Approximate Instancing for Modeling Plant Ecosystems

Albert Garifullin [1,3], Vladimir Frolov[1,2] and Anastasiya Khlupina [2]

[1] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[2] *Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, Moscow, 125047, Russia*
[3] *Gaijin Entertainment, https://gaijinent.com, Snezhnaya st., 26, Moscow, 129323, Russia*

Abstract

Modeling and rendering large scenes with thousands of plants is still a challenging problem. Geometric models of individual plants consist of millions of triangles each and their complexity must be reduced in advance to make real-time rendering possible. Existing solutions usually implemented as a part of plants generator, make an ecosystem simulator an indivisible all-in-one solution which is hard to modify and integrate. The proposed algorithm performs approximate instancing over a set of plants represented with a specific structure. Groups of structurally similar branches are replaced with instances of one of them during the clustering process. Also, a new fast and universal procedural plants generation method is proposed. This algorithm collects statistics of spatial distribution of branches in the original set of plants and creates new plants trying to imitate parameters from original ones using instances of existing branches. Our generator is able to amplify the amount of plants in the ecosystem with small time and memory overhead. Unlike most existing algorithms the whole process is independent from the original plants generator in our solution.

Keywords

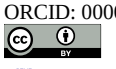
Vegetation, ecosystem simulation, plant modeling, approximate instancing

1. Introduction

Generating and rendering of complex scenes with vegetation is an important and challenging task with applications such as computer games, landscaping and architecture visualization. Procedural modeling is an interesting and widely used approach to synthesize a large variety of vegetation. The use of procedural modeling techniques can help graphics artists to create complex scenes in less time than if they were completely shaped by hand using 3D modeling software. Procedural modeling is not intended to fully replace the work of artists, but to collaborate with their productivity and get more diverse and detailed results.

Procedural generation has been studied for decades, but despite this, there are a lot of challenges in implementing this approach in real-time applications. The most important are three of them. First is *creating procedures and algorithms* that synthesize realistic and detailed models, which is basically the main task of procedural modeling itself. The second challenge is *controllability*: with the abstraction provided by procedural models, artists should retain an acceptable level of control over certain details of the generated scenes. The third challenge is *managing the large amount of data* that is generated by procedural algorithms. This is a less studied problem and the main focus of this research.

Most procedural algorithms assume that all needed data should be prepared before the rendering process. In modern applications we usually need detailed models with hundreds of thousands of triangles and large scenes with thousands of models, so every algorithm intended to create procedurally large scenes faces a memory consumption problem. The data should be processed so it would be possible to store it on a user's hard drive and at the same time should remain in form easy for rendering. Existing ways to solve this problem are described in the related works section. Most of them reduce a

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia.
EMAIL: albgar-14@yandex.ru (A. Garifullin); vfrolov@graphics.cs.msu.ru (V. Frolov); nastya_75@mail.ru (A. Khlupina)
ORCID: 0000-0003-3802-1774, (A. Garifullin); 0000-0001-8829-9884 (V. Frolov); 0000-0003-0622-2512 (A. Khlupina)


large set of unique plants or their parts to a relatively smaller set by replacing similar models with instances of one of them [2] or generate plants from a predefined set of pieces [4],[5]. These algorithms are able to dramatically reduce the amount of generated data but all of them are generator-specific. This can be a challenge when it comes to implementing such system in some complicated application, such as computer games. Modern approaches usually lead to ecosystem modeling and rendering as a solid and indivisible process which is performed by a large and highly interconnected system ([5],[9]). In practice, it usually means that game developers should re-create this system as a part of their game engine to make a modeled ecosystem match specific to game logic.

This paper demonstrates how some essential parts of ecosystem modeling can be separated from others and work as independent modules. To show the capabilities of this system, results of its work with two different generators are presented. The first one is created especially for this project, while the second is taken from a third-party open-source project.

2. Related works

According to Hart [6], [7], applications that use procedural synthesis of geometry may be classified into two paradigms: data amplification and lazy evaluation.

Applications that follow the data amplification paradigm synthesize all geometry before the rendering process. Recent surveys of procedural generation [11], [12] show data amplification is mostly used in the industry. According to them, preparation of scene with procedural generation usually consists of two separate steps: plants generation and placing in a landscape. With this approach individual trees are generated using an offline process: either procedural generation or manual generation by an artist. There are many frameworks for generating models of individual trees, such as SpeedTree [1], Xfrog [15] or SideFX TreeGenerator [16].

Then large scenes are created by placing instances of tree models. As a result all the trees in forest originate from a relatively small number of models, usually with precomputed LODs (we call this “tree library” further). Such tree instances can be placed manually or based on some spacing algorithm, but they are often repeated in an unrealistic fashion due to few unique trees in the original tree library. The number of original trees in this approach is very limited because of data size needed to represent a single tree, which is at least several megabytes. Due to this limitation, placed trees do not correctly match the surroundings, especially in complex urban scenes with many obstacles. Even modern complex placing algorithms, such as [17], do not correctly represent the plants growth response to their environment.

There are few approaches that are able to generate complex and diverse forest scenes with acceptable memory footprint and do not have limitations mentioned above. Deussen et al. in [2] developed a system for realistic modeling and rendering of plant ecosystems. In this work individual plants are generated using a procedural approach. Then the geometric complexity of the scene is reduced by **approximate instancing**, in which similar plants, groups of plants, or plant parts are replaced by instances of representative objects before the scene is rendered. A simple example of approximate instancing is shown on Figure 1. Assuming that the characteristics of each plant are described by a vector of real numbers, a clustering algorithm is applied to the set of these vectors in order to find representative vectors. There are a number of complex ecosystem generators based generally on the same concept. For example, in [13] a method for simultaneous generation of several similar plants with shared data is proposed. Another approach is described in [4]. This paper presents a method to create plant models from a finite set of pieces and a set of rules that define the matching possibilities of the pieces. In [5] this concept is implemented in an ecosystem simulation system. A set of branch modules is generated from predefined prototypes, and modules are combined to represent tree structure at each growth step. Authors simulate the growth of each module and plant as connected sets of modules with morphological and physiological parameters.

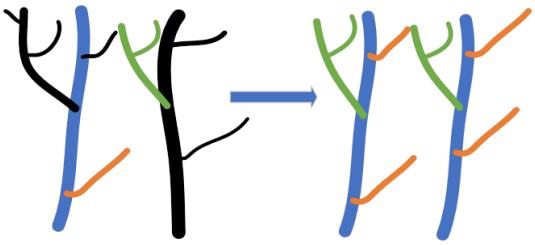


Figure 1: Simple example of approximate instancing. Black branches are replaced with instances of colored ones.

These *ecosystem generators* are able to create realistic scenes, but take a lot of time and memory. More importantly, such systems are very complex and interconnected, which makes them hard to modify and even harder to integrate in any project. In the case of video games, such ecosystem generator needs to be re-implemented as a part of the game engine.

Another possible solution was proposed by Kenwood et al. [3]. In this paper L-systems are used to generate individual trees. The L-system grammars are algorithmically modified to use instance cache for tree branches. Instances can represent a range of structures, from a single branch to multiple branches or even an entire tree. This method was implemented by B. Carey [14]. Despite the significant reduction in the amount of required memory, L-system grammars for tree generation is a restricted approach. They produce rather simple and not very realistic models as they don't take into account many biological factors and tree interaction.

With lazy evaluation real-time visualization of large procedural scenes is possible without the need to store a large amount of data. For this, the procedural synthesis is performed on demand only when the system needs data. Unlike data amplification, which is a sequential approach, lazy evaluation usually follows an asynchronous client-server architecture. Hart [6], [7] proposed a scene graph technology called Procedural Geometric Instancing (PGI) that employs lazy evaluation.

Also, Predictive Lazy Amplification method was proposed in [8]. It is a combination of data amplification and lazy evaluation paradigms. The described system is available to generate large scenes with relatively small memory usage. However, the lazy evaluation for plants rendering is uncommon in modern real-time graphics applications because of the need to create a specific separate graphics pipeline for it and all the challenges to make it consistent with other rendering processes.

The problem of managing large amounts of data taken from procedural algorithms is the less studied part of plants ecosystem generation. It is rarely discussed separately which leads to the fact that common solution for it does not exist. All current approaches are specific and exist as an addition to the procedural plant generator. As a result, data compression algorithm needs to be re-invented in every new ecosystem simulator.

3. Proposed method

3.1. General pipeline

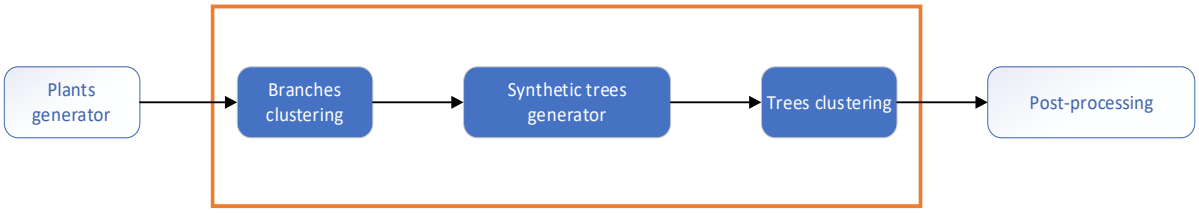


Figure 2: General pipeline

Figure 2 shows a general pipeline using proposed plant data compression and amplification algorithms. It consists of 5 main steps. At first, some procedural plants generator creates a set of plants in a specific format used for clustering, it gets some input data based on its own requirements. Some other things, such as terrain generation can be done on this step too. Then “raw” plants data input

processed by branch clustering algorithm and a single set of branches from all plants is created and then divided into several clusters – groups of similar branches. This process is based on the branch distance function, which represents the similarity of two branches. Next, for each cluster a center is found. Branches in centers form a set of basic branches and all others are replaced with instances of them. This set of instances is then amplified in a module called "synthetic trees generator". It creates new trees similar to original ones, constructing them from instances of basic branches. Then amplified data is used for the second clustering step, where the same algorithm is applied to whole trees. This step is optional, but very useful for impostor generation. The final step is post-processing, which is expected to include impostors generation and preparing simplified geometry for rendering with different levels of detail.

The proposed algorithm contains only the steps 2-4 while the first and the last steps are performed by external modules. Clustering and synthetic tree generation will be described in sections below. Impostors generating together with LOD system and effective rendering are not in focus in this paper, but as most commercial applications would have it, our implementation contains these algorithms to prove that it is possible to generate and render different levels of details for proposed plants structures.

3.2. Clustering and approximate instancing

The generator provides a set of trees with some specific structure. This structure represents plant hierarchically. In this section and further we will assume that **branch** is a structure, composed of segments, joints, leaves, and all of them are from child branches of all joints and from child branches of child branches and so on recursively. The branch itself with no child branches would be named **main stick**. Figure 3 shows structure of a branch.

Each branch has type and level, if branch B has level N, then all child branches of its joint will have level N+1. Type is a natural number, describing some generator specific set of features, needed for rendering and further processing. Usually, all branches of a tree have one type representing plant species. The plant itself is a level 0 branch.

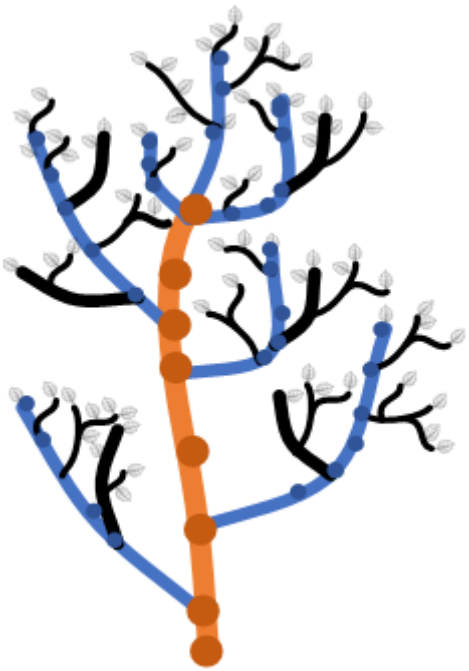


Figure 3*:* Branch structure

The memory amount needed to store even a thousand high-detailed trees is unacceptable for real-time applications, so the data should be compressed significantly. The clustering algorithm, described in this section, is able to construct a set of basic branches and instancing data from given plants. The instances of these basic branches will form plants with look and structure very similar to original ones.

This process is called approximate instancing. Unlike most existing solutions, the described algorithm is able to perform it based only on a given plant structures, without knowledge about how the generator works. The clustering algorithm consists of 4 main steps as shown on Figure 4.

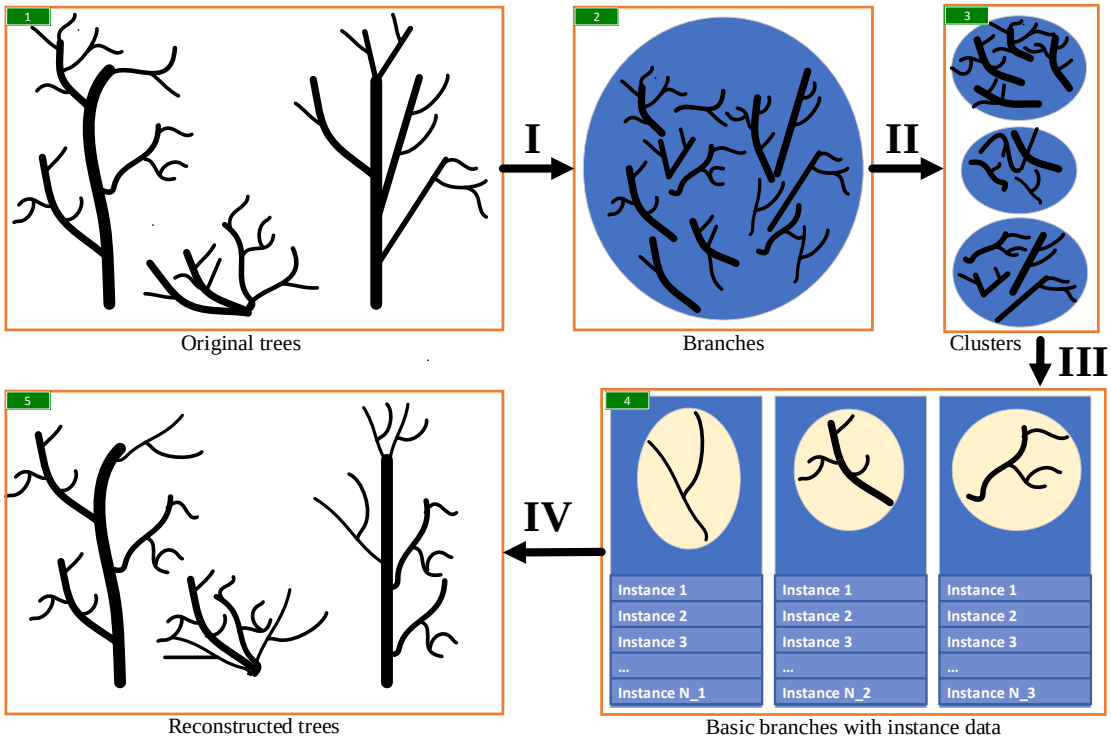


Figure 4: Clustering process. Original set of trees is decomposed to the set of branches (I). Then this set is clustered (II). A basic branch is taken from every cluster and all others are replaced with its instances (III). Finally, a set of trees can be reconstructed from basic branches and instance data (IV).

3.3. Branch distance function

Clustering is an abstract procedure that can be applied to any set of objects as long as we know distances between them. In the next section the branch distance function is described.

The branch is called normalized if it has an axis aligned bounding box with unit size and its main stick is co-directional with x axis.

Let ***B*** be the set of all normalized branches, then

$$d : B\ x\ B \rightarrow [0,1]: d(a,a) = 0, \qquad d(a,b) = d(b,a)$$

is a distance function. Define simple distance function first:

$$d_s = 1 - a * m_{struct} + (1-a) * m_{spatial},$$

m_{struct} describes the level of structural similarity of branches. It is based on a joints matching process. Assume that we have two main sticks of normalized branches $\mathbf{S_1}$ and $\mathbf{S_2}$. Let's create pairs of joints (one from the first stick, second from another) that have distance <= $\boldsymbol{\delta}$. Then he will choose pairs so that the sum of the distances between the joints in them is the smallest, and each joint is included in no more than one pair. The leaves are also taken into consideration. We do not make a pair of two joints if one of them has a leaf, and the other does not. Give each pair a weight $w \geq 0$ based on the difference of stick thickness in joints and branches levels. If $i = (S_1.level, S_2.level)$ $then$ $w \leq f_i$, f_i – constant, level importance.

Then the comparing process is continued with child branches belonging to matched joints. If there are more than one of them, we look over all possible pairs and choose the best fitted ones. *Figure 5* illustrates this process – first, the main sticks joints are mapped, then its child branches are compared. Finally, we got a set of matched joints pairs and their weights for them and sum them all. Then calculate the maximum weight sum.

$$W = \sum_i w_i, \qquad W_{max} = \sum_i n_i * f_i,$$

n_i – is a number of joints with $level = i$ in both branches, then $m_{struct} = \frac{W}{W_{max}}$.

To calculate spatial similarity level, a density field for each branch is calculated. Density field is a 3d array where each cell represents some region in the branch bounding box and the value shows how many joints and leaves are inside this region. Exact weight of each joint and leaf depends on its size and some predefined constants. $m_{spatial}$ is a Normalized Mean Square Error (NMSE) between density fields of branches. To calculate d from d_s we rotate one of the branches around its main stick and find $d_s = d_s(\boldsymbol{\alpha})$ for different angles $\boldsymbol{\alpha}$. Finally $d = d_s(\alpha)$.

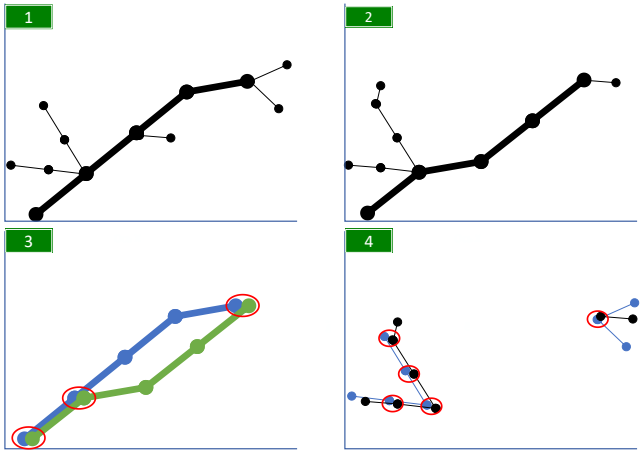


Figure 5: nodes mapping for structural distance calculation. (1), (2) – normalized branches to be compared. (3) Node mapping of main sticks. (4) The same process on child branches from mapped nodes

3.4. Implementation

The branch distance function described above is a mainly mathematical abstraction, but with some optimizations it can be implemented rather effectively. The main idea is to use only a fixed set of rotations (30, 60, 90 degrees etc.) and create density fields for each rotation before clustering process.

Also, it's obvious that we should never put very dissimilar branches in one cluster, which means that distance, as soon as it exceeded some limit d_{max}, means "branches are not similar at all". In implementation of this algorithm, possible distance is estimated during the calculation. As soon as the estimation of minimal distance exceeds the limit, the process is finished.

The implemented algorithm is also able to handle branches with different bark and leaves textures in one cluster. Id's of needed textures are saved for every instance in the cluster and then used by renderer. The only limitation is that we assume leaves in all original trees to be two-sided quads with semi-transparent texture and the same size.

Branch distance calculation algorithm is implemented on GPU using OpenGL compute shaders.

All needed data is prepared on the CPU side and put into several Shader Storage Buffer Objects. Compute shader's threads perform the algorithm described above and fill the buffer with distance between all branches (distance table). Using this distance table hierarchical clustering is performed on the CPU side. Assume that we have N branches $\{B_1 \ \ldots \ B_N\}$ and table of distances $\{d_{ij}\}$ between them. Then hierarchical clustering algorithm creates several sets (levels) of clusters – on first level there are N clusters $\{B_1\} \ldots \{B_N\}$ and the next level is created from the previous one by merging two closest clusters in one. The process stops when it is impossible to merge clusters without having dissimilar branches $(d \geq d_{max})$ in one of the clusters. To calculate distance between clusters from d_{ij}, *Ward distance* is used. The clustering algorithm and its theoretical basis described in [10].

The overall complexity level of this algorithm is $O(n^2)$, where n is the number of branches for clustering. The Table 1 shows time needed for clustering for different n. The compression ratio = *number of branches / number of clusters* is the main indicator that shows how much clustering can reduce the amount of memory required. It highly depends on d_{max} parameter. With high value almost

every number of branches can be packed in a rather small set of clusters, with low value the compression ratio is nearly constant. Table 2 represents compression ratio for different d_{max} values and different number of branches.

Table 1
Clustering time

Number of branches	Clustering time (seconds)
1250	11.86
2500	49.97
3750	111.09
5000	195.31
6250	291.78

Table 2
Compression ratio of the proposed method

Number of branches	$d\_max = 0.6$	$d\_max = 0.65$	$d\_max = 0.7$	$d\_max = 0.75$
1250	31.6	70.6	133.4	240.2
2500	30.2	83.5	192.6	357.7
3750	34.2	98.0	266.0	532.0
5000	34.6	103.2	275.1	550.2
6250	35.4	107.4	322.4	680.8

Figure 6 and 7 shows the same small group of trees with different number of clusters. There is no simple answer for what d_{max} is optimal, lower value results in more diversity with larger memory requirements. To measure visual quality decrease after clustering, NVidia FLIP [23] and SSIM [24] metrics are used. The images of clustered trees group were compared with images of the original group. Images from different points were taken and then averaged. Table 3 shows FLIP and SSIM values for different d_{max}.

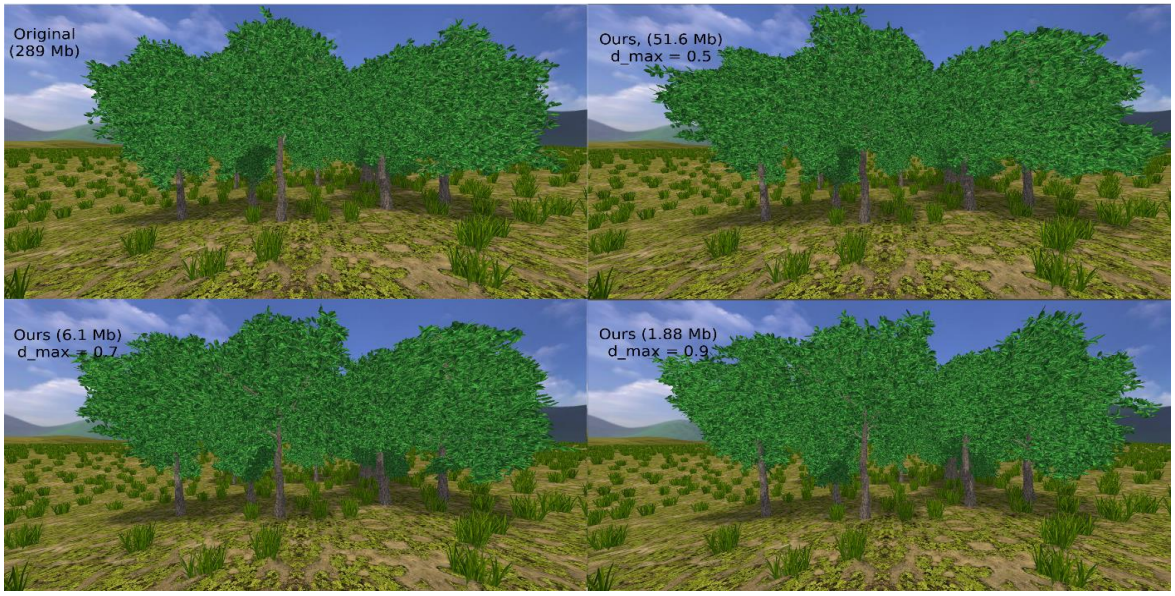


Figure 6: One group of trees without clustering and clustered with different d_max parameter and memory needed for each group. Due to leaves, the difference is incomprehensible. On fig. 7 it is better visible, the same scene is rendered.

Table 3
FLIP (and SSIM) mean error. For the case with leaves (fig.6).

Number of trees	$d\_max = 0.6$	$d\_max = 0.65$	$d\_max = 0.7$	$d\_max = 0.75$
10	0.027 0.052	0.027 0.053	0.029 0.055	0.029 0.055
25	0.0385 0.076	0.038 0.077	0.039 0.077	0.039 0.078
100	0.043 0.049	0.043 0.081	0.044 0.077	0.05 0.078

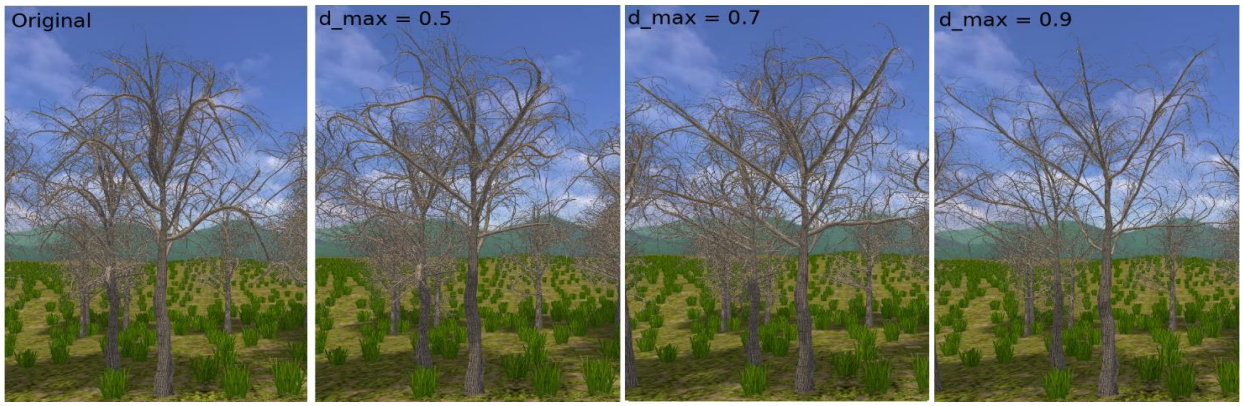


Figure 7: changes in tree structure after clustering with different d_max. Same scene as of fig. 6.

4. Synthetic trees generator

Plants compression algorithm described in the previous section is time consuming and moreover, many procedural generation algorithms can take up to several minutes for a single tree. It usually happens because they try to be biologically precise and simulate light, water and nutrients distribution. However, often there is no need to be so precise. This section will describe an algorithm to amplify the original tree set i.e. create new plants from basic branches taken from the clustering step. The general pipeline for it is shown on fig. 8.

Each plant or separate branch can be described with some set of parameters such as type, length, thickness, number of segments and child branches etc. These parameters can be treated as random variables and we assume that they are equally distributed for plants with the same type (plant type is set by generator and can denote plant species in a biological sense and some features). The probability distribution for all these random variables is estimated based on a given group of branches and random variable generators with which these distributions are created.

To make things easier we consider that the type of random variable is known in advance based on some empirical consideration, e.g., branch length is normally distributed and number of segments is a discrete random variable with values from 1 to 100.

The second important part of the preparation step is calculating 2D and 3D density fields for a whole scene with a group of branches. Then finally a set of new “synthetic” plants is created. Each plant is composed of basic branch instances. The type of a tree, its size, number of segments and parameters of child branches are all from random number generators made on the preparation step. After it a plant is “inspected” with some heuristic that can decide if something in the generation went wrong and plant should be recreated.

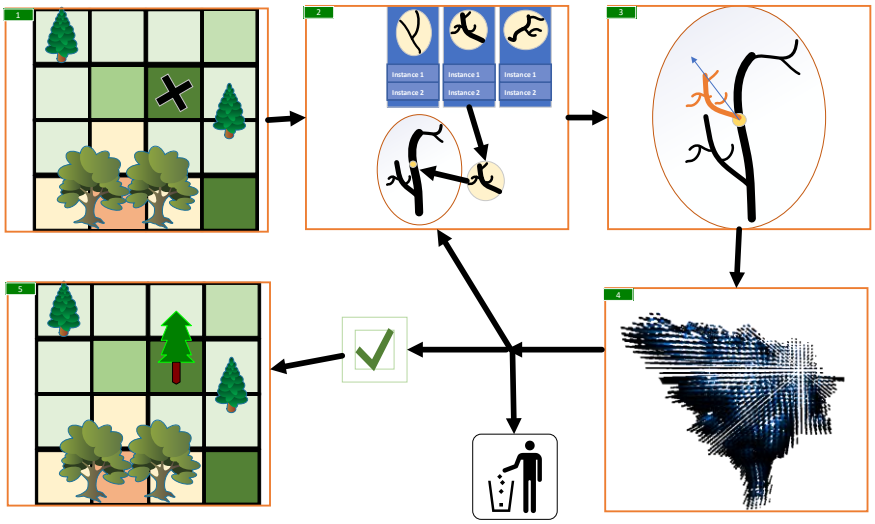


Figure 8: a synthetic tree generation pipeline. (1) An optimal place for tree is chosen, (2) Choose a place on a tree where to grove new branch and choose basic branch. (3) Set size and orientation of branch instance. (4) Recalculate density field and estimate its quality. (5) Decide whether we should grow another branch, plant the tree or refuse it and start again.

This generator works much faster than most procedural tree generators, it actually spends several seconds for preparation and then only a few milliseconds per constructed plant, while most procedural plants generators spend seconds or even minutes for a single tree. Table 4 shows generation time comparison for different procedural algorithms. The quality of the result is generally dependent on a number of given examples and the diversity of the original generator. The results of its work with different generators are presented on Figure 9 and Figure 10. Plants on the right on these scenes are created procedurally and then simplified by clustering, plants on the left are "synthetic". Synthetic plants generally look similar to procedural generated ones. So, the synthetic trees generator provides a universal and fast way to amplify the number of generated plants from existing parts.

Table 4
Generation time comparison.

Generator	Nodes count per tree	Generation time
Our "synthetic trees" generator	30000 - 50000	3-5 msec
Our procedural trees generator	30000 - 50000	2-4 sec
Self-organizing trees generator [20]	225000	21 sec
Plastic trees generator [21]	9000	64 sec
"Inverse" trees generator [22]	400-500	40-60 min
SnappyTree generator [18]	4000-5000	130-150 msec

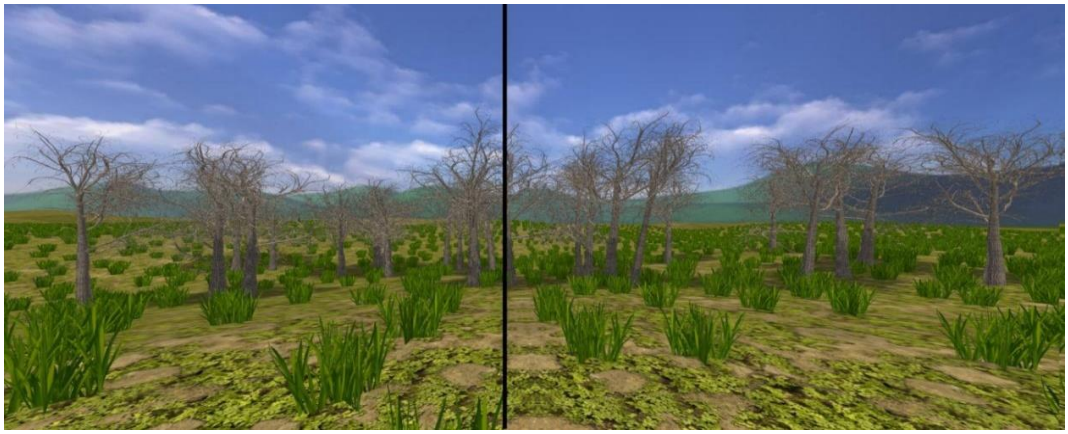


Figure 9: synthetic (left) and original (right). Synthetic mimic main features but can have less diversity and worse interaction with environment.

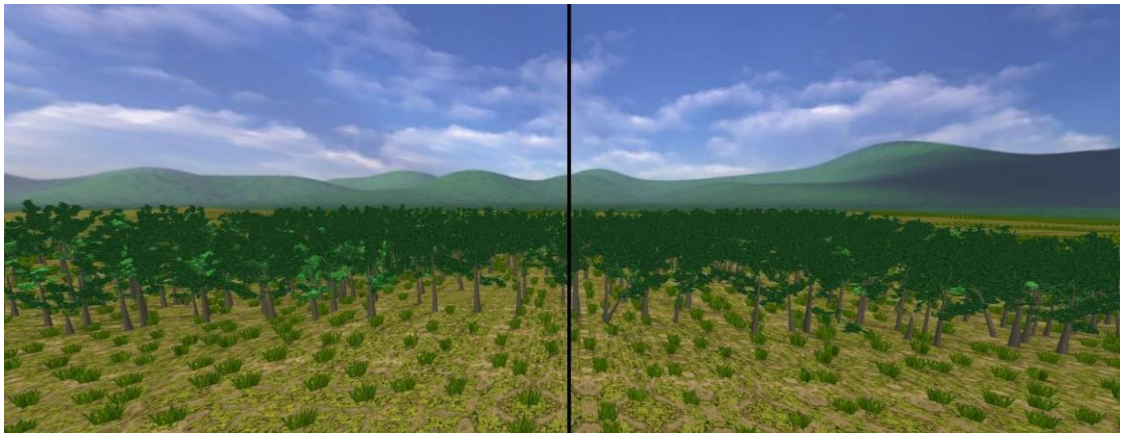


Figure 10: large scene with synthetic (left) and original (right) trees. On a large scale differences become less visible

5. Conclusion

A new method for reducing memory requirements for vegetation scenes is proposed. It is based on the approximate instancing concept and reduces the plants' geometry to the instances of a relatively small set of basic parts, which are chosen during the clustering process. Also, a method of constructing new plants, similar to procedurally generated ones, from basic parts, is proposed. Both methods could work with any procedural plant generator until it can provide plant structure data in a specific form. It makes it possible to use this method as a "black box" in different generators.

6. References

[1] SpeedTree IDV Inc, 2017. URL: http://www.speedtree.com/

[2] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, P. Prusinkiewicz, Realistic modeling and rendering of plant ecosystems, in: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, 1998, pp. 275-286.

[3] J. Kenwood, J. Gain, P. Marais, Efficient Procedural Generation of Forests, Journal of WSCG, 22(1) (2014) 31-38.

[4] V. Burkus, A. Kárpáti, Animated Trees with Interlocking Pieces, in: Proceedings of CESCG 2018: The 22-nd Central European Seminar on Computer Graphics, 2018.

[5] M. Makowski, T. Hädrich, J. Scheffczyk, D. L. Michels, S. Pirk, W. Pałubicki, Synthetic silviculture: multi-scale modeling of plant ecosystems, ACM Transactions on Graphics (TOG)

38(4) (2019) 1-14.
[6] J. C. Hart, The object instancing paradigm for linear fractal modeling, in: Proceedings of the conference on Graphics interface ’92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 224–231.
[7] J. C. Hart, Procedural synthesis of geometry, in Texturing & Modeling: A Procedural Approach, Third Edition, Morgan Kaufmann, 2003.
[8] C. S. Cordeiro, L. Chaimowicz, Predictive lazy amplification: synthesis and rendering of massive procedural scenes in real time, in: Proceedings of IEEE SIBGRAPI Conference on Graphics, Patterns and Images, 2010, pp. 263-270.
[9] G. Cordonnier, E. Galin, J. Gain, B. Benes, E. Guérin, A. Peytavie, M.-P. Cani, Authoring landscapes by combining ecosystem and terrain erosion simulation, ACM Transactions on Graphics (TOG) 36(4) (2017) 1-12.
[10] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, 2nd ed., Springer, New York, 2009, pp. 520–528.
[11] J. Freiknecht, Procedural content generation for games, Ph.D. Thesis, 2021. URL: https://madoc.bib.uni-mannheim.de/59000/1/Procedural%20Content%20Generation%20for%20Games.pdf
[12] J. Freiknecht, W. Effelsberg, A survey on the procedural generation of virtual worlds, Multimodal Technologies and Interaction 1(4) (2017) 27.
[13] J. Kim, Modeling and optimization of a tree based on virtual reality for immersive virtual landscape generation, Symmetry 8(9) (2016) 93.
[14] B. Carey, Procedural Forest Generation with L-System Instancing, Master’s thesis, 2019. URL: https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc19/02/MastersReport.pdf
[15] Greenworks Organic Software. Xfrog procedural organic 3D modeler, 2017. URL: http://xfrog.com
[16] SideFX Tree toolset, 2020. URL: https://www.sidefx.com/tutorials/tree-generator
[17] T. Niese, S. Pirk, M. Albrecht, B. Benes, O. Deussen, Procedural Urban Forestry, arXiv preprint (2020) arXiv:2008.05567.
[18] P. Brunt SnappyTree procedural trees generator, 2012. URL: http://www.snappytree.com/
[19] J. Komppa C++ port of SnappyTree generator, 2015.URL: https://github.com/jarikomppa/proctree
[20] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, P. Prusinkiewicz, Self-organizing tree models for image synthesis, ACM Trans. on Graphics 28(3) (2009) 1-10.
[21] S. Pirk, O. Stava, J. Kratt, M. A. Massih Said, B. Neubert, R. Měch, B. Benes, O. Deussen, Plastic trees: interactive self-adapting botanical tree models, ACM Transactions on Graphics 31(4) (2012) 1-10.
[22] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, B. Benes, Inverse procedural modelling of trees, Computer Graphics Forum 33(6) (2014) 118-131.
[23] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, M. D. Fairchild, FLIP: a difference evaluator for alternating images, Proceedings of the ACM on Computer Graphics and Interactive Techniques 3(2) (2020) 1-23.
[24] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE transactions on image processing 13(4) (2004) 600-612.