Terrain generation based on real world locations for military training and simulation

Peter Dam, Fernanda Duarte, Alberto Raposo
Tecgraf - Computer Graphics Technology Group
Department of Informatics / PUC-Rio
Rio de Janeiro, Brazil
{*peter, fernandaduarte, abraposo*}*@tecgraf.puc-rio.br*

***Abstract*—The task of recreating a real world location in a virtual environment is never easy, and a high degree of similarity is crucial for specialized training and simulation sessions, which are becoming ever more employed in the military to improve training methods. Allowing the users to recognize the location in the virtual environment through the use of real maps, for example, increases user engagement, helping to increase the overall quality of the training session. One factor that must be accounted for is the availability of data to perform the creation of these virtual environments, especially in locations such as areas with low population density or small cities in South America. In this paper we present a method to enable the creation, with limited data availability, of a very large, high quality, and optimized environment for ground-level simulations using Unity 3D, one of the current state of the art game engines.**

***Keywords*-virtual environment; real world; terrain generation; serious games; training; simulation;**

I. Introduction

With the improvement in technology, both in hardware and software, such as game engines, it has become increasingly possible to perform high quality simulation in virtual environments. Games have already reached a very high level of realism and will most probably continue to breach the gap, which lends favorably to the non-entertainment domain, in which we place applications focused on training and instruction. These are commonly categorized as serious games, which have been increasingly used by instructors due to the realism and immersion.

Amid those who make use of serious games, the military is constantly making use of them for training and recruiting purposes. Examples such as America's Army (1) has been used for many years now as a tool for communication, allowing players to explore and learn about the United States' Army, and even for recruitment purposes. Another tool that has been used by several armed forces is Virtual Battlespace (2) (VBS), currently in its third version. Unlike the first example, VBS provides an editing environment, where instructors can design a session tailored towards their specific training needs.

Among the different skills that might need to be developed, one is to be able to properly operate military combat vehicles. One example of games employed toward this end is Steel Beasts (3), which has been adopted by several armies. Physically operating the vehicles is very costly, the vehicles need to be moved to the location and consume fuel during the operation, which makes repetition prohibitive. Another factor that must be accounted for is risk, accidents may happen, possibly incurring in injuries and financial loss, which further limits the training possibilities. In order to explore other possibilities one solution is making use of a virtual environment, where the desired skills can be tested thoroughly. Furthermore, it is possible to create situations to explore reactions to enemies, allowing the trainees to acquire experience in dealing with a vast array of adverse situations before actually encountering them in reality.

One difficulty, however, is finding a simulator that suits the needs, be it because the desired vehicle is not available, or because the terrain is either a fictitious location or some location that does not reflect the desired type of terrain and obstacles. As such, we began to develop such a simulator tailored to the specific needs of the Brazilian Marine Corps. It was requested that the location in which to simulate be of Votuporanga, a small city in the state of São Paulo, Brazil. We chose to use Unity3D (4) as the underlying game engine both because it provides great features and optimization, as well as due our team's familiarity with the engine. Another interesting feature presented by the engine is the Asset Store. All this together allows us to focus on the important part of the simulator, leveraging the features provided by the engine and third party assets.

The environment is the first part we worked on and is the focus of this paper. For this part the key aspects comprise of accurate terrain elevations and appearance, correct placement of the roads and the presence of buildings and housings. The last part need not be exact, but close enough as to impose the same physical and visual obstacles as would be encountered in the real location. Furthermore, it is important that the sky is accurate as well, in regards to the correct placement of the sun at any given date.

We will first evaluate related work in regards to different solutions for generating terrain, then we will describe the method we developed. Next we will present our results and, finally, present our conclusion.

II. Related Work

There are a few different approaches to terrain generation, which will be employed to generate one of the following: a completely fictitious location, a real world location or a fictitious location based on real world data. Among these approaches, we can divide them into three categories: manual modeling, procedural generation, and real world modeling. Even though we are working towards a real world location, we can use techniques employed in other categories as well.

A. Manual Modeling

Manual modeling of terrain is a very laborious task, requiring a lot of time and specialized tools and personnel. Due to the very high costs, this is currently not a common method of terrain modeling both in the entertainment and the serious games industries, with most developers rather adopting a hybrid approach between procedural generation and manual modeling (5).

B. Procedural Generation

Procedural generation of terrain models is the use of algorithms based on randomness to generate unique content. The advantages of this approach is that a huge amount of data can be generated quickly (once the algorithm has been implemented). One can alter parameters and constantly adjust until a desirable outcome is achieved. The downside is that some inconsistencies or unlikely features may occur, so it is usually necessary to manually edit the outcome. This applies not only to the terrain topography, but can also be applied to other aspects such as road networks (6), vegetation, housing placement and more (7).

In regards to terrain modeling, this seems to currently be the most popular, usually employing a hybrid approach, either between manual and procedural or between procedural and real data. Smelik et al. (8) describe an interesting method where the user can easily sketch a terrain with desired features and their system will generate a terrain matching those features. This allows instructors to create scenarios where the terrain will suit their needs. On the other hand there are approaches, such as the one employed by Parberry (9), that use real world data as input to lend greater credibility to the outcome of procedural methods.

C. Real World Modeling

Finally there is also the possibility of representing the real world in a virtual environment. There are many different types of data that need to be processed in order to rebuild the location, and the availability, precision and resolution of the data vary greatly depending on the region of interest. Wells (10) describes a solution for real-time generation of terrain, but this presents a problem for our case: their solution makes use of data that is not readily available for the region of Votuporanga, and, for that matter, the majority of the world other than North America and Europe. Furthermore, by generating the terrain in real-time they guarantee that their world representation is always up to date with the latest available data, but, on the other hand, they must waiver the benefits of pre-processing the terrain in order to improve performance, as well as the possibility of manually editing the terrain to further adjust for better suitability or to remove any flaws caused by the method or by bogus data.

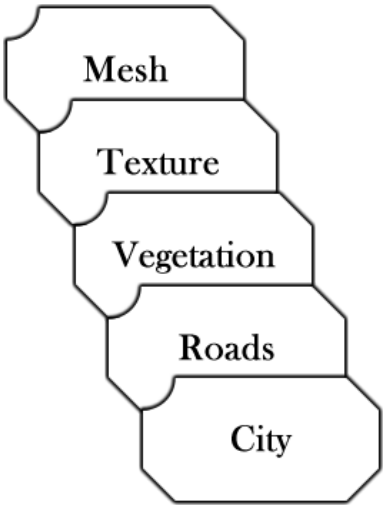


Figure 1. Layers of the terrain.

III. Method

In this section we will describe the methods we followed and developed to achieve the desired results. The process can be divided into separate modules, corresponding to layers of the terrain, as shown in Fig. 1. First we must build the topography, upon which we will apply the ground textures. Then we will build vegetation, roads and finally the city elements such as housings. Most of the steps are individually automated, but the whole terrain is not built as a single batch process because we need to constantly evaluate the results of each step and adjust parameters until the result is satisfactory. At the end we still did some manual intervention either to fix some errors or to make the terrain aesthetically more interesting.

A. Topography

The first part in the process is to generate the terrain mesh. To do this we must first obtain a digital elevation model (DEM) from a provider. While some providers, especially those using airborne measurements, may be able to provide higher resolution data, we found that for the region of interest the spaceborne data sets provided by the United States Geological Survey (USGS) and National Aeronautics and Space Administration (NASA), called the Shuttle Radar Topography Mission (SRTM) (11), were readily available and accurate enough (12) for our purposes, even though there are methods of reducing errors and improving accuracy

(13). Between the different versions of the SRTM we chose version 3.0 (SRTM Plus) (14), henceforth referred to as SRTM.

First the correct data needs to be found, and SRTM does not have an API, it is an organized archive of data split into tiles, indexed by the southwestern coordinate of each tile. The chosen coordinates for any region of interest most probably will not match a tile of SRTM data. Depending on the size of the region of interest it will either be contained inside one tile or spread across two or more tiles, so we must find the tiles that contain the data for the chosen coordinates and extract it from the tiles in which our chosen region is found.

This data is then is interpreted into a matrix and fed into the terrain system of the engine. Likewise, it would be possible to generate the mesh, in which case each point would become a vertex of the terrain and the vertices in between the points would be calculated via some interpolation method such as a bilinear or bicubic (15) interpolation, however this is not necessary since the engine we use has a highly optimized built-in terrain system (16). This, however, yields a terrain that is too smooth, so in order to improve this we then add some randomized roughness using perlin noise (17).

Due to the size of the designated location we chose to split the terrain into tiles for two reasons. The first is that we are able to obtain higher resolution data for smaller regions, so instead of obtaining data for one large terrain, we divide the terrain into smaller tiles. The other reason is that by doing so, we can reduce the amount of data we must load to the memory at run-time, as will be explained later.

The chosen size of each tile was 1 km^2 in order to simplify the detection of thresholds (which will be explained later in this paper), resulting in a 16x16 grid. Since each tile is processed individually, the previous process of adding a randomized roughness causes the edges to mismatch. The next step then is to process the edges of all the terrains to join the neighboring vertices. Note that in this instance when we say neighboring we mean vertices of different terrain tiles that have the same coordinates on the horizontal plane (in our case X and Z). There are two cases of neighboring vertices: a point shared by two terrains and a point shared by four terrains, which is the case of corner vertices. For all neighboring vertices we set the new height to the average between the two or four values of that point. Fig. 2 shows one tile of the terrain obtained from the DEM.

One advantage of the Unity3D engine is its asset store. Among these assets, one that is very useful for obtaining DEMs, as described in this subsection, and satellite imagery, as will be described in the next subsection, is called Real World Terrain (RWT) (18). RWT is able to download and process the DEM into multiple terrain tiles with corresponding satellite images. However we still need to apply perlin noise and adjust the edges. RWT applies the satellite image as the texture of the terrain, but we will not be using the satellite image as a texture, rather as input to the next step.

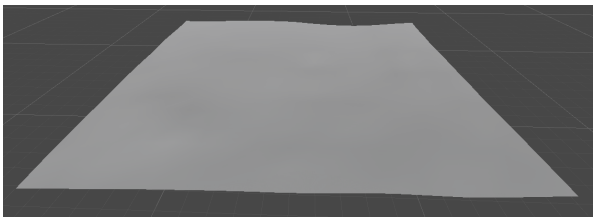


Figure 2. One tile of a processed DEM.

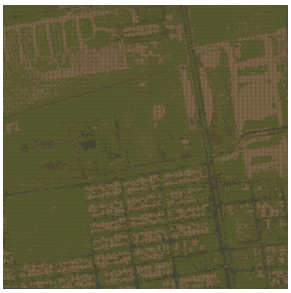
B. Satellite Imagery

Once the elevation data has been processed we now must obtain the satellite imagery, in Fig. 3(a) is shown one tile of the satellite image. There are a variety of providers which offer differing resolutions and quality regarding, for instance, shadow and cloud removal. Even the available images of higher resolution, however, are insufficient to use as texture for the terrain. In the case of a terrain of 1 km^2 size as mentioned in the previous section, a 4096x4096 sized satellite image would mean each pixel in the image amounts to approximately 25 cm. Even if managing 256 (16x16 tiles) images with such a resolution was feasible for real time performance, it is still not enough for high quality graphics, especially considering applications where the camera will be placed near the ground.

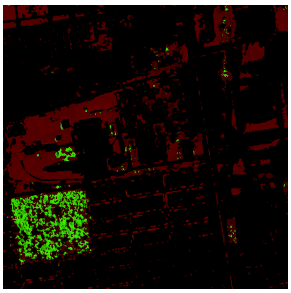
Instead, we list the different types of possible surfaces (grass, dirt, stone, asphalt, etc.) and then process the image to identify what type of surface is found on each point. For this reason a clean image (one with minimal shadows and clouds) is best. The surface values are added to a matrix called a splat map, which is used by the engine's terrain system to blend different textures. This way we are able to use high resolution textures for each surface type and tile them across the terrain according to the surface we encountered at each point in the satellite image, resulting in a terrain that retains the general surface characteristics, as shown in Fig. 3(b), but with better performance (since for all tiles we will only load the surface textures instead of 256 high-resolution satellite images) and graphical quality when viewed up close.

To process the image and identify the different types of surfaces, we first convert it from RGB (Red, Green, and Blue) to HSV (Hue, Saturation, and Value) color space. Some studies (19) (20) have shown that the use of HSV color space for image segmentation tasks can provide better results when compared to RGB. By decoupling luminance and chromaticity information, the HSV space definition gets closer to how we perceive and experience color, making the extraction of features based on color information more intuitive. Thus, by testing different thresholds, we can easily find the range for each HSV component that best identifies each surface type, generating masks indicating the corresponding

(a) Satellite image



(b) Retexturized terrain



(c) Vegetation mask

Figure 3. Processing one tile of the satellite image.

pixel locations. For instance, for grass identification, we chose pixels with a hue between 0.19 and 0.36, saturation between 0.20 and 0.80, and intensity (or value) between 0.10 and 0.80.

In addition to the textures, we used the same procedure for vegetation placement, which is the process of instancing the 3D models of grass and trees. One mask is generated for grass and one for the trees, these masks are encoded into a single image, as shown in Fig. 3(c), with grass being the red channel and trees the green channel. For the grass mask we used the same HSV filter as the grass texture filter, however to detect trees our thresholds are a bit more restrictive, since the trees usually have a darker hue than the grass. For the trees we found the best results with a hue between 0.23 and 0.36, saturation between 0.1 and 1.0 and intensity between 0.10 and 0.27. These masks are then fed into a process which randomly spawns the grass and the trees based on a set of rules in order to generate some variety.

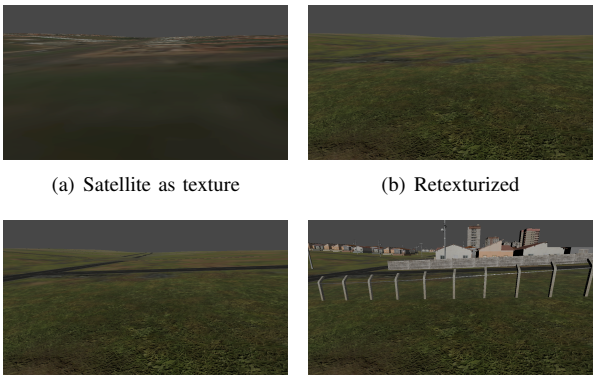
C. Road Data

Roads are another crucial aspect of the terrain. To build the road network we access Open Street Maps (OSM) (21), a crowd sourced database, which means all data is submitted by users. OSM has an API which provides data in JSON format as a list of nodes and a list of roads that refer to those nodes. Nodes with only one reference are normal members of a road, and nodes with more than one reference are an intersection node. The nodes, however, contain only 2D information (latitude and longitude), which means the height is not informed. To obtain the height we cast a ray from each 2D point downwards towards the terrain to discover the height for the nodes.

OSM data also contains certain parameters that define characteristics of a road, of which, for our application, the two most important are road width and surface type. One key issue with OSM data is that, due to being crowd sourced, not all data might be present, and when present, might not be accurate. The road width, when not supplied, is defaulted to according to the legislation of the region of interest, whereas the surface type defaults to asphalt. An improvement we would like to implement is to use the satellite image in these cases to attempt to assess the surface type.

Once the road width and surface type are defined we build the road mesh. To build the road mesh, because the OSM nodes are not evenly spaced, we first subdivide the list of nodes into an evenly spaced list of points. This also allows us to define a spacing value that will avoid the road cutting into the terrain at certain points, since the default distance can be large between nodes, of which we detected the height casting rays downwards on the terrain, and thus the terrain in between nodes might contain slight peaks and slopes. Reducing the distance between each point minimizes the chance of these peaks appearing through the road, but instead the road respecting these variations. With this new list of points, for each point we detect the direction based on the next point and obtain the perpendicular vector, which we will call side vector, by doing the cross-product of the direction vector and the upwards vector. Based on the road width we move half that width from the center point to each side and do a new ray cast to get the height of each of the two vertices to ensure any variation of the terrain is accounted for.

After all the road meshes have been created, we then slightly depress the terrain beneath all roads so that they sit neatly on the terrain without producing any artifacts due to potential Z-fighting, a phenomenon well described in (22). We currently do not treat intersections, so the roads currently overlap. With the road mesh created we are able to use another set of high definition textures for the road material, as well as detect when the user is on a road or not, via collision detection.



(a) Satellite as texture (b) Retexturized

(c) Roads (d) Final

Figure 4. Ground-level views.

D. City Elements

Finally, the last part of the automated process is inserting the city elements, such as houses and roadside objects. While OSM is also prepared to store this kind of data, the area we chose lacks this information, so we had to choose a different approach. For this part we used the road network as a basis for instancing the city elements. This last part is actually not a fully automated process because we must choose portions of the map in which we would like to instantiate the elements.

For this task we once again resort to the engine's asset system, finding an asset called CityGen3D (23), which, among a couple of other features we did not use, is capable of randomly instancing elements along the roads. We built a set of 5 different houses that match the style of the location and fed them into the system, which instantiates these houses randomly following the road network using the same OSM data we parsed in the previous step. Some abnormalities do occur since the road network for the chosen location is not as organized as most locations in the northern hemisphere, and these issues we had to manually adjust.

IV. Results

In Fig. 4 the different steps of the automated process are shown, where in Fig. 4(a) we see the terrain with the satellite image applied as texture. In Fig. 4(b) the image has been replaced by surface textures based on the automatic filtering. Afterwards, in Fig. 4(c) the roads have been placed and finally the city elements are placed, as seen in Fig. 4(d).

The layer that took longest to solve was the city layer. After many attempts at obtaining or extracting real world data from multiple sources, we decided to use a procedural generation tool to instantiate these elements. The result of this generation is satisfactory for our needs, even if it is still not fail proof and requires a more hands-on approach than the rest of the layers.

(a) Real world

(b) Real time rendered

Figure 5. Comparison of real and virtual environments.

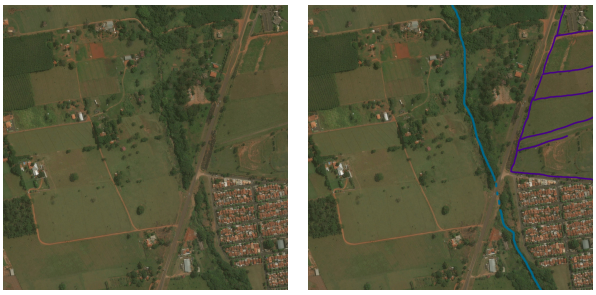
The final terrain is satisfactory for our use, which is for terrestrial locomotion. The high quality textures of the surfaces, complemented by the vegetation and roads allow us to have a great render quality and a recognizable terrain, meaning that subjects have been able to successfully navigate the terrain based on real maps, as well as encounter the obstacles and elevations that they would if performing the same actions in the real world. In Fig. 5 it is possible to see a photo of a location and a frame of the real time render of the same location in the virtual environment, in which the only manual intervention was the insertion of the football goals, the fences and the warehouse-like constructions visible on the left-hand side of the image.

Furthermore, we were able to keep the system optimized enough as to run adequately on lower-end systems based on the current standards. We have been able to achieve a sustainable 40 fps running at 720p resolution on computers with GPUs such as nVidia 720 with 8 GB of RAM. Some compromises were made in order to be able to maintain a good frame rate on sub-optimal computers. In Fig. 6 it is possible to see another real world and virtual world comparison. In this case, the vegetation density, on the right-hand side of the image, is lower in the virtual world in order to improve frame rates.

(a) Real world

(b) Real time rendered

Figure 6. Comparison of real and virtual environments.



(a) (b)

Figure 7. Manual detection of features in a satellite image.

A. *Manual Intervention*

Due to the lack of data on some types of elements, once the automated process was completed, we still needed to perform some manual intervention to insert specific elements, such as walls, fences and bodies of water. In Fig. 7 an example is shown of a tile containing a creek and fences. In purple we annotated the fences and in blue the course of a creek, found via photos of the location and complementary maps. The automatic detection of these elements is very difficult, and extracting any further information, such as the depth of the creek, is not possible only from satellite imagery, therefore these elements have been inserted manually.

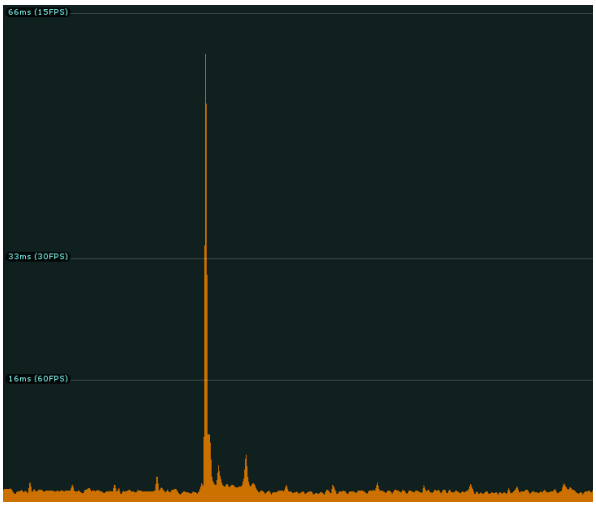


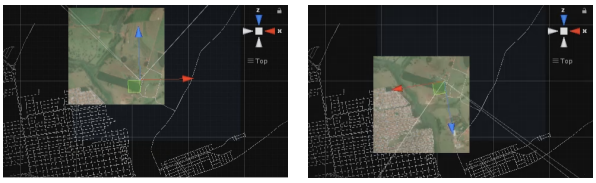
Figure 8. Spike in physics processing due to floating origin trigger.

B. *Floating origin*

Because of the size of the requested area (256 km^2), we will eventually run into a numeric precision issue, since we will need too many bytes to represent the position of objects in the world. In order to avoid this issue, we adopted a solution very similar to the "floating origin" described in (24). Unlike that solution, however, we allow traditional navigation to occur and only update the origin at specific thresholds because moving the world is a costly operation and thus not viable on a frame-by-frame basis. In Fig. 8 it is possible to see a spike in the physics processing due to updating all the physics once a floating origin is triggered. Instead we established a threshold and every time the player reaches that threshold we move the whole world, including the player, back to the origin, creating an offset. This process is only done locally, the server still tracks all objects in their original position, however the client applies a local offset to all objects.

C. *Dynamic loading*

Another issue that arises due to the very large size of the virtual environment is memory, CPU and GPU demand. In order to keep the system optimized and able to execute on standard computers with reduced loading times, we chose to not have the whole terrain loaded at all times. Instead, the system will always keep the current and neighboring terrain tiles loaded, to minimum a distance of 2 km in any direction, reading and constructing each new tile and unloading the unused ones, as illustrated in Fig. 9. This way we drastically reduced the initial loading time by about 1500%. Furthermore, due to the whole environment not being loaded in memory, we reduced memory consumption from approximately 13 GB to short of 7 GB, as well as a noticeable improvement in frame rates. This optimization comes at the cost of slight "hiccups" when transitioning



(a) (b)

Figure 9. Dynamic loading of terrain tiles.

between tiles due to the loading of a few new tiles, as can be seen in Fig. 10.

We tested three different approaches to the management of terrain tiles, based on the options available to us in the chosen engine. The first approach we attempted was saving each terrain as its own scene (25). A scene, in Unity3D, is a native data structure that has an API that allows us, in run-time, to easily perform asynchronous load and unload operations. The performance of this approach was satisfactory, however one issue was that it drastically increased build time. This is not an issue for the end-user, but it was a nuisance for the development team. We then decided to attempt creating prefabs (26), another of the engine's features. Prefabs are reusable objects that the developers create during development in order to store the complete state of an object and easily instantiate it during run-time. With this approach we were able to reduce the build time, but we had to implement a management system that would pseudo-asynchronously, by means of a feature called Coroutines (27), instantiate the tile prefabs. We felt we could still improve upon once we realised that Unity3D stores terrain data in a very similar was as it does with prefabs. Furthermore, the terrain system has a method that allows us to instantiate a terrain based on these native terrain data files. Using the aforementioned pseudo-asynchronous loading method, instead of instantiating prefabs, we constructed the terrain directly from the terrain data. This is a very slight optimization, reducing final build time and file size, but with virtually no difference in run-time performance.

V. Conclusion

It is possible to find some methods to load real world data as a terrain in the virtual world, but we believe that one key step to improve the quality of the final application was processing the satellite image into different masks, allowing us to use high resolution textures instead of the satellite image as the floor texture, as well as placing the 3D vegetation with higher accuracy. One notable feature of the region we built is that, in spite of having hills, it is not mountainous, which might require a special treatment for our texturization method. Our greatest challenge was the placement of the city elements, especially housings and buildings, because in the Votuporanga area there was no

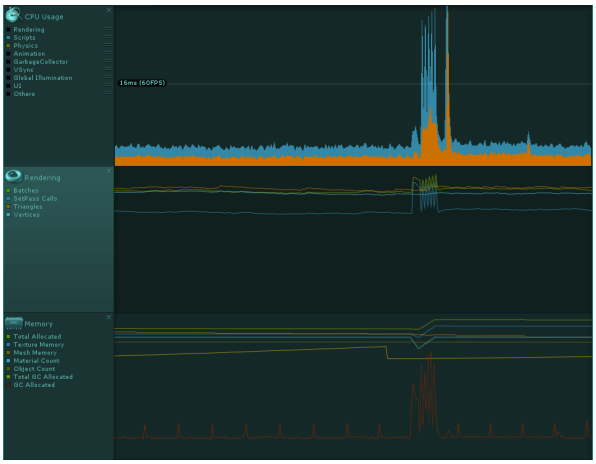


Figure 10. Spike when dynamically loading terrain tiles.

such data available. Our current solution for placing the city elements is where we expect to improve next, because even though the results are acceptable, we feel we can improve the method by increasing the algorithm's autonomy and obtaining better results.

Another step that we hope to improve is the creation of the road network, where we currently have two different improvements to introduce. By using the satellite image we can possibly assess lacking information from the data sources, namely the width of the road and the surface type. Also, we still intend to improve the intersections. We currently are able to cut the mesh of the roads that intersect, but we still haven't developed the method to rebuild the intersection mesh considering every possible entry angle and blend the potentially different surface types.

The use of a special case of the floating origin method and even more importantly splitting the terrain into tiles in order to dynamically load and unload them was key to enabling us to build a very large environment all while keeping it running at a good frame rate. We were surprised to note that we were able to run a session, as seen in Fig. 11, with 16 computers at low specs by today's standards (GeForce 760 GPU and 8 GB RAM) with such a complex environment and with high graphical quality. It is possible that we can reduce the "hiccups" when loading terrain tiles by using a new suite of features recently made available by the game engine, such as the C# Job System, the Entity Component System and Burst Compiler (28).

Overall, we believe the method explained in this paper is very suitable, especially when using open or free data sources in regions with poor data availability, such as those in less populated regions of South America. It is important to keep in mind that some degree of manual editing will still be necessary with our current methods, but for large terrains in particular it is crucial to build these automated processes in order to reduce the amount of manual labor and be able

Figure 11. Multiplayer session with 16 trainees.

to quickly create the virtual environment.

The task of creating a virtual environment that truly represents the real world is by no means an easy one. In regions such as the one we worked on, with missing, sparse, and untrustworthy data, the task is even more difficult. On the other hand, technology has evolved very much: gaming engines and third party assets have greatly improved the workflow and possibilities of what can be created, allowing us to reach a very satisfactory result. The results we achieved in the amount of time and size of development team would be unfathomable a few years ago.

Acknowledgment

We would like to thank the Brazilian Marine Corps for the support and for promptly providing the necessary information to successfully develop our research.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Alberto Raposo would like to thank CNPq for the support granted to this research.

References

[1] "America's Army," https://www.americasarmy.com/, 2019, [Online; accessed 08-July-2019].

[2] "Virtual Battlespace," https://bisimulations.com/products/virtual-battlespace, 2019, [Online; accessed 08-July-2019].

[3] "eSim Games," https://www.army-technology.com/contractors/training/esim-games/, 2019, [Online; accessed 08-July-2019].

[4] "Unity Real-Time Development Platform," https://unity.com/, 2019, [Online; accessed 08-July-2019].

[5] R. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "Integrating procedural generation and manual editing of virtual worlds," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 2.

[6] P. B. Germano, "Geração de Malhas Rodoviárias na GPU," Ph.D. dissertation, PUC-Rio, 2014.

[7] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds," in *Computer Graphics Forum*, vol. 33, no. 6. Wiley Online Library, 2014, pp. 31–50.

[8] R. M. Smelik, T. Tutenel, K. J. De Kraker, and R. Bidarra, "Declarative terrain modeling for military training games," *International journal of computer games technology*, vol. 2010, p. 2, 2010.

[9] I. Parberry, "Designer worlds: Procedural generation of infinite terrain from real-world elevation data," *Journal of Computer Graphics Techniques*, vol. 3, no. 1, 2014.

[10] W. D. Wells, "Generating enhanced natural environments and terrain for interactive combat simulations (genetics)," in *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM, 2005, pp. 184–191.

[11] J. J. Van Zyl, "The Shuttle Radar Topography Mission (SRTM): a breakthrough in remote sensing of topography," *Acta Astronautica*, vol. 48, no. 5-12, pp. 559–565, 2001.

[12] M. Rexer and C. Hirt, "Comparison of free high resolution digital elevation data sets (aster gdem2, srtm v2. 1/v4. 1) and validation against accurate heights from the australian national gravity database," *Australian Journal of Earth Sciences*, vol. 61, no. 2, pp. 213–226, 2014.

[13] D. Yamazaki, D. Ikeshima, R. Tawatari, T. Yamaguchi, F. O'Loughlin, J. C. Neal, C. C. Sampson, S. Kanae, and P. D. Bates, "A high-accuracy map of global terrain elevations," *Geophysical Research Letters*, vol. 44, no. 11, pp. 5844–5853, 2017.

[14] "NASA Shuttle Radar Topography Mission (SRTM) Version 3.0 (SRTM Plus) Product Release," https://lpdaac.usgs.gov/news/nasa-shuttle-radar-topography-mission-srtm-version-30-srtm-plus-product-release/, 2013, [Online; accessed 28-June-2019].

[15] R. E. Carlson and F. N. Fritsch, "Monotone piecewise bicubic interpolation," *SIAM journal on numerical analysis*, vol. 22, no. 2, pp. 386–400, 1985.

[16] C. Tchou, "2018.3 Terrain Update: Getting Started," https://blogs.unity3d.com/2018/10/10/2018-3-terrain-update-getting-started/, 2018, [Online; accessed 28-June-2019].

[17] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[18] "Real World Terrain," http://infinity-code.com/en/products/real-world-terrain, 2019, [Online; accessed 28-June-2019].

[19] S. Sural, G. Qian, and S. Pramanik, "Segmentation and histogram generation using the hsv color space for image retrieval," in *Proceedings. International Conference on Image Processing*, vol. 2. IEEE, 2002, pp.

II–II.

[20] F. Garcia-Lamont, J. Cervantes, A. López, and L. Rodriguez, "Segmentation of images by color features: A survey," *Neurocomputing*, vol. 292, pp. 1–27, 2018.

[21] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, 2008.

[22] A.-A. Vasilakis and I. Fudos, "Depth-fighting aware methods for multifragment rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 6, pp. 967–977, 2012.

[23] "CityGen3D," https://www.citygen3d.com/, 2019, [Online; accessed 29-June-2019].

[24] C. Thome, "Using a floating origin to improve fidelity and performance of large, distributed virtual worlds," in *2005 International Conference on Cyberworlds (CW'05)*. IEEE, 2005, pp. 8–pp.

[25] "Unity - Manual: Scenes," https://docs.unity3d.com/Manual/CreatingScenes.html, 2019, [Online; accessed 09-July-2019].

[26] "Unity - Manual: Prefabs," https://docs.unity3d.com/Manual/Prefabs.html, 2019, [Online; accessed 09-July-2019].

[27] "Unity - Manual: Coroutines," https://docs.unity3d.com/Manual/Coroutines.htmlCoroutines, 2019, [Online; accessed 09-July-2019].

[28] "DOTS - Unity's new multithreaded Data-Oriented Technology Stack," https://unity.com/dots, 2019, [Online; accessed 09-July-2019].