

EAGLE EYE - ROMANIA 3D MAP FEATURE EXTRACTION

Horia Stefan Lupescu¹
Costin-Anton Boiangiu²

ABSTRACT

This paper describes an application that simulates a virtual 3D map of Romania and the improvements we have done, so that the program will use the system's resources better, run faster and improve user-interaction.

The application uses two sets of images. The first set is a collection of satellite photos of Romania in jpeg format, using the Stereo70 projection coordinates. The second set of pictures consists of grayscale images in GeoTiff format, which represent the height of the land. The program uses the grayscale images in order to simulate the 3D feature.

The program is organized as a client-server application. The server's job is to store the large number of files and make small computations, coordinate transformations mostly. The client requests data from the server and uses the images it receives in order to generate the terrain, creating a video-game like environment. The user is allowed to browse the map, in any direction he wishes, or jump to any location on the map.

Special to the program is that it includes algorithms for region classification in large satellite images from the map of Romania using textural measures, namely Haralick Features and tonal information. Two classification techniques are proposed: one of them uses the Mahalanobis distance to get the most appropriate class and the other uses a machine learning algorithm, Support Vector Machines (SVMs).

KEYWORDS: Haralick features; Mahalanobis distance; client-server; Support Vector Machine;

1. INTRODUCTION

The Eagle Eye application's purpose is to create a virtual 3D world, by processing real data. In order to achieve this, a very large image database is needed. This database consists in two very large sets of pictures. The first set of pictures consists of grayscale images, stored in GeoTiff format. The information these pictures provide, represents the elevation of the land. The application uses the data it collects from these pictures in order to simulate the three-dimensional feature of the virtual world. The results are spectacular, especially in the mountain areas. The second set of pictures consists of real satellite images. These pictures are rendered in order to create the virtual world as close to reality as it can be. The user has the freedom to browse the map in any direction he wants, jump from location to location or even zoom in or out. The result is very close to a video game, only that the virtual world is created using real-life data.

¹ Engineer, horia.lupescu@cti.pub.ro, "Politehnica" University of Bucharest, 060042 Bucharest, Romania

² Associate Professor PhD Eng., costin.boiangiu@cs.pub.ro, "Politehnica" University of Bucharest, 060042 Bucharest, Romania

Because the application is very computationally intensive and because the large number of files that need to be processed, the program had to be organized as a client-server application.

The server is responsible for the 70 GB of pictures that represent the input data. The server listens on a certain socket. If a client initiates a request, the server's job is to transform the coordinates from Stereo70 (used in Romania) to WGS84 (World Geodetic System 1984) used by the elevation images. Also, the server performs the transformation from WGS84 to Stereo70. The server finishes the coordinate transformation and delivers the requested data to the client.

The client represents an OpenGL application. Its role is to generate the virtual world, allow the user to explore it and interact with it. The client displays only a small part of the map at a time. If the user starts exploring the map, then the client has to request new data from the server.

The client has been developed in C++ and the server has been developed in C.

Feature extraction and object recognition represent common topics in computer vision domain. In particular, extraction of features from large satellite images is one of the most interesting and challenging field in computer vision because of various reasons:

- the luminance, which may be different depending on the moment of the day;
- the weather conditions - the possible clouds that can be present when the pictures are taken which appear like a fog or like a white spot;
- the shadows of the objects such as trees, buildings etc. which disturb the recognition and introduce confusion.

The software developed can detect the following types of patterns:

- forest
- land (agricultural terrain, grassland)
- water (lakes and rivers)

In order to automatically detect patterns, it is necessary to have a set of measurements with very similar values for patterns in the same category and very different for patterns in different categories. For this purpose, we used textural measures and colors. The component that calculates the textural measures uses functions from the OpenCV library. The software includes two different techniques of classification, mainly based on the textural measures. The two classification methods will be described in detail in the following chapters.

The accuracy of the detection is good, having an average of 89%, although it can be improved. There are types of forest which are confused with land and the shadows of the trees in a forest are classified as water because of the homogeneous texture. The patterns recognized are marked in the output image with particular colors.

The solutions of this problem cover multiple research domains such as: machine learning, computer vision, parallel application. Processing only one satellite image takes an important amount of time because of the computationally intensive operations, even if it is used an innovative algorithm. Therefore, to reduce the runtime, we parallelized the code using a transparent mechanism with OpenMP threads. The tradeoff of using parallelism is given by the speedup obtained.

2. CASE STUDIES AND ALGORITHM DEVELOPMENT

Client-server architecture is used because the application data is stored in a very large database. The client-server communication is performed using Windows sockets and a very simple protocol.

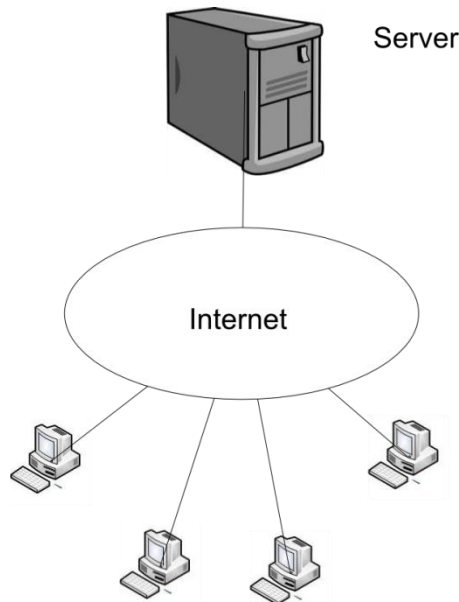


Figure 1. The Client-Server architecture

The Server

The server is not just a database server, but also computes coordinate transformations from Romanian Stereo70 projection system in WGS84 (World Geodetic System 1984). When a client request is received, the corresponding satellite image (jpg) is fetched. The related elevation data is loaded from up to four TIFF images. The data is then sent to the client in order to be processed. The World Geodetic System is a standard for use in cartography, geodesy and navigation and is the reference coordinate system used by the GPS (Global Positioning System).

One of the improvements we have made to the project was to send the pictures in one data transfer, instead of using two different transfers. The elevation data and the satellite image are packed and sent together in the same transfer frame to the client.

The Client

The OpenGL client represents the biggest part of the project. The program is responsible with processing the images received from the server, generating the ground texture, displaying the virtual 3D map and interacting with the user. The environment is similar with a video game, allowing the user to move anywhere, using the keyboard. The client also allows the user to introduce new coordinates and jump directly to those coordinates.

The ground texture can be generated in two forms: using the satellite images, which results in a very good virtual simulation of the real world, or by using three textures: sand, water and snow. The second option plays an aesthetic role and it permits the user a better analyze of the land’s height.

The application displays a few information messages on the screen, such as local coordinates, the amount of memory used by the program and some other information the user might find useful. When the program is busy, usually for generating new grounds, the “Loading...” message is displayed and the user needs to wait until the textures have been processed.

After a request, the client receives a response from the server, which consists in a set of images: the TIFF image and the jpg image. The data is processed and then the ground is generated depending on the elevation information gathered from the TIFF image. It is obvious that the 3D effect can be observed only for landforms and not for buildings, roads, etc., which means that the user can’t browse the cities as interactively as browsing the country’s land.

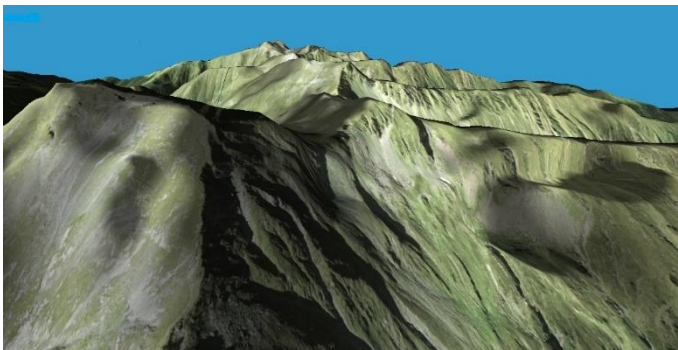


Figure 2. The picture above represents a mountain landscape and is a good example of 3D visualization.

The virtual map is displayed over 9 ground squares. The 9 squares are organized in a 3x3 matrix. The user’s position is always above the central ground square.

2	5	8
1	4	7
0	3	6

Table 1. This table illustrates how the ground squares are numbered. The user is always above the central square, which is square number 4.

Only 9 squares can be displayed at a time, which means that in every moment, the map is drawn using 9 square grounds. If the user crosses the central ground's borders, the matrix is reorganized, so that the user is still above the central square. The squares will be renumbered and new grounds will be loaded, depending on the direction the user is going. The grounds that are closed to the user's position will be loaded, while the grounds that are further will be deleted from the system's memory in order to make room for the new ones.

	2	5	8
	1	4	7
	0	3	6

Table 2. An example of how the squares are managed.

In the table above, it is illustrated an example of how the program managed the grounds in the previous version of the application. In this case, the user is moving from left to right. The red column represents the grounds that are deleted from the memory, while the yellow column represents the new grounds that are loaded. The matrix needs to be rearranged as follows:

- The first column (squares 0, 1 and 2) is deleted;
- The previous second column (squares 5, 4, 3) becomes the first column (squares 2, 1, 0 respectively);
- The previous third column (squares 8, 7, 6) becomes the second column (squares 5, 4, 3 respectively);
- The third column consists of the new grounds that were loaded. The squares are numbered 8, 7 and 6.

The problem with this algorithm is that the new grounds are loaded when the user reaches the central square border. Because of this, the loading time was too long. In order to shorten the waiting period, we designed a new preloading algorithm, which fetches and renders the new square grounds gradually, depending on the user's direction and the distance between the user and the square's border.

3. PRELOADING GROUND

As we mentioned before, the application renders the virtual map using 9 ground squares at any given time and the user is always above square number 4. When the client reaches the square's border while moving, a new part of the map must be generated. That means that some of the squares need to be deleted in order to make room for the new ground squares that will be loaded.

In the previous version of the application, the new grounds were loaded when the user reached the border of the central square. The user had to wait for a significant period of time while the new images were processed and the textures were generated and displayed. This made movement around the map a bit uncomfortable, because of the big “Loading...” pauses the program had to make in order to display the new world.

The interruption that occurred when changing squares was quite big, because the system had to process at least 3 new ground squares, depending on the user’s direction of movement.

What we tried to do in order to reduce this interruption was to anticipate the direction in which the user was going. While the client is going towards its desired destination, the ground squares will be gradually loaded, one at a time, as the user is approaching the border of the central square. Thus, the new square grounds are requested, processed and drawn each at a time. That means the big interruption in the previous version, has been divided in 3 smaller pauses, each no longer than maximum 2 seconds, depending on the image that is being processed. In this way, the transition from one square to another is done smoothly, without any “Loading...” pauses. The new grounds are stored into a stack, called *groundStack*.

The algorithm works like this, in order to decide when to load each piece of new ground:

- When moving left-right or up-down (keys A and D), the function *getTarget()* identifies the square towards the user is moving and computes the distance to the border. If the user moves north or south, the target is square number 5 or square number 3 respectively. If the user moves east or west, the target square is 7 or 1 respectively.
- The *newGroundID()* function is called in order to check if it is time to load a new square, depending on the distance to the border.
- The *loadNewGround()* function identifies exactly which square needs to be loaded, depending on the user’s direction.
- The *groundStack* stack is used to store the preloaded grounds. The new grounds are stored in this stack until the user changes from one square to another. After this, the stack is emptied. The stack is also emptied if the user changes direction suddenly, before reaching the border. That means that the user changed his mind, making the preloaded grounds (if any) unnecessary.
- The *reloadMap()* function was strongly modified for improving memory management and to use the preloaded grounds, if available in the *groundStack*. Each call to this function empties the stack and resets the global variable, *isLoading*, used to check the existence of the preloaded grounds. The reset is done by calling the *resetNewGround()* function.
- The *idleFunc()* functions was modified to generate texture and display lists for the preloaded grounds.

If the user is right on the border (exactly between 2 adjacent squares), the *isOverGround()* functions does not provide a solution. This function’s purpose is to check if the user’s coordinates are above the current displayed ground. So, to avoid a deadlock in this scenario, the program backtracks the user one step inside the central ground.

The *newGroundID()* procedure will determine when the new square grounds should be loaded and depending on the its result, the *loadNewGround()* function will compute which new square grounds are going to be preloaded.

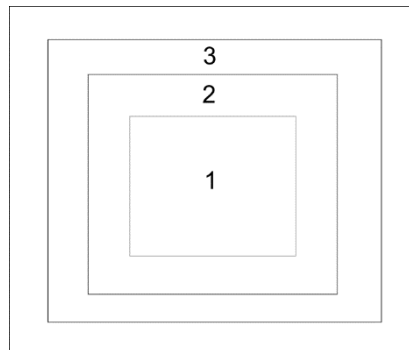


Figure 3. Preloading areas used by the *newGroundID()* function

The above figure shows how the central square is divided into 3 preloading areas. If the user is above square 1, the first request to generate a new ground is sent. If the user is above square 2 and outside square 1, the second request is sent. The same applies for the third request. All the requests are generated if the distance from the user's position to the border is less than half of the square's length.

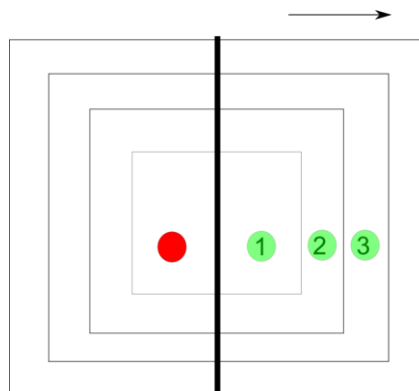


Figure 4. Example of preloading areas

If the user is moving from left to right as shown in the picture above, the central square is divided in two halves. In the left half, marked by the red circle, nothing happens. As soon as the user moves into the right half, the requests are sent according to the position of the green circles. The new grounds will be loaded and processed, only if the user keeps moving towards the right border.

If the user changes direction suddenly (goes left, up or down), all the new square grounds that have been loaded in advance, will be deleted, by calling the *resetNewGround()* function. If the user decides to go in the same direction as before, not all the grounds may

be preloaded, depending of the user's position. The same function is called if the user modifies the viewing angle or zooms in or out of the map.

For example, if the user is at green circle number "2" while moving right, then two grounds are preloaded. Next, the user decides to go left, but not enough to be placed near circle number "1". In this scenario, the *groundStack*, which consists of the two preloaded grounds, is emptied. If the user decides to go right again, then only one ground would be preloaded (marked by circle number "3"), because that is the only condition that is valid in the *newGroundID()* function. So, when the user reaches the square border, two new grounds will be loaded.

4. TERRAIN RECOGNITION

The application is composed of multiple components:

- Feature extractor - used to extract the textural features for a sample of fixed size of 32x32 pixels
- Mahalanobis classifier [6]- the component responsible to classify the pixels of an image using the Mahalanobis distance
- SVM classifier - use Support Vector Machines technique [3][4] with the "one against one" approach for the multiclass classification
- Training set builder - an auxiliary component which has the purpose to build a set with textural measures for each class that must be identified
- Mean vector and covariance matrix creator - a component specific to the Mahalanobis classifier [7] which computes texture descriptors (the mean vector and the covariance matrix) for each class in order to use them as models.

Texture Classification

Firstly, for each class that we want to extract, i.e. forest, land and water, we have chosen a set of significant samples (of resolution 32x32 pixels) with their textures and computed the Haralick features [1] - measures that were stored in a configuration file (named "HaralickMeasures") to be used forward by the classifiers.

Secondly, to classify the pixels of a large satellite image, we split the image into squares of fixed size - 32x32 pixels -, apply a classification algorithm for each square and label it with the resulted class.

Mahalanobis Distance Approach

After reading the configuration file, we stored the measures in a map which have as keys - the classes and as values - vectors of vectors of doubles. To prepare the necessary things for computing the Mahalanobis distance, we calculated the mean vectors and the covariance matrix for each class using the data from the map and stored them in a map as well. This

time, the map contains as values structures which encapsulate the mean vector and the covariance of each class.

For each square from the satellite image, that we want to classify, we compute the Mahalanobis distance along each class and label the square with the class that has the minimum distance only if the distance is lower than a threshold equal to 50. We mark a square as belonging to a class by enhancing one distinct color for each class. Actually, we let the bands corresponding to the wanted color and reset the other bands. For example, to obtain yellow which is formed from green (G) and red (R), we reset the blue (B).

We have chosen the colors as follows:

- green - forest
- yellow - land
- blue - water

"uncolored" - unknown

Support Vector Machines

A more powerful technique recently applied to the problem of remote sensing data classification is the Support Vector Machines (SVMs) [5]. SVM, also called Support Vector Network, is a supervised machine learning algorithm that analyzes data and recognizes patterns, used for binary classification and regression analysis. Intuitively, given a training set with examples from both classes to be learned, the basic SVM builds a linearly separable model that distinguishes between the two classes. Then, the classification problem can be modified to handle non-linearly separable data by mapping data into high-dimensional spaces. Moreover, as most of the applications involve multiclass classification, a number of methods have been proposed to generate multiclass SVMs from binary SVMs. The main idea behind SVM classification technique is to separate the classes with a surface that maximize the margin between them, the decision surface being defined by the boundary points. This method is independent of the dimensionality of the feature space, so the surface can be called hyperplane. The data samples that are closest to the hyperplane are termed support vectors.

Feature Extractor using OpenCV

To extract the Haralick features necessary for the classification, we have to do the following steps:

1. for a given square in the input image, knowing the coordinates, create a new image and convert it from RGB to the gray-scale format;
2. having the gray-scale image, create a GLCM structure which corresponds to this image;
3. compute the descriptors for the GLCM created previously;
4. get the necessary descriptors;
5. release the memory allocated: the image and the GLCM structure.

5. RESULTS

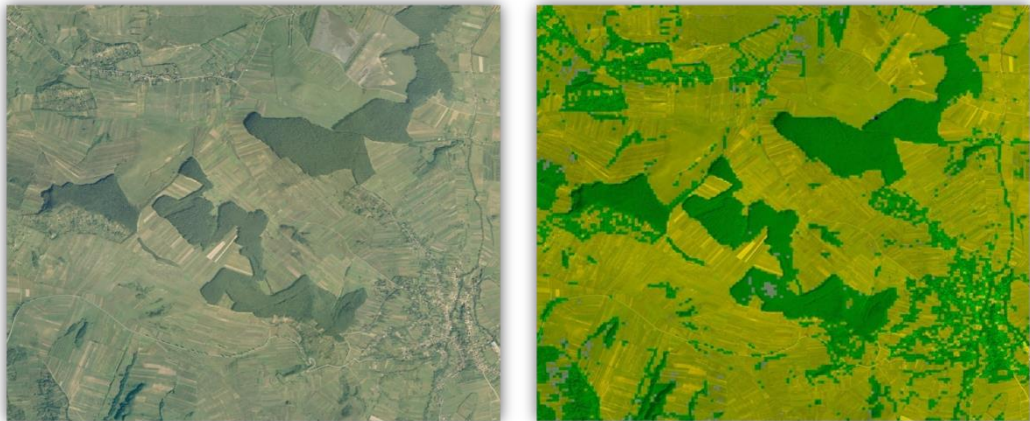


Figure 5. Example of land and forest recognition

As you can see in the above example, the algorithm manages to correctly identify the terrain in the picture with a success rate of almost 100%.

6. CONCLUSIONS AND FUTURE WORK

The feature extraction domain is an exciting field of research. The big number of studies and published scientific articles stand for the popularity and rapid expansion of the research in this area. Libraries with algorithms are implemented to aid programmers to quickly develop projects. The purpose of this thesis was to extract features that can be detected using patterns, like forest, land or water, in order to be used for creating and rectifying maps, detect deforestation and other environmental changes.

To conclude with, we have covered two main techniques of classification based on a set of textural features extracted from samples of images significant for each class that has to be detected.

In the case of the method that uses the minimum Mahalanobis distance for classification, the accuracy is reasonable 70% for images with similar characteristics of quality to the samples chosen to build the training set: hue, saturation, contrast, brightness, sharpness, etc. Having a general training set, the overall accuracy for multiple images is good. If we use a distance limit to classify as unknown the squares with the distance over the limit, we obtain too much unknown samples and thus, the overall accuracy is smaller.

For the SVM classifier, the accuracy is almost the same with the Mahalanobis distance classifier when using the training set which contains only textural measures, the Haralick features [2]. However, after adding to these measures, the tonal information (colors) and 4

specific samples for the tested image, the results were significantly improved, getting an overall accuracy of 85%.

Moreover, in order to obtain a better accuracy, we have introduced a correction phase that uses only colors to distinguish between the class water and the class land. The configuration parameters used for this phase are specialized for the tested image with just a few samples of land and water that are confused in the previous phase. This method gives an average accuracy of approximatively 90%, the class accuracy for water being 97%. However, further specialization of the configuration parameters for a subset of images is not desired because this would make the recognition process semi-automatic and it would be hard to process a large number of satellite images.

REFERENCES

- [1] Robert M. Haralick, "Statistical and Structural Approaches to Textures," Proc. of the IEEE, vol. 67, no 5, 1979, pp. 786-804.
- [2] Robert M. Haralick, K. Shanmugan and Its'Hak Dinstein, "Textural Features for image classification", IEEE Trans. on System, man, and Cybernetics, Vol. SMC-3, No. 6, pp. 610-621 November 1973
- [3] Ben-Hur A and Weston J, "A User's Guide to Support Vector Machines", Technical report, <http://pymml.sourceforge.net/doc/howto.pdf>, 2012, accessed 10 Oct 2014
- [4] Andrew Ng, CS229 lecture notes. <http://cs229.stanford.edu/notes/cs229-notes5.pdf>, 2012, accessed 10 Oct 2014
- [5] Weston, J "Support Vector Machine". Tutorial, http://www.cs.columbia.edu/~kathy/cs4701/documents/jason_svm_tutorial.pdf, accessed 10 Oct 2014
- [6] Mahalanobis Distance, <http://www.cnblogs.com/lyfruit/archive/2013/03/22/2975381.html>, accessed 10 Oct 2014
- [7] Mahalanobis Distance description, http://www.jennessent.com/arcview/mahalanobis_description.htm, accessed 10 Oct 2014
- [8] Herisanu, A.; Hupca, O.; Lupescu, H.S.; Dobre, R.; Slusanschi, E., "Eagle eye - Feature extraction from satellite images on a 3D map of Romania," Tier 2 Federation Grid, Cloud & High Performance Computing Science (RO-LCG), 2012 5th Romania , vol., no., pp.23,26, 25-27 Oct. 2012