Animated Trees with Interlocking Pieces

Viktória Burkus*
Attila Kárpáti†
Supervised by: László Szécsi‡

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper presents a method to create plant models that can be displayed efficiently and directly on the graphics card, with real-time animation. Models are constructed from a finite set of pieces and a set of rules that define the matching possibilities of the pieces. We use skeletal animation to allow model instances of identical topology, created using the same pieces and rules, to assume unique poses. A level-of-detail scheme using hardware tessellation is also presented. We propose a tiling-based method for texturing the models, providing seamless, repetition-free patterning, approximating the great variety observed in nature, using only a limited set of example textures. Integration with foliage rendering using billboards is also discussed.

Keywords: Tree Rendering, Procedural Geometry, Skeletal Animation, Texture Generation

1 Introduction

In computer games and in all computer visualization applications where vegetation is involved, the challenge of producing and displaying extensive vegetation geometries arises. Authoring highly detailed individual models, especially if a large number of plants is needed, is rarely feasible. Therefore, procedural generation of geometries is required. In order to ensure efficient rendering, some level-of-detail scheme must be employed to reduce the geometrical complexity of plants at a large distance to the viewer.

GPUs are able to render instances of the same geometry extremely efficiently. This is particularly important in computer games that aim for less cost-intensive solutions. In this paper, our basic idea is to build tree models directly from such instanceable pieces. The pieces must fit together seamlessly to produce a solid model. Therefore, they must be interlocking when fitted together appropriately. When generating natural geometries, however, repeating identical motifs are conspicuous, and impair credibility. Therefore, even instances of the same base geometry must be processed in some way to introduce variability. In this paper, we propose composing trees from instances of a few model pieces, and describe both skeleton animation and tile-based procedural texturing to achieve variability. We explore the application of hardware tessellation to set level-of-detail adaptively. Foliage is added using a billboard-based method.

*burkus.viki@gmail.com
†karpati.attila.a@gmail.com
‡szecsi@iit.bme.hu

2 Previous work

2.1 Tree generation and modeling

Lindenmayer systems (L-systems) are the classic approach for plant simulation using formal rules [12]. The *L-PEACH* model [1] and space colonization algorithms [13] incorporate the competition for space and influencing environmental factors into the tree growth simulation. Plastic trees [10] allow interactive design by manipulating environmental factors. This is achieved by reverse engineering a skeleton for a tree model representing uninhibited growth, then pruning this model and distorting its structure using skeletal animation.

In this paper, we do not address the problem of tree growth simulation, but use a simple rule-based approach for generating trees. However, we instantly generate both the tree geometry and a skeleton that could be used for manual or simulated animation as described in the above literature.

SpeedTree [14] is an entire suite of modeling and rendering solutions for tree rendering. Trees are built by setting up *generators* the produce a graph of interconnected *nodes*. Our approach is similar in that trees are created by connecting pieces. However, these pieces directly correspond to geometric building blocks, so our output is not a triangle-mesh tree model, but a collection of instanceable pieces. Also, our solution is much more limited in scope, focusing only on most efficient rendering of relatively simple models.

2.2 Tree rendering

Instancing is a hardware feature that can be used to render many identical pieces of geometry with drastically reduced CPU-GPU communication overhead. It is not necessary to display each instance of the geometry with a separate *draw call*. Instead, the parameters that vary from instance to instance (e.g. transformation matrices), are uploaded in an *instance buffer*, and all objects are displayed with a single draw call, launching GPU pipeline operation. The advantage of instancing comes from rendering, when we avoid issuing individual draw call for all copies. Our approach is based heavily on this technique, as we build the entire vegetation using a few instanceable pieces, making them unique by means of blended transformations and texturing.

Instancing is also used in systems like SpeedTree [14]. However, SpeedTree only instances complete plants, and not their parts.

Approximate instancing [6] is a kind of instancing used for procedurally generated, highly varied and detailed models that do not contain precisely identical parts. When using the technique for rendering plants, entire specimen or their parts are replaced by similar geometries taken from a finite piece set. Thus, hardware instancing can be used to achieve efficient display and the difference may remain unnoticed. Our approach can be considered the reverse of approximate instancing because we build the objects from a finite set of pieces, and then make these objects unique, while they can still be rendered by instancing.

When rendering a large number of three-dimensional models (e.g. a forest), it is not feasible to display them at uniform level of detail. Geometries close to the camera appear large on-screen, requiring high level of detail, while the geometries away from the camera should be displayed at a low level of detail. This achieves better visual experience than using only low-detail models, with much smaller resource requirements than using high-detail models everywhere. Therefore, the process of model generation should make sure that models are available at different levels of detail. This is typically achieved by simplifying a complex model, [5]. Our approach is again the opposite: we build trees using low-polygon-count pieces, and add details using hardware tessellation and displacement mapping where necessary.

2.3 Animation

For animating plants, especially branches of trees, it is possible to use skeletal animation and skinning. The method we have devised and implemented is very similar to that of Pirk et al., called *plastic trees* [10], which also uses skeletal animation to animate tree branches and trunks. Contrary to them, our aim is not to simulate foliage formation influenced by biological effects. In our method, animation has two important goals. Firstly, the joints of the pieces can be set to obtain different poses, so models with the same piece configuration may have different appearance. Secondly, we can simulate the movements due to wind or other forces with further, generally finer changes to the pose.

2.4 Texture generation

In our approach, simple geometries are rendered, and we rely strongly on textures to provide individual details. We also generate these textures procedurally. Therefore, texture synthesis approaches are of interest. Extracting statistical properties from example images to synthetise similar textures is a basic approach [11]. Another possibility is finding pixel values based on similar neighborhoods [16, 8]. *Image quilting* [7] stitches together motifs from an example image to produce new textures. Tiling-based methods [3] create a set of tiles that fit seamlessly if placed adequately, and random tilings can be used to produce new textures. A tiling-based approach allows the non-periodic real-time texturing of indefinitely large surfaces based on very little tile placement data. Therefore, it is ideal for our problem of texturing tree pieces.

For generating the tiles themselves from non-tileable photo examples, we make use of the stitching algorithm from Image quilting [7], but in a slightly different way than Cohen et al. did [3].

2.5 Foliage rendering

While billboards are poor for displaying entire trees, the artifacts are diminished if billboards only represent branches or groups of leaves. Static billboard clouds are often used [4]. The approach of *2.5D impostors*[15] uses camera-facing billboards to render leaf clouds. The image on the billboard is not a static image, but it is dynamically generated depending on the view direction. The billboards are composited with each other and the solid geometry of the scene (e.g. branches) according to the correct depth information of the leaf clouds in billboards. With this method, the correct parallax and occlusion effects are observable when the camera moves. In this paper, we utilize this method for foliage rendering.

3 Solution overview

Figure 1: Generated trees.

We propose a method for creating trees (Figure 1) that can be efficiently rendered on a video card. These trees are composed using pieces taken from a finite set of possible shapes. The pieces interlock with other pieces with proper positioning, rotation, and scaling, creating a continuous surface. All pieces are rigged for skeletal animation, and bones of interlocking pieces are overlapping, producing a connected skeleton for the entire tree model. Thus, plants with the same topology, created by assembling the pieces in the same way, may also have individual pose, and the movement due to the wind can also be realized. For texturing the models, we suggest a tile-based procedural method, which allows each plant to be unique. The model can be efficiently displayed by the GPU, all instances of each piece type can be displayed with a single draw call and the automatic selection of the detail level is ensured by the tessellation unit. In section 4, we describe the assembly of the pieces, their animation, and the level-of-detail scheme. In section 5, we suggest a method for texturing. We describe the implementation of foliage rendering in section 6.

4 Tree model

4.1 Piece modeling

The first task is to create the polygon mesh models of the pieces of the tree and its branches. We demonstrate the operation of our method with just two models (see Figure 2): a slightly narrowing cylinder (the I piece) and a model of a symmetrical fork with co-dominant stems (the Y piece). These pieces are most suited for species like the almond tree. Asymmetric forks are possible with just appropriate scalings in bone transformations, discussed in section 4.3. Branch–stem attachments (a $\vdash$ piece), trifurcations (planar or 3D Ψ pieces) could be handled in a very similar manner, but those are not demonstrated in this paper.

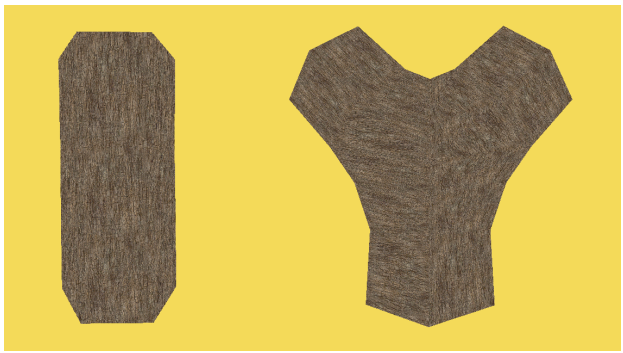

Figure 2: The I piece and the Y piece, forming the set of pieces we use to demonstrate our approach.

The pieces must be quad meshes to enable tessellation and tiled texturing. They must conform to several other restrictions to make them interlocking with themselves and one another. The constraints are explained in the following sections, where the reasons for them become apparent.

4.2 Piece assembly

The second step is to produce complete tree models using the pieces. This means the algorithmic production of the necessary transformations to position, rotate, and scale the pieces into interlocking poses. For the pieces to fit together, we formulate several requirements for the piece geometries. We define edge loops on the polygonal models where the models should fit together. We refer to these as *rims*. Each model must have a *root rim* and may have any number of *leaf rims*. The root rim connects the piece to the tree trunk, or a parent stem. Additional pieces can be added to a particular piece joining at a leaf rim. All rims must be homographic in the geometrical sense, i.e. there must exist, for any two rims (even if they are in different pieces), a projective transformation that maps one to the other. This makes it possible to transform a piece in such a way that its root rim interlocks with the leaf rim of a parent piece, producing a continuous surface.

We have chosen a regular hexagon as the universal rim edge geometry, and limit the transformations to similarities. This means that there is not only one, but six possible transformations fitting a root rim to any leaf rim. This increases the variability of possible models, while smaller twists are easily handled with skeletal animation later. This choice also allows low-poly piece geometries, while more details, and smoother branches, can be implemented in the tessellation and texturing phases.

The root rim of all models has the same orientation and size in their respective model spaces. Identical root rims guarantee that we can fit any other root of another piece to the leaf rim with the same transformations. These transformations are stored as metadata for the piece models. When modeling the pieces, the transformations also have to be produced. We refer to them as the *building transformations* of the piece model. Figure 3 shows a two-dimensional drawing of the Y model. The building transformation for the right leaf rim is the transformation that transforms the lower red coordinate system into the upper red coordinate system.

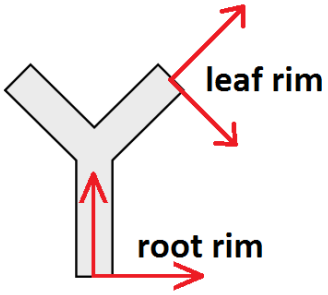

Figure 3: A building transformation maps the root rim of any piece to a leaf rim.

To determine the poses of the piece instances, we first compute the *topological transformations* (i.e. the piece transformations for the non-animated tree). The first step is choosing a root piece. All other pieces are matched in the root piece's modeling coordinate system. In other words, a transformation must be calculated that transforms the piece from its own modeling coordinate system into the root piece's modeling coordinate system. These can be obtained recursively. The topological transformations of the pieces that interlock with the root piece are the same as the building transformation defined for the root piece's leaf rims. The topological transformation of any piece interlocking with any parent piece consists of two parts. The first transformation is the topological transformation of the parent, which transforms the new piece to the parent's pose, root rims coinciding. The second transformation is a building transformation. Applying its topological transformation on each piece leads to the pieces aligned to form a coherent tree. Then the whole tree can be transformed into the world's coordinate system from the root piece's modeling coordinate system. For this, the modeling transformation of the tree should be applied to all the pieces after the topological transformation. Figure 4 shows the tree constructed in such a manner.

Figure 4: A tree constructed of Y pieces only, using the topological transformations.

4.3 Skeletal animation

To animate tree pieces, we use linear blend skinning animation. We define a skeleton for morphing the polygon mesh model of a piece. The root joint is always at a predetermined position and orientation in the center of the root rim. The vertices of any rim can be linked to exactly one joint, with unit weight. Thus, in a tree model composed of pieces, the skeletons can be connected by simply connecting the root joint of every piece's skeleton to the terminal joint that belongs to the parent piece's matching leaf rim.

Figure 5 shows the skeleton hierarchy of the Y model. When animating the tree, the root joint is connected to the parent's terminal joint rigidly, as rotating the root joint would cause the rims not to fit.

The bone transformations for the tree are computed similarly to the topological transformations discussed earlier. The difference is that building transformations are now replaced with joint transformations of the animated skeleton. The root bone transformation of each piece instance is computed by taking the terminal bone transformation of the parent, and appending the terminal-joint-to-leaf-rim transformation.

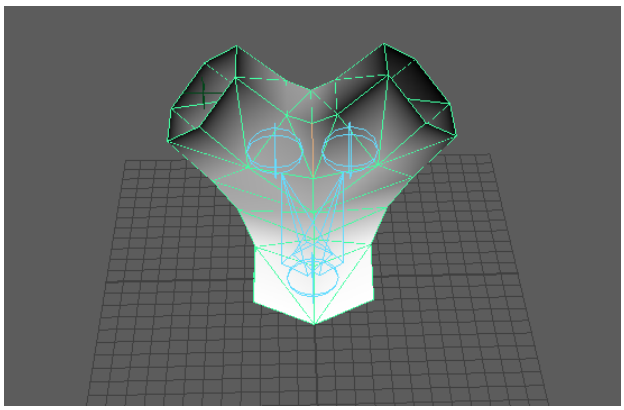

Figure 5: Skeleton hierarchy of the Y model. The coloring of the model illustrates the vertex blending weights that belong to the root joint. The vertices of the root rim are completely white, so they are entirely dependent on the root joint. The vertices of leaf rims are completely black, so they are not bound to the root joint.

In typical implementations, the maximum number of blending weights per vertex is four. The *blend indices* vertex attribute identifies the bones that have non-zero weights. This allows any geometry rigged for skeletal animation (including our pieces) to have a high number of bones. We place the transformations of all bones of an animated tree in a single buffer. For every piece instance we record the offset where its bone transformations are located in the buffer. This offset is added to the blend indices in the skinning vertex shader. Thus, there is no limit imposed on the number of bones or pieces.

Figure 6 shows an animated tree in an altered pose.

Figure 6: A skeletally animated tree with randomized joint transformations.

4.4 Level of detail

In order to produce pieces in different levels of detail we use hardware tessellation. Depending on the distance from the camera, we suggest an exponentially decreasing resolution level with a suitably selected maximum for the running environment. Since our model is a quad mesh model and normal vectors are known, the task is a *PN patch tessellation*. We considered two algorithms that solve this problem: *Phong tessellation* [2] and *local, polynomial G1 PN quads* [9]. We found Phong tessellation more appropriate to tessellate multiple models at runtime, as it has lower computational cost.

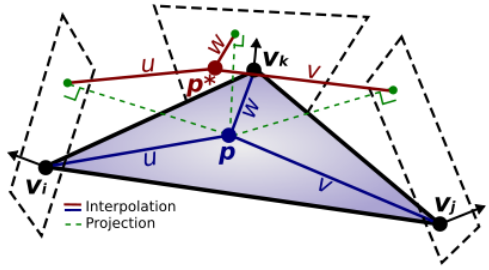

Figure 7: Phong tessellation [from Boubekeur et al.[2]].

Figure 7 illustrates the tessellation of a triangle as described by Boubekeur et al.[2]. However, we work with quads instead of triangles. Consequently, the algorithm had to be adapted to tessellating quads. The difference is that when calculating interpolation weights, we use bilinear interpolation over four points. This can be implemented in a domain shader when using hardware tessellation.

We guaranteed that rim vertices of interlocking pieces match without tessellation. When tessellation is used, the matching of newly created vertices must also be guaranteed. Firstly, it is necessary to guarantee that the number of new rim vertices on both pieces are equal. Secondly, they must also have matching positions. To guarantee that the tessellation levels are equal, it is enough to guarantee identical on-edge tessellation levels, because the models fit at the edges. This can be achieved if the tessellation levels of edges are only dependent on the properties of the two vertices. It is also important that both vertices are symmetrically included in the calculation, so the tessellation level does not depend on the order of the vertices. For example, the average of two positions is symmetrical, so it does not depend on the order of the vertices. As a result, the tessellation level of the edges are determined based on the average of the two vertex positions and its distance from the camera position. The tessellation level of the edges and inner points should be similar to get a good tessellation. For this reason, we determine the tessellation level of the inner points according to the distance of the camera from the average position of all four vertices of the quad.

The position of the new vertices depends on the position of the vertices and the normal vector. Accordingly, when joining the vertices of the rim, not only the positions must be the same, but also the normal vectors. This is a property of piece model geometry that we would want not only for tessellation, but also for continuous shading. Therefore, not only the positions, but the normals are also pre-determined for all rim vertices. For the normals to be consistent with the surface orientation, the quad loops at the rims always form a (tessellated) cone surface with a universally fixed aperture angle. Note that this does not impose a fixed-aperture cone shape on tree branches, as joint transformations may include longitudinal scalings to achieve any aperture. Figure 8 shows the original model and a tessellated version.

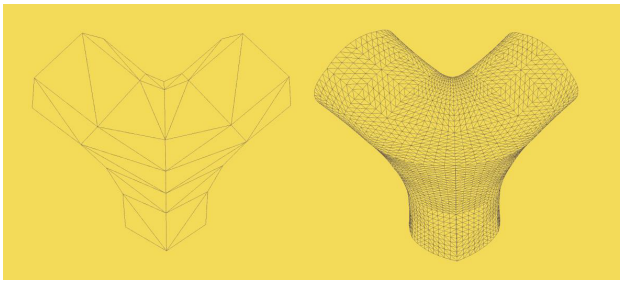

Figure 8: Different levels of detail achieved using GPU tessellation.

5 Texture generation

5.1 Tiling for piece geometries

In order to enhance the detail and plausibility of the tree, bark or other surface details can be realized by texturing. The first possible solution is to apply the same texture image on the surface of every piece instance, but this should be avoided because all pieces would look the same. Instead, we propose *texture tiling*, placing a different image on every quad's surface. Our goal is to select an image semi-randomly from a predetermined set of textures, so that the tile configuration on each model instance will be different, which results in pieces with unique appearance. We need to observe alignment rules to match the tiles seamlessly. These are usually expressed in terms of tile edge colors that must be identical for two tiles to match. Our current implementation assumes a special limited case, where all model edges have a fixed color (see Figure 9 for a color coding of edges). This means that the frame of the texture tiles fitting on a quad is fixed, and only the insides of the tiles may change. We propose a method to generate such sets of tiles in section 5.2. Please note that our method can easily be generalized to any sets of tiles with any edge colorings, e.g. using the random tile placement described by Cohen et al. [3].

We create a texture array that we upload the tile texture images to. With our simple assumption of a static patterns on all edges, the quads of the model can be classified ac-

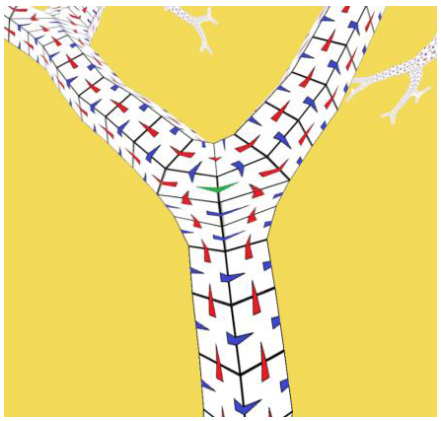

Figure 9: An example of a edge colors for tile matching. Black edges are quad borders. Color triangles on the polygon border indicate edge color.

cording to the rules that the tiles they accept must obey. It is possible to tile the I model with a single such class of tiles, and the Y model only needs a second class of tile near the fork, as indicated by the numbers shown in Figure 10.

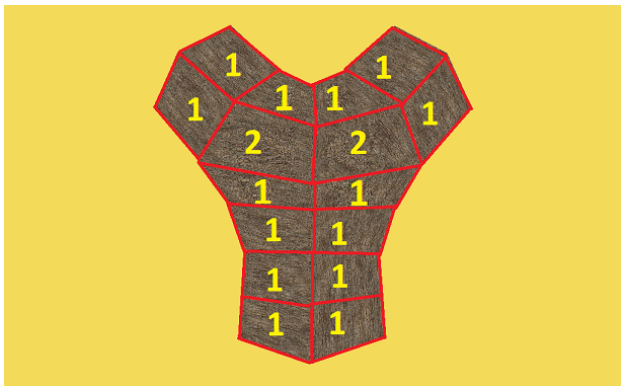

Figure 10: Types of polygons. These numbers determine what kind of images the quads accept from the texture array.

Figure 11 represents the texture array. Multiple images can be added to the type 1 and 2 polygons:

Any quad may be assigned one of the matching tiles at random. This assignment should be different for every piece instance, and part of the instance data, along with the bone index offset for the animation. In order for the pixel shader to decide which slice of the texture array should be used for the currently rendered polygon, this data about the tile arrangement must be passed. The information on the processor side can be saved in a single array that contains the indices of the tiles, i.e. the texture array slices. The array size is the quad count of the model, and it is indexed with the primitive ID available in the GPU pipeline. The number read from the array is used to address the texture array.

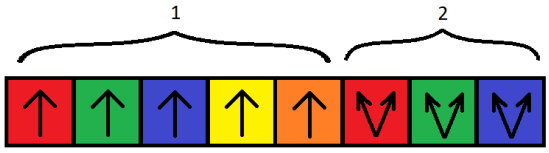

Figure 11: Illustration of the texture array. Different colors represent different patterns, similar arrows represent images with identical matching options.

5.2 Tile set generation

To create texture tiles with edges matching given patterns, we use a variant of the stitching process described for image quilting [7]. The frame of the texture is given as four images aligned that must appear along the edges, but the inside is freely selectable. This means we are able to generate a large number of textures even with identical edge colors.

In order to generate such textures from photo images, we select a few (appropriately generic) photos to serve as edge patterns, one for every edge color. Then, any image we would like to show up in the tile is stitched together with all four edge images using the image quilting approach. This means that a minimum error cut separating the overlapped images is found using dynamic programming.

Figure 12 shows the tile set we use.

Figure 12: Texture set. The picture on the right shows an example usage of the tile set which is shown on the left side.

6 Foliage

To render foliage, we chose 2.5 dimensional impostors [15], because they offer correctly depth-tested results with minimal artistic input. The positions of the billboards are determined by the pieces of the tree, which can be assigned foliage in the tree generation phase. All billboards

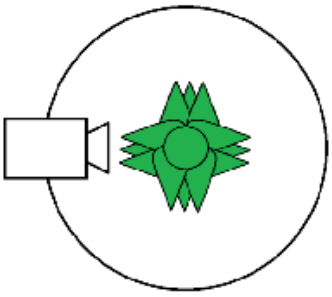

Figure 13: First render stage.

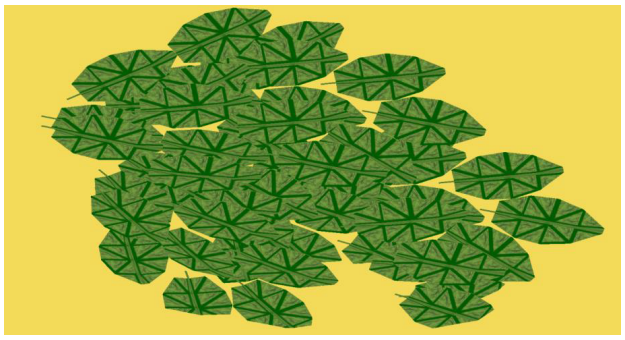

Figure 14: Result of the first render pass producing the 2.5D impostor texture.

display the same leaf cloud. In case of 2.5 dimensional impostors, the billboards are supplemented with depth information . Billboard fragments can be depth-tested against one another, and against the solid tree geometry. As the view direction changes, the geometry on the billboard is also rotated, so the impostors are always re-rendered according to the view direction of the camera.

Using a modeling program, we created a leaf cloud model. Rendering is executed in two passes. In the first pass, the model is rendered into texture. (Figure 14). The direction of the camera does not change, but its position is aligned with the surface of the sphere around the cloud, with the camera facing the leaf cloud at the center of the sphere. Figure 13 shows the first pass. When rotating the camera, the leaf cloud is rendered at a different angle. The texture of the billboards must always be rendered according to the view direction of the camera. As a result, the two-dimensional image of the texture appears three-dimensional in motion. In the alpha channel of the resulting texture, we save the distance information, that is, the distance between the position and the billboard, which helps to determine which leaf is closer to the camera when multiple images close to each other are drawn. In the second pass, the rendered image is used as a billboard positioned near the appropriate branches. The alpha is used to determine fragment depth.

Figure 15 shows the result.

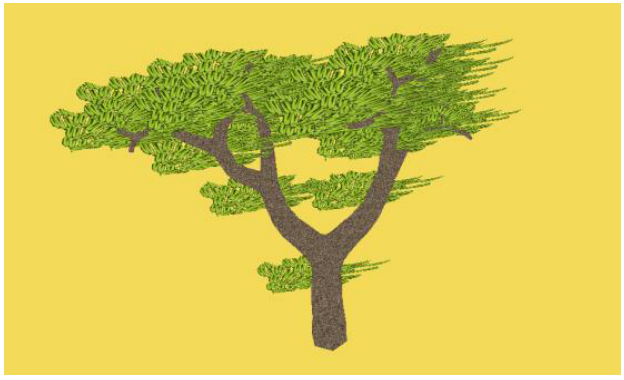

Figure 15: Foliage rendered with the 2.5D impostor technique.

7 Implementation

We implemented our method using *DirectX11* and *C++*. *HLSL* was used to program shaders. We used the *Effect Framework* for pass scripting, shader constant management, and texture resource binding. The development environment was *Visual C ++ 2010 Express*. For modeling, we used *AutoDesk Maya* modeling software, from which models were exported to *DAE* format. Importing these files into the project was done using *Assimp*. Assimp is an open source library primarily for C and C++, which made it easy to download multiple 3D file formats into our project. In addition, we used *Boost*, which also provides libraries for C++, so we have extended our implementation with modern C++ language elements. To create 2D texture arrays, we used the *texassemble* program, which delivers the result in dds format.

8 Results and Conclusions

The completed implementation was run on two different computers. We generated trees during the execution and recorded the performance without animation (Figure 16). To measure the performance impact of the tessellation we measured the frame time with various tessellation levels. At the highest level (10) it was 2.8-3.0ms, and at the lowest level it was around 2.0 ms (measured on an NVidia GTX 950M).

Our solution fulfills our initial requirements. The trees build up algorithmically from finite set of pieces. The trees are bearing a certain degree of variation. The trees' textures are randomly chosen from the predefined texture set, while the foliage is aligned to the branches.

We implemented two algorithms for building trees. One of the algorithms alternates between the two kinds of pieces. The second chooses randomly from the set of elements using equal probability. Thus, we did not yet explore how more complex rules or grammars could be used to replicate real-world tree topologies.

In the future we would like to further demonstrate

Fák száma	GPU: GTX 980 Ti CPU: i7 4790K	GPU: GTX 950M CPU: i5 6200U
1	0.5319ms	3.1250ms
10	0.9523ms	4.3478ms
50	3.5714ms	11.1111ms
100	6.2500ms	25ms
250	17.5439ms	38.4615ms
500	28.5714ms	83.3333ms
1000	58.8235ms	166.6667ms

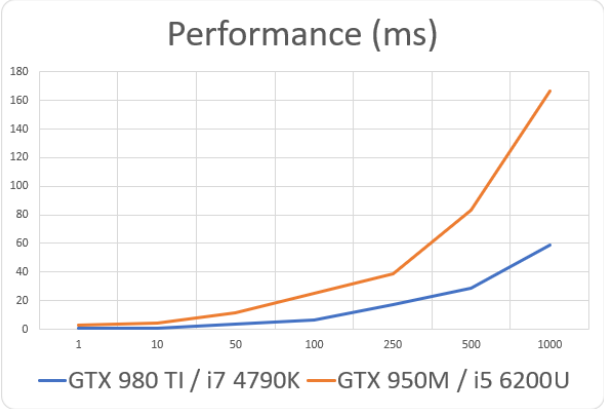

Figure 16: Measurement results.

the capabilities of our solution reconstructing actual tree species and their bark patterns. We also plan to simulate motion due to wind attaching springs to the skeleton joints in a physics engine.

9 Acknowledgements

This work has been supported by OTKA K-124124.

References

[1] MT Allen, Przemyslaw Prusinkiewicz, and TM DeJong. Using l-systems for modeling source–sink interactions, architecture and physiology of growing trees: The l-peach model. *New phytologist*, 166(3):869–880, 2005.

[2] Tamy Boubekeur and Marc Alexa. Phong tessellation. In *ACM Transactions on Graphics (TOG)*, volume 27, page 141. ACM, 2008.

[3] Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. *Wang tiles for image and texture generation*, volume 22. ACM, 2003.

[4] Xavier Décoret, Frédo Durand, François X Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 689–696. ACM, 2003.

[5] Qingqiong Deng, Xiaopeng Zhang, Gang Yang, and Marc Jaeger. Multiresolution foliage for forest rendering. *Computer Animation and Virtual Worlds*, 21(1):1–23, 2010.

[6] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM, 1998.

[7] Alexei A Efros and William T Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346. ACM, 2001.

[8] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. In *ACM Transactions on Graphics (ToG)*, volume 24, pages 777–786. ACM, 2005.

[9] Chavdar Papazov. Local, polynomial g 1 pn quads. In *International Symposium on Visual Computing*, pages 63–74. Springer, 2014.

[10] Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Randomir Mech, Bedrich Benes, and Oliver Deussen. Plastic trees: interactive self-adapting botanical tree models. *ACM Transactions on Graphics*, 31(4):1–10, 2012.

[11] Javier Portilla and Eero P Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International journal of computer vision*, 40(1):49–70, 2000.

[12] Przemyslaw Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253, 1986.

[13] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. *NPH*, 7:63–70, 2007.

[14] SpeedTree. SpeedTree. `http://www.speedtree.com/`, 2018. [Online; accessed 18-February-2018].

[15] Gabor Szijarto and Jozsef Koloszar. Real-time hardware accelerated rendering of forests at human scale. *Journal of WSCG*, 12(1–3), 2004.

[16] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.