Final report, Blend'it project

Benjamin Boisson, Guillaume Combette, Dimitri Lajou,
Victor Lutfalla, Octave Mariotti*, Raphaël Monat,
Etienne Moutot, Johanna Seif, Pijus Simonaitis

2015-2016

Abstract

The goal of this project, "Blend'it", was to design two open-source plug-ins for Blender, which is a computer graphics software. These two plug-ins aim to help artists creating complex environments. The first of these plug-in should be able to realistically animate crowds, and the other one to design static environments by making different elements such as rivers and mountains automatically interact to produce realistic scenes.
Although this kind of software is already developed in the Computer Graphics industry, there are often unavailable to the public and not free.

*was abroad during the second semester, he did not contributed to the second part of the project

Contents

1 Introduction **2**
- 1.1 Objectives . . . 2
- 1.2 State of the art . . . 3
- 1.3 Team . . . 3
- 1.4 Choice of Blender. . . . 4
- 1.5 Structure of the project . . . 4

2 Blender exploration team **5**
- 2.1 Blender discovery . . . 5
- 2.2 Unitary tests . . . 5
- 2.3 Code coverage . . . 5
- 2.4 Code organization . . . 5

3 Crowd plug-in **6**
- 3.1 Human animation . . . 6
- 3.2 Path generation . . . 6
  - 3.2.1 Implementation . . . 6
- 3.3 Path interpolation in Blender . . . 8
  - 3.3.1 Creating a path . . . 8
- 3.4 GUI . . . 8
  - 3.4.1 The Map Panel . . . 9
  - 3.4.2 The Crowd Panel . . . 9
  - 3.4.3 The Simulation Panel . . . 9

4 Environment plug-in **11**
- 4.1 Overview . . . 11
- 4.2 Features . . . 11
  - 4.2.1 Mountains . . . 11
  - 4.2.2 Roads . . . 11
  - 4.2.3 Vegetation (forests) . . . 11
  - 4.2.4 Cities . . . 12
- 4.3 Mixing Features . . . 12
  - 4.3.1 FeatureTree . . . 12
  - 4.3.2 Environment . . . 13
  - 4.3.3 Interaction with Blender . . . 13
- 4.4 Graphical User Interface (GUI) . . . 13

5 Conclusion **15**

1 Introduction

1.1 Objectives

Blender is a complete open source 3D software, and can create 3D scenes from scratch, from modeling to animation and rendering. The goal of this project is to improve Blender, by providing two new plug-ins to help artists creating rich environments.

Our aim is to add to Blender the possibility to generate large, crowded environments. There are two independents parts: first generate environments in a semi-automated way. Secondly, generate and animate people in the environment. People's movements must be consistent with the environment and other people's movements. There are some examples of what has been created by some researchers in figures 1 and 2.

(a) Real picture (b) Animation example from [1]

Figure 1: A crossing in Tokyo

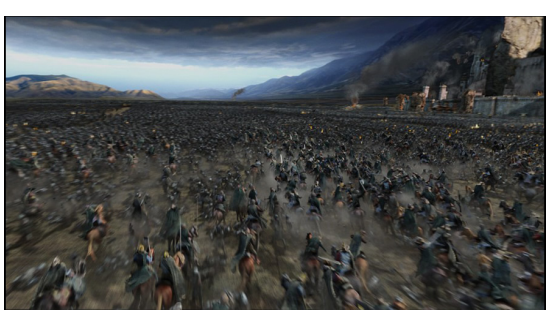
Note that Blend'it does not aim to be a realistic crowd generator, in the sense of a usable simulator for sociological or scientific experiments. The goal is to provide a complete generator for crowd and environment for an artistic purpose only, that *looks like* a realistic crowd.

Figure 2: Image taken from [2]

There have been a lot of researches around the procedural generation of crowds and environments in computer graphics. But the large majority are implemented as research prototypes or in commercial software. To the best of our knowledge, nothing exists for widely used free software like Blender. 3D graphics is a domain that is still closed-source, and every professional 3D artist uses costly software. We think it is an important point to move some proprietary technologies to the open-source world, to fill a little bit the gap.

1.2 State of the art

Animation of large crowds is really developed in the computer graphics industry, enabling commercial 3D software to easily render massive scenes with large crowds. For example, a software called Massive ([3], Fig. 3a) dramatically changed the habits of 3D studios: it helped generating the armies being displayed in *Lord of the Rings*. The Golaem software ([4], Fig. 3b) is also able to render massive crowds in different situations, and is used by the studio producing *Game of Thrones*. Nowadays, even commercial standalone 3D software like 3DS MAX [5] is able to generate realistic crowds. VUE is a software able to generate massive, realistic landscapes, as presented in Fig. 4. However, these software are really expensive: Massive costs up to $16,000 per year, and Golaem $5000 per year. On the contrary, no open-source software provides this kind of feature. In the past, scripts had been developed, but they were not providing realistic results, and are not working anymore.



(a) Scene generated using Massive

(b) Scene generated using Golaem

Figure 3: State-of-the-art crowd generation and animation

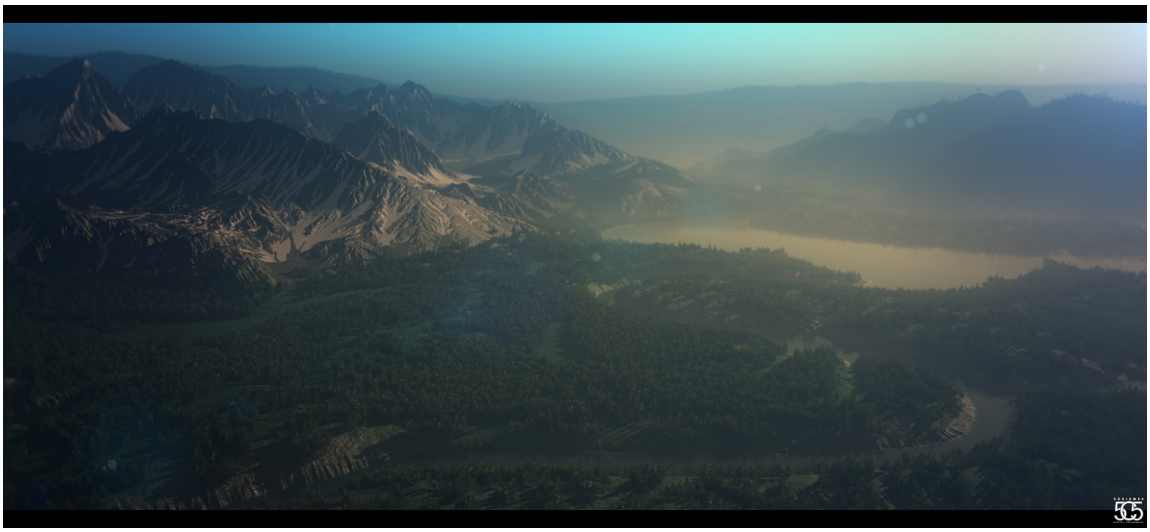


Figure 4: State-of-the-art landscape generation using VUE software

1.3 Team

We are 9 Master students from the *École Normale Supérieure de Lyon*, France. As a part of our First year of Master in Research in Computer Science, we need to develop and code a project of our choice (here, "Blend'it"), during the whole year. We are supervised by an Associate

Professor from the ENS Lyon, Eddy Caron. There is a project leader, Etienne Moutot, and a deputy leader, Raphaël Monat. Please do not hesitate to contact us, by writing an email at *first name . last name@ens-lyon.fr*.

All our work is hosted on Github, and can be consulted on `github.com/blendit`. We also have a website: `blendit.github.io`.

1.4 Choice of Blender.

The big advantage of producing a code based on Blender is that we can use its Python API, which is very powerful. With it, one can manipulate every objects and part of the Blender interface without re-compiling Blender or even modifying the source code. Thus, we can concentrate only on the new parts of the project, that is our algorithms. We did not need to code another render engine for example. The other advantage is the already existing Blender community on which we can rely to have some feedback on our project.

1.5 Structure of the project

The two plug-in are currently independent, one is able to simulate the motion of crowds, and one is able to generated environments. In each plug-in, we split our code into a "pure" part, independent of Blender, and one interacting with the Python API of Blender. This has two advantages: we can test the "pure" part easily (testing the Blender plug-in automatically is more difficult), and we may code interface with other softwares too.

Outline

- In section 2, we present the work of the Blender exploration team, done by Raphaël Monat, Etienne Moutot and Pijus Simonaitis.
- Section 3 presents the work done by Dimitri Lajou, Victor Lutfalla, Johanna Seif and Pijus Simonaitis on the crowd simulation.
- Section 4 presents the environment plug-in, done by Benjamin Boisson, Guillaume Combette, Raphaël Monat and Etienne Moutot.
- Section 5 concludes this report.

2 Blender exploration team

During the whole project, this team had two aims. Our first goal was to discover Blender and Python while other teams focused on the bibliographical work, to then teach everyone how to use Blender efficiently and code in Python. Our second goal was to guarantee the quality of our code by setting up unitary tests and code coverage tools.

2.1 Blender discovery

Blender is written in C++ and Python, but it has a powerful API in Python, so we did not have to modify the C++ core of Blender. Blender is also able to show the Python source of its interface, or give the Python command associated to a manual operation. These features were really helpful to develop efficiently our programs.

2.2 Unitary tests

For each new class we implemented, we also created a unit test file. The execution was handled by the unitary test library of Python. We linked our Github repositories with Travis, a website that runs the tests every time a pull request is created.

The advantage of creating these tests is that it forced us to test deeply our code, and more importantly, to notice when code modifications break some other parts of the code. This last feature is really important for big projects, to avoid creating bugs when modifying the code.

2.3 Code coverage

We also linked our Github repositories with Coveralls, to see what part of the code was covered by our unitary tests. To ensure a good coverage, the pull requests had to increase coverage, or they would automatically be rejected.

2.4 Code organization

This team also organized globally the code of the project. The code of each plug-in is separated into two parts: one independent from Blender, containing the *core*, that is, the main algorithms and data structures. The second part implements the interface with Blender.

The advantage of this organization is that the first part can be tested automatically using unit tests. The part using Blender cannot be tested automatically with Travis, because it needs a lot of interaction with the graphical interface. Thus, we tried to put maximum of the complex algorithm in the first part, to allow them to be tested extensively.

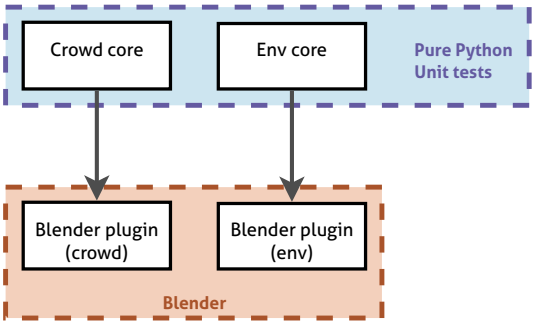


Figure 5: Code organization

3 Crowd plug-in

The goal of this work package was to implement and integrate into Blender an efficient and realistic motion of a crowd. At first we wanted to implement two features: a general movement algorithm and another algorithm that would control the animation of the crowd. The second part was quite difficult to implement and thus we decided to concentrate ourselves on the first part.

The initial goal was to implement two parts: one creating trajectories for the crowd and another one for animating individual movements. The second part was really hard to implement and was closely linked with the animation of models, which is a domain where we did not have any expertise. We thus decided to focus on the first part.

Our code is split into two parts: one that is independent from Blender and one that depends on it (see Fig. 5). The first part create a set of key points that will represent the movement of one person and the second interpolates a trajectory from those points. Before presenting these two parts, we explain the difficulties encountered when we tried to model human animation.

3.1 Human animation

At the beginning we considered animating Humans and started by analyzing a theoretical survey of computer animation of Human walking [6] and looking for what was already done in Blender concerning automatic walking. All Blender-related resources on walking animation are gathered on a web-page [7]. Walk-o-matic and stride add-ons were used in previous versions of Blender to "help interactively design rough passes of a walk" and "quickly create cycles for background or extra characters", however both were broken on Blender 2.7x (last version).

Due to the lack of existing tools concerning automatic walking we considered creating such a tool ourselves. We started by analyzing tutorials on Blender character animation (for example [8]) and creating such a motion manually. However we soon realized how difficult it is to generate a realistic walk due to the complex physics behind the movements and decided that it is out of the scope of our project. We have shifted our attention to the more basic task of creating a path in Blender and making an object following it with varying speed given by our algorithm.

3.2 Path generation

The first algorithm is inspired by those two papers: [1] and [9]. We chose them because it relied on graphs rather than cellular automaton, and we were more familiar with graphs. We also thought this approach would be more precise.

The idea is the following: a graph (grid) algorithm will generate a general path for each individual and another algorithm will prevent collisions between individuals.

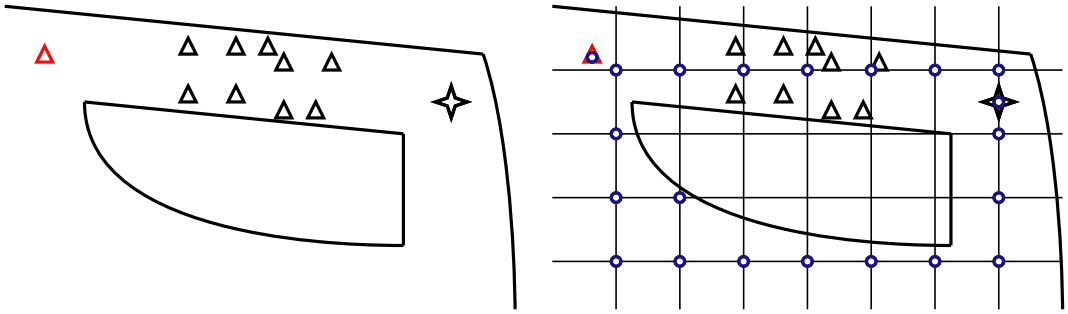
Time is discretized, and for each time, we compute the direction to go for each individual. We move the individual in this direction, and iterate this principle. This is how we get the final set of points for each individuals.

3.2.1 Implementation

We develop here the implementation of 3 features: graph-based approximation, allowed velocity fields to avoid collisions, and the final computation of the allowed movement.

The guide graph The first part that we tried is to use a graph to "approximate" the space to compute trajectories. We created a data structure representing nodes and edges of the graph. This was quite easy to do since the graph is supposed to be a grid, it is regular. Then

we needed a minimum distance (1 to 1) algorithm on the graph. For that we chose the A^* algorithm. Unfortunately, our implementation of the A^* was really costly in time and even more in memory. This fact forced us to abandon the graph in the rest of the development. We could test it for small values, and for a small number of calls, but the A^* procedure would have to be called thousands of times which makes this impractical. To replace the graph we used the euclidean distance, but we had to remove static obstacles. Indeed these obstacles were taken into account in the graph (vertices are removed inside the obstacles) and impacted the shortest path algorithm A^* but euclidean distance cannot take into account such obstacles. Also the points (representing people) were not able to avoid packed (with people or obstacles) places anymore and just went straight to there goal. Packed places were taken into account by the edges of the graph which weight depended on the crowdedness of the neighbour cells.



(a) A setting in a crowd animation (b) The graph-based version

Figure 6: The idea of the graph

Allowed velocity field This part deals with collision avoidance, it is the part that took us the more time to implement. The problem is the following, you have a set of individual (i.e. points) with current velocities. You want for each individual a set of velocities that will ensure that if we pick one in it then we will not collide with another individual on the way (see Fig. 7 for an example). To do that the algorithm makes a lot of geometrical computation as explained in [9]. We used the Python library Shapely to represent geometrical structures.

This library has some very useful tools but some functionalities did not work very well with floating-point numbers and the small approximations that they entail. The errors linked to the floats are one of the main reason it took us so much time.

The other was that some geometrical shapes were hard to represent and to compute. Also due to the absence of the guide graph, the individuals were not able to go around obstacles so it induced bugs, the points tends to get closer and closer to the limits of the obstacle until they cross it through errors and thus go through obstacles. In the end, this part does return collisions free velocities in 95% of cases. There are still some bugs that we were not able to fix.

Computation of the allowed movement This part involves minimizing a function on the velocity field of each individuals: each individual will choose a movement minimizing the energy used to get to his/her goal. For that we had two options. The first one was to implement a simplex algorithm but the graph did not yield linear constraints so this was not a valid way to minimize our function. The second method was to choose an angle and increment it according to a $d\theta$ and minimizing the objective function (either a graph-based distance or the euclidean distance, corresponding the energy used) on those angles.

This part is fully functional but relied on floating-point numbers again, involving computational errors.

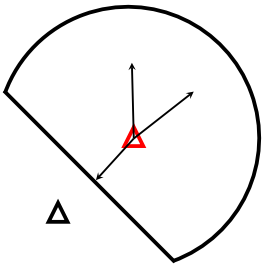


Figure 7: Allowed velocity field

3.3 Path interpolation in Blender

The algorithm described in the previous paragraphs outputs individual's coordinates at every time step and from these we have to interpolate a continuous path in Blender.

3.3.1 Creating a path

There are two ways to make an object move in Blender

1. Fix where an object should be at a given time and then modify the interpolation of the movement to make it realistic.

2. Use Blender structures for paths (Bezier curves, NURBS curves) and various ways to couple an object and a path (*Follow Path* Constraint, *Clamp To* Constraint).

All of these options generate movement, however our work was to find the one that could be automated easily, that would be compatible with a data structure given by our algorithm, that would be precise and easy to modify for the user afterwards.

The first option was compatible with our data structure. However, realistic interpolation of the path and the possibility to modify a path afterwards posed us a lot of questions. We have chosen the second option which is a more conventional one, makes a clear distinction between a path and an object following it and is easy to modify for an artist afterwards.

The main problem to solve was a discrepancy between the data structures: position of an object on the path (Bezier or NURBS) is given through its distance along the path from the starting point while in the data structure given by our algorithm position of the point is accessed through its coordinates. This being said we had to find a way to link a position in the space to the length of the path from the starting point to that position. Blender 2.76 having no add-on to measure the length of the path made this task more complicated as we had to familiarize with the mathematics behind the interpolation of Bezier curves and NURBS.

We have chosen Bezier curves instead of NURBS because their mathematical properties allowed us to guarantee that an individual will be at a given place at a given time while NURBS interpolates a path that only comes close to the given points but not necessary passes through them.

3.4 GUI

The approach we chose was to have a GUI that would be completely integrated in the Blender GUI.

There are three layers of GUI in Blender: windows, banners, and panels.

We chose to make panels for simplicity reasons and we put those in the right banner of the 3DVIEW window.

Once we mastered the making and integration of panels with simple functionalities in Blender we started to separate the functionalities we wanted for the GUI of the crowd plug-in.

3.4.1 The Map Panel

The first panel we decided to create was one to create the map and grid on which the crowd will evolve.

We created it with four main functionalities:

- save and load the current map
- adjust size and origin of the map
- adjust grid size (crucial for the algorithm)
- add exclusion zones

3.4.2 The Crowd Panel

The second panel provides three of methods to create a crowd:

- save and load
- initialization with default settings
- initialization with random settings

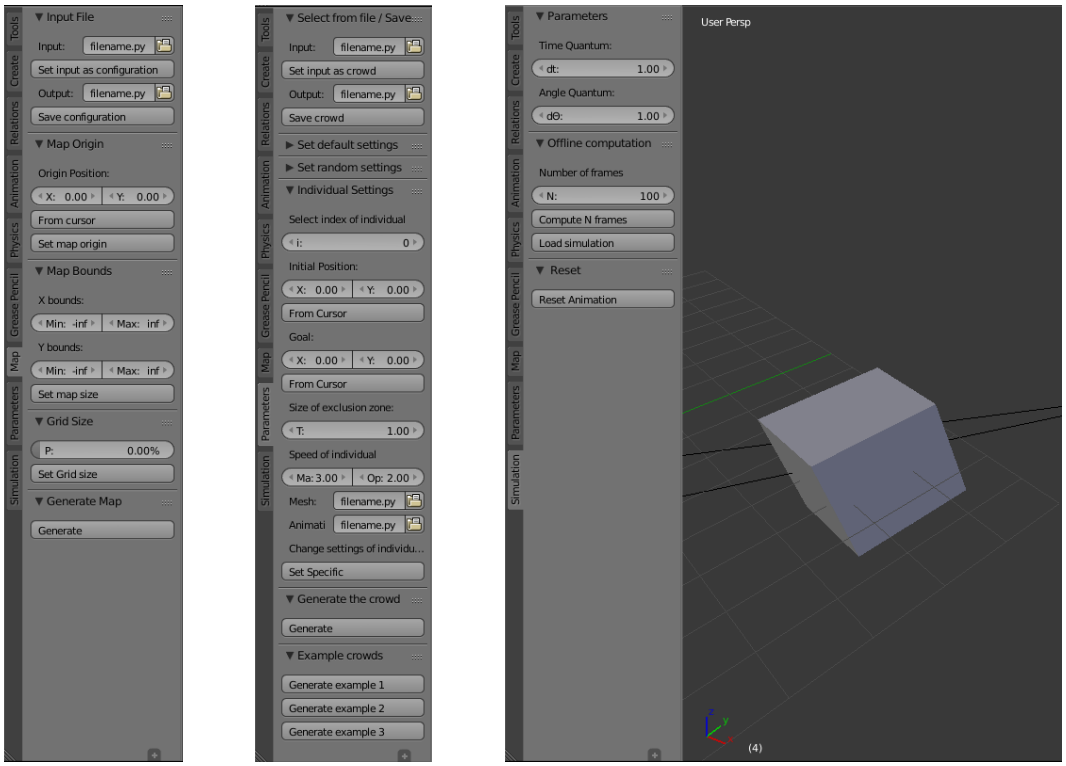
We also chose to add a customization feature: from an existing crowd (initialized with one of the three methods above) the user can select an individual and change all of its settings: initial position, goal, size, optimal speed, maximal speed and animation of the individual.

3.4.3 The Simulation Panel

The last panel allows the user to launch the computation of a crowd animation and to render it in Blender.

The user must first set the time quantum, number of time quantum to be computed and angle quantum and then the user can launch computation and load the resulting animation into Blender's 3DVIEW.

There is also a save and load functionality for the animation.



(a) Map panel (b) Crowd panel (c) Simulation panel

Figure 8: The three panels of the GUI

4 Environment plug-in

4.1 Overview

The architecture of the plug-in comes from [2], to provide an intuitive way of building an environment: by drawing features on a map. The most difficult part in this approach is that intersecting features are allowed and should give a result consistent with user expectations.

As a starting point, we focused on generating a height map, using the concepts from [10]. It had the advantage of being able to merge different environments (mountains, cities, ...) smoothly whenever they intersect, rather than using the more convoluted conflict-resolution mechanism of [2].

The basic unit managed by this plug-in is a *feature*, which lies in a certain area, has a certain height profile and knows how to interact with the other features that may intersect it. From the collection of all the features drawn on the map by the user, our plug-in builds a tree encoding how to build the final height map by successively merging sets of features together.

4.2 Features

Features are individual "areas". Each feature contain several pieces of information:

- The heightmap on it (the height of the terrain on every point of the feature).
- Models that can appear on the feature (trees, buildings, ...).
- How the feature interact with other features when intersecting

We now present some features we implemented.

4.2.1 Mountains

Our first implementation of the Mountain feature corresponded to what is presented in [10], where mountains are generated procedurally. However, the 2D random functions we tried did not yield satisfying results. That is why we switched to another version, where we import at random a part of a heightmap taken from a website [11]. This website import the heightmaps through satellite imaging.

4.2.2 Roads

Road are polylines with constant height. It means that a road is a succession of contiguous segments, and the height of the feature is the same constant everywhere on the road.

4.2.3 Vegetation (forests)

Vegetation takes a model as input (as well as the polygonal shape). It duplicates this models randomly over the shape. The height is zero everywhere.

Last version allow to set a list of models for one forest instead of only one model.

4.2.4 Cities

We planned a feature that could generate cities, following a three-step process. First, draw the street network. For this task we chose to use the method of [12], which provides a more or less intuitive way for the user to control the output. Then, divide each street-delimited block into parcels. We chose to follow the approach of [13], which seemed to give realistic results. Finally, put a building in each parcel. We did not look much into that last part because using buildings of random height was thought to be a reasonable approximation.

However, even the first step turned out to be challenging to implement. The principle of the chosen method was to describe the shape of the street network by a tensor field, mapping points to 2×2 matrices. It is described by a combination of basic fields given by the user. The streets are then drawn as the stream lines of the eigenvectors of the field. The main issue here was to represent the street network under construction so that:

- street drawing starts from points belonging to old streets and satisfying certain constraints
- new intersections are correctly handled while drawing a street
- the right eigenvector is always chosen

without having to reimplement the computation of eigenvectors or the numerical approximation of a differential equation.

4.3 Mixing Features

Now we want to mix these features to be able to generate full environments.

For that, we first generate a *Feature Tree*, to build a hierarchy of the different features. Using this tree, we generate a heightmap and list of models of the environment (a model is a 2D position on the map and a path to a 3D model). And with all these elements, we finally create the Blender 3D scene.

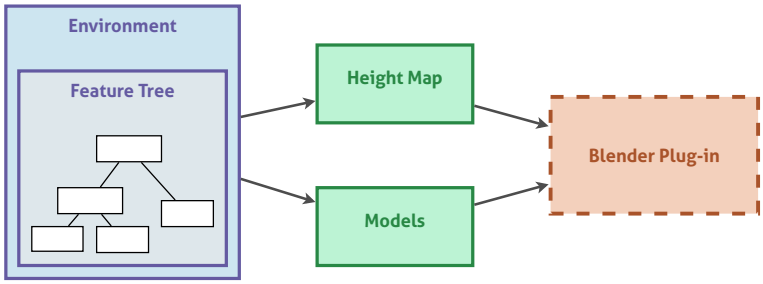


Figure 9: Organisation of the Environment plug-in

This algorithm does everything without knowing what the feature are. Thus, it can mix any type of features. The user can define its own feature because the algorithm is general enough to handle any features.

4.3.1 FeatureTree

This data structure takes as input a list of features, and outputs a structured tree of the features, according how they are suppose to interact.

The height of a point on the full map is just a recursive call of the height in the tree (see Fig. 10).

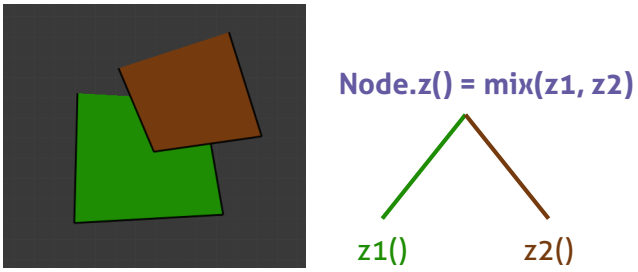


Figure 10: Tree for height computation

Every feature has 3 different type of interaction with other features:

Blend: the intersection of the two features is the mean of the two features (mix = $\frac{z1+z2}{2}$). This is what we use for mountains.

Replace: the feature erase all features it intersects with (mix = z1). This interaction is used for lakes, and rivers.

Addition: the feature adds it height to features lower in the tree (mix = $z1 + z2$). This kind of interaction is used for forests and roads.

When two features intersects and have different fusion modes, the priority is as follows:
Replace > Addition > Blend

The tree is generated by iterating over all feature intersections, and generating appropriated nodes for the 3 different possible interactions. Thanks to the library *Shapely*, geometric manipulation for generating intersections of polygons, were quite easy to code.

4.3.2 Environment

Now that the FeatureTree is generated, it is easy to generate the heightmap: for every pixel of the heightmap, compute the height of this points. Heights are then normalized and written into a png file.

For the models, we look at all models inserted by "individuals" features, and we add them to the global list of models, removing those which are "erased" by *Replace* features. Then the height (z position) of every model is computed, thanks to the previous heightmap and the (x,y) position of every models.

4.3.3 Interaction with Blender

Now we have everything to create our environment! The Blender plug-in uses previous data structures to generate a plane and applies to it the heightmap. It then import all models and insert them in the scene using their (x,y,z) position.

All parameters of the features and feature tree are modifiable via a graphical interface.

4.4 Graphical User Interface (GUI)

A picture of the interface is presented in Fig. 11. Using the interface, the user can choose the feature he wants to draw. He can then draw it using a pencil integrated in Blender, drawing polygons here. After having completed the drawing of the features, the user can ask for the generation of the environment. It can then hide the polygons, and also modify parameters that are feature-specific.

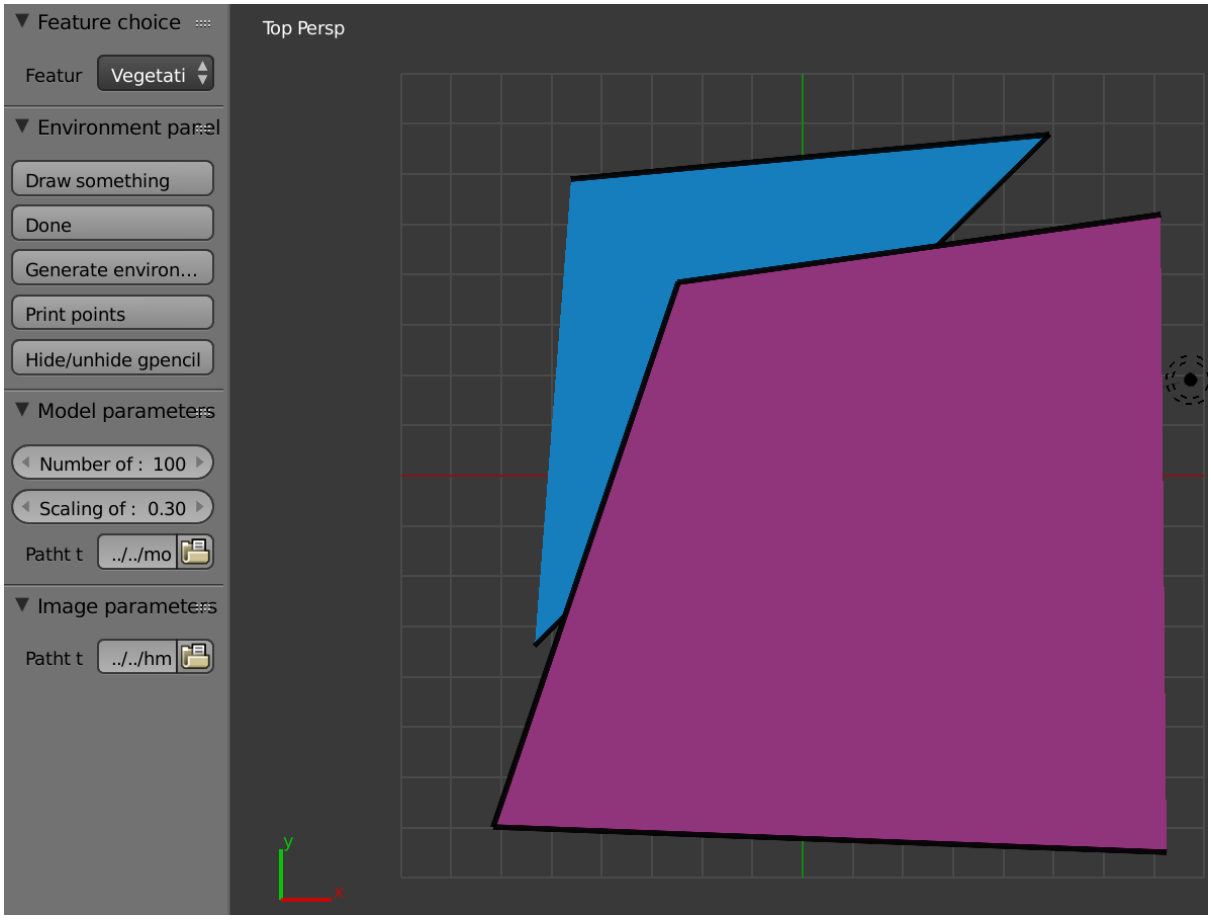


Figure 11: GUI of the Environment plug-in, with two features (in blue and magenta)

5 Conclusion

At the end of this project, we were able to render the following animation using our two plug-ins: `blendit.github.io/demo.mp4`. Our global proposal is mainly respected: we are able to generate environments provided by the user in an intuitive way, and we are able to generate crowds where people avoid each other. Thanks to a lot of planning, and to the unitary tests, we did not have any tricky bug to fix during the development of our project. Still, we have been slowed down by some difficulties: we did not realize the animation of individuals would take so much time; we understood the graph implementation was too computationally expensive too late to try another approach; we also did not had the time to develop as many features as we wished concerning the generation of environments.

Future work

Our implementation is functional, but can be improved. Among possible improvements, we could try to make both plug-in interact more. Concerning the crowd simulation part, it would be interesting to develop an efficient version of the control of the path using a graph, and maybe add options to create way-points. Another thing to do is to create an animation for the individuals. Concerning the environment generation, the city feature is not ready yet, and live modifications (going backward to edit a feature) are not supported.

References

[1] S. J. Guy, J. Chhugani, S. Curtis, P. Dubey, M. Lin, and D. Manocha, "Pledestrians: A least-effort approach to crowd simulation," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, (Aire-la-Ville, Switzerland, Switzerland), pp. 119–128, Eurographics Association, 2010.

[2] R. M. Smelik, T. Tutenel, K. J. De Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Comput. Graph.*, vol. 35, pp. 352–363, Apr. 2011.

[3] M. Software, "Massive software." `http://www.massivesoftware.com/`. Accessed: 2015-16-12.

[4] G. Crowd, "Golaem crowd simulation for maya." `http://golaem.com`. Accessed: 2015-16-12.

[5] A. Software, "3ds max autodesk software." `http://www.autodesk.fr/products/3ds-max/overview`. Accessed: 2015-16-12.

[6] F. Multon, L. France, M.-P. Cani, and G. Debunne, "Computer Animation of Human Walking: a Survey," *Journal of Visualization and Computer Animation*, vol. 10, pp. 39–54, 1999.

[7] B. Foundation, "Blender wiki – ressources on animated walks." `http://wiki.blender.org/index.php/Dev:Ref/Requests/Animation2.6`. Accessed: 2015-16-12.

[8] S. Lague, "Video tutorial: Blender character animation: Walk cycle." `https://www.youtube.com/watch?v=DuUWxUitJos`. Accessed: 2015-16-12.

[9] J. van den Berg, S. J. Guy, M. Lin, and D. Manocha, *Robotics Research: The 14th International Symposium ISRR*, ch. Reciprocal n-Body Collision Avoidance, pp. 3–19. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[10] J.-D. Génevaux, E. Galin, A. Peytavie, E. Guérin, C. Briquet, F. Grosbellet, and B. Benes, "Terrain modelling from feature primitives," *Computer Graphics Forum*, vol. 34, no. 6, pp. 211–227, 2015.

[11] `http://terrain.party`. Accessed: 2016-08-05.

[12] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," *ACM Trans. Graph.*, vol. 27, pp. 103:1–103:10, Aug. 2008.

[13] C. A. Vanegas, T. Kelly, B. Weber, J. Halatsch, D. G. Aliaga, and P. Müller, "Procedural generation of parcels in urban modeling," *Comput. Graph. Forum*, vol. 31, pp. 681–690, May 2012.