uses four random numbers obtained through the original vector and all possible combinations of adding 1 to the elements. Then first two and last two numbers are interpolated between and the result is again interpolated.

Three dimensional noise does the same, but with eight numbers. Through interpolating between pairs we obtain four values. These are interpolated between in pairs again. And the results are again interpolated.

All return values are truncated to $(-1.0, 1.0)$.

3.13 Generator

This class provides an abstract interface for implementing height-map generator. Each generator is initialized with a seed and provides method for obtaining height in a given direction from the center of the sphere.

Basically as `ROAM_Sphere` defines implementation of LOD algorithm and rendering and thus by changing it for something else, one can change how the actual terrain is handled. By implementing new generator and using it instead of the default one, one can change how surface heights are generated. `Noise` object provides heart for each generator and it can be easily replaced with another implementation as well.

3.13.1 DW Generator

DW stands for Dynamic World. This generator class implements simple fractal Brownian motion to produce more interesting results than plain Perlin

noise that is quite smooth. `GetHeight()` method is parametrized with direction vector and number of passes k. `DW_Generator` class is also parametrized with two values stored as private members. These are weight factor and exponent (or lacunarity). `GetHeight()` method then does this:

{x, y, z and k are supplied as function parameters}
$w := 1.0$
$n := 1.0$
$result := 0.0$
for 1 to k **do**
  $h := w * noise(x \cdot n, y \cdot n, z \cdot n)$
  $w = w \cdot w_{factor}$ {w_{factor} is class a member}
  $n = n \cdot exponent$ {$exponent$ is a class member}
  $result := result + h$
end for
return result

If we look at the pseudo-code we will see that `w_factor` influences how much noise from larger octaves will used. And `exponent` influences how smooth the noise will be or how big differences will occur between two close points. Experimenting with these parameters can produce more interesting results than default ones (2.0 for exponent and 0.5 for weight factor), however it can also produce completely weird landscape.

Also note that if k is big enough the execution of this method can take considerable amount of time. Because 3D noise function will be called k times. In current implementation it constitutes of 8 lattice lookups and 7 linear interpolations. Each lattice lookup constitutes of 3 multiplications and 3 modulo operations. Together we get approximate time complexity:

$$O((6 \cdot 8 + 7) \cdot k)$$

Where k is number of passes. Theoretically this isn't that bad, but 55 multiplications with default number of 10 passes can cost some time especially when we account number of generator's `GetHeight()` queries – which happen every time new triangle is introduced (every split operation creates four new triangles).

4 Further improvements

This program serves as a demonstration of what is possible. It is not complete and there is open space for further improvements. These include (not definitive list):

- Implementing some sort of interpolation between LOD levels to reduce visible popping of triangles.

- Optimizing ROAM algorithm implementation to split triangles "in-place" i.e. not creating recursive tree structures, but holding only leaves.
- Optimizing triangle creation so two triangles sharing the longest edge don't calculate new vertex position twice, which increases number of `Generator.GetHeight()` calls.
- Introducing better triangle occlusion mechanism, so not all triangles facing the camera are sent to the GPU, i.e. introducing some sort of line of sight algorithm or other apparatus for efficient weeding out those triangles that are facing the camera but aren't in its point of view.
- Searching the possibility of using another algorithm for planet representation, one that would be more efficient in terms of vertex data updates. At this moment we need to discard data from the old frame and begin rendering from scratch, because we don't know how much the structure changed between them.
- Researching the possibility of using normal maps to increase level of detail and visual quality without changing the geometry.
- Introducing scene anti-aliasing for better visual quality.
- Implementing GUI for better user interaction.
- Introducing axial tilt of the planet.
- Improving weather zoning – mainly implementing some persistent noise into the shaders.
- Using other texturing techniques instead of tri-planar texturing that requires three texture fetches.
- Writing water shader for visually appealing water surfaces.
- Rendering star field and sun.
- Implementing real-time atmospheric light scattering.

References

[1] Shreiner D., Woo M., Neider J., Davis T., *OpenGL: Průvodce programátora (authorised transl. of* OpenGL programming guide: The official guide to learning OpenGL, version 2, 5th edition*)*. Computer Press, a.s., Brno, 1st edition, 2006.

[2] Perlin Ken, *Making noise*. Talk about Perlin noise available on-lin on http://www.noisemachine.com/talk1/

[3] Duchaineau M., Wolinsky M., Sigeti D., Miller M., Aldrich C., Mineev-Weinstein M. *ROAMing Terrain: Real-time Optimally Adapting Meshes.* Available on-line on https://graphics.llnl.gov/ROAM/ 1997

[4] O'Neil Sean, *A Real-time procedural universe, part two - rendering planetary bodies* Available on-line on Gamasutra.com 2001

[5] Nguyen, H. et al., *GPU Gems 3* NVIDIA Corporation, USA 2007