SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 3009

GENERIRANJE MODELA BILJAKA I NJIHOV PRIKAZ U STVARNOM VREMENU

REAL TIME RENDERING OF PLANTS

Ana Stepić

Zagreb, Svibanj 2013

Real time rendering of plants

Assignment:

Research models of plants. Review methods used to produce such models. Review L-systems, usage of different methods of rendering textures in plant modeling, and procedural techniques. Develop algorithms for plant generation. Discuss Levels of Detail (LOD) usage in real time applications.

Build a software implementation using Unity 3D engine, evaluate and compare mentioned methods. Present results on various examples. Append used algorithms, source code and results with due explanations and documentation. Cite used literature and received help.

Student:
Ana Stepić

Mentor:
Željka Mihajlović

ACKNOWLEDGMENTS

Thanks to my husband, Andrija Stepić, for all his love, help and support, and pushing me to achieve my goals.

To my mentor, prof. Željka Mihajlović, for her understanding and patience.

And last but not least, thanks to every developer responsible for making Unity all it is today, giving us all an indispensable tool like no other for visualization of our dreams and ambitions. Working with Unity has been an enjoyable experience and I intend to work with it in the future as well.

Thank you all, for without you this would have never been possible.

Table of Contents

Introduction

Since the dawn of humanity, plants had played a major role in our existence – from building shelters out of wood to eating their fruits and seeds. Even in this age and time, plants are still a major part of our lifestyle.

As the age of computer graphics began, men have tried to virtualize everything about our world, both on microscopic to macroscopic level. It is only natural that we try to represent things important to our everyday life such as plants. Rendering of plants is used in numerous areas for countless purposes. Plants are rendered to enhance the experience of games, architectural models and virtual reality. They are used in virtual encyclopedias and other educational software. Growth and behavior of plants is also researched using computer and mathematical models.

The need of rendering and modeling of plants has forced development of many techniques and methods. This article will review the most important real time rendering and modeling techniques.

This thesis covers billboard related plant rendering techniques, procedural methods of plant creation and modeling, their impact on performance and usage of different levels of detail (LOD) for better real time performance. The implementation and testing of the methods was done in 3D engine Unity using C# as the programming language.

1. Geometric modeling of plants

1.1. Billboarding and Billboards

Billboarding is a technique that adjusts the object's orientation so it would face another object, usually the camera. However a ***Billboard***, is not always an object that has been *billboarded*, the word is also often used in computer graphics to refer to a flat textured plane (in the future, term billboard will denounce both types).

Billboards are popular in real time applications that require a large number of polygons. No matter how much graphics hardware and software keeps improving, it is never enough. Developers keep taking advantage of hardware improvements and pushing it to its boundaries, always struggling with the amount of polygons that can be displayed with a decent frame rate.

Billboards are used to cut back on the number of polygons required to model a scene by replacing geometry with a texture. Billboards are majorly used for foliage, especially distant plants. When we look at a simple tree and think about its structure and what would be required to model it (leaves, branches, trunk, roots, flowers, fruit and seeds are some of the things we would need to model), we can easily come to the conclusion that for a decent representation we would need a large amount of polygons. Just one tree could take up all our polygon allowance, and what if we wanted to model a forest? The number of polygons increases drastically.

Instead of modeling such complex objects with polygons, they are often replaced by a textured quad. Billboarding is used to guarantee that the quad is always ***facing*** the camera (or the view plane, depending on the type of billboarding applied), therefore the viewer never realizes that the billboard is actually a flat surface. ***Facing*** meaning that the normal of the polygon is pointing in the direction of the viewer as can be seen later in Figure 1.3.

This method is performance efficient – it requires only minimal amount of polygons. However it also has some downsides. The shading of the billboard will not be

correct, especially if the same texture is used for all orientations. It would appear as if the light source was moving along with the viewer.

There are four types of billboarding often used – viewpoint oriented, view plane oriented, spherical and cylindrical. [1] [2]

In a flat plane, direction of the plane would be determined by the direction of the normal of the plane's surface. The normal points in the direction of the target the plane is facing.

If the target was a center of a sphere, and the plane was orbiting around the sphere in all directions – that plane would be a spherical billboard. In spherical billboarding there is no restriction to the orientation of the object.

Cylindrical approach restricts the rotation of the plane to a vector (usually one of the axis). Which would mean that the plane is moving perpendicular to the surface of a cylinder. For foliage, and most other objects cylindrical approach is used to restrict rotation of the object to the Y axis. All the billboarding methods covered in this paper are cylindrical.

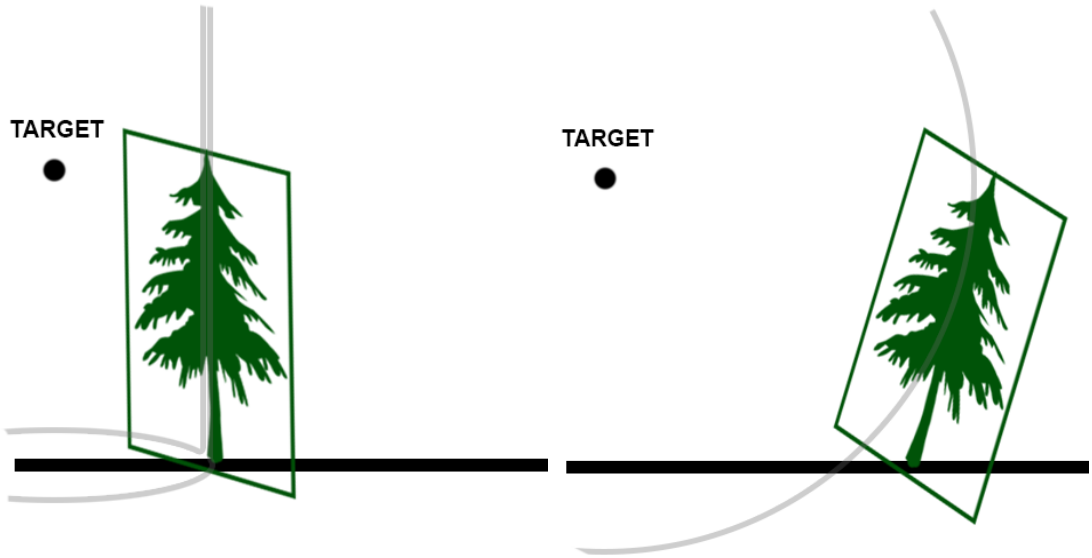

Figure 1.1: Left - Cylindrical Billboarding Right - Spherical Billboarding

The most often used billboards and the simplest of them are geometrically made of a single flat quad polygon or two triangles forming a quad (as can be seen in Figure 1.2).

Figure 1.2: Left – a simple billboard consisting of two triangles, Right – same billboard but with a transparent texture applied

1.1.1. Viewpoint vs. View plane oriented Billboards

Each time the camera moves, all billboards need to adjust their direction (if it is incorrect) so they would always be facing their target.

“The target” in viewpoint oriented billboards is the viewpoint, the camera. Viewpoint billboards always turn towards the camera. If the billboard is close enough to the viewer and he is moving, it would appear that the “tree” is rotating towards the viewer. Trees don’t rotate in real life, so experiencing large alterations of angle would look bizarre. Which is why viewpoint oriented billboarding is best used method for distant objects, particles or directional billboards (more on them in the next chapter).

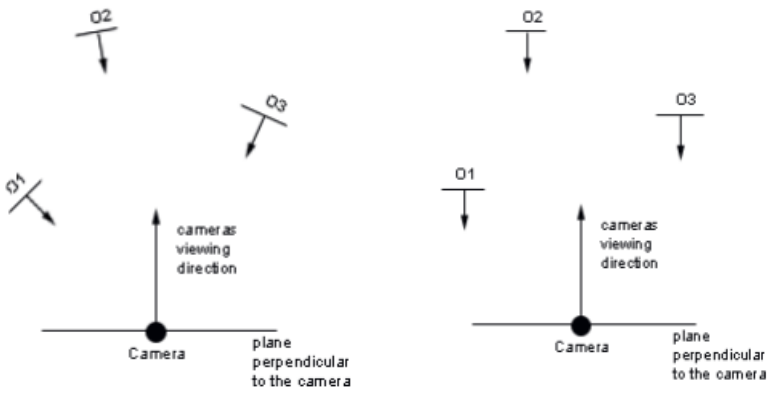

Figure 1.3: Objects in the scene O1...O3 are billboarded planes, arrows represent their normals Left – Viewpoint oriented billboards, Right – View plane oriented billboards

"The target" in view plane oriented billboards is the view plane – a plane perpendicular to the camera. Billboards turn so their normal would be perpendicular to the surface of the view plane. This way, polygon of the billboard would be parallel to the view plane. This method is often used in 2D applications and is sometimes used in 3D applications as well.

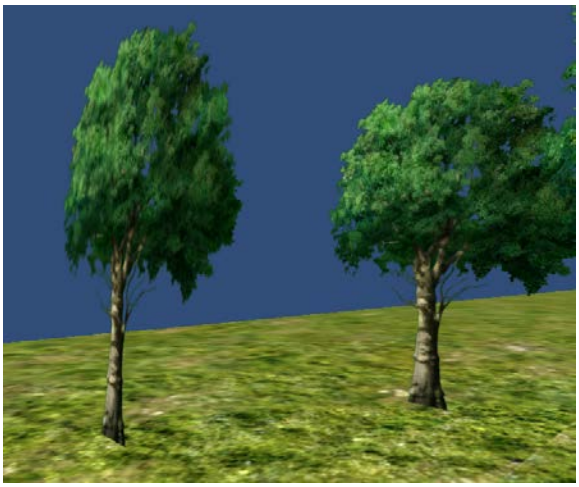

Figure 1.4: Regular textured quads – their two dimensional nature becomes obvious when looked from different angles

Figure 1.5: Left – viewpoint oriented billboards, Right – view plane oriented billboards

Table 1: Advantages and disadvantages of Billboarding

Advantages	Disadvantages
✚ Fast and Cheap (Low polygon count) ✚ Looks beautiful from afar ✚ Good for small foliage (such as grass) even up close ✚ Works well for 2D applications	▬ Unlit – does not respond to dynamic lighting. Shading in used textures is not accurate. ▬ Textures used usually need transparency (they require an extra channel – alpha channel) ▬ Doesn't work if the object is close enough to be seen from above or below ▬ If close to a moving camera, objects are noticeably turning towards the viewer

Implementation in Unity

Unity supports most image formats – relevant ones for billboards being those which support transparency. Formats that have an alpha channel such as .psd (Phostoshop file), .tiff, .png, .bmp and .tga are recommended for use with billboards. However formats that don't support partial transparency such as GIF and BMP are also supported.

Billboard implementation in Unity is straight-forward and simple. Unity simplifies changing directions of objects by using the function LookAt(worldPositionToLookAt, WorldUpVector). Since the billboards should always rotate around Y axis – the Up vector is equal to the Y axis which is the default value for the World Up vector. The second parameter is therefore redundant for it defaults to world up vector which is exactly needed in this case.

In case of viewpoint oriented billboards, as the first parameter of the function the active camera's position is sent. In case of view plane oriented billboards, it looks to the point in space that is in the direction of the view plane. The direction of the view plane is determined easily – it is equal to the back vector of the camera (which is negative of the front vector of the camera).

```
public class CameraFacingBillboard : MonoBehaviour
{
       // If false billboard is view plane oriented, if true it is viewpoint oriented
    public bool ViewpointOriented = false;

    void Update()
    {
       Vector3 lookAtTarget;

       if(ViewpointOriented)
            lookAtTarget =Camera.main.transform.position;
       else
              lookAtTarget =transform.position - Camera.main.transform.forward;
        // makes sure that we are going to rotate only around the Y axis
       lookAtTarget.y = transform.position.y;
       transform.LookAt(lookAtTarget);
    }
}
```

Figure 1.6: Script that attached to an object makes it behave like a viewpoint or view plane oriented billboard.

However, implementation of billboards is not something usually done on the CPU. To improve performance it is done either with shaders (GPU is able to billboard objects much faster than CPU), or objects are billboarded only at the start of the application (this would require limiting camera movement so it wouldn't see billboards from awkward angles). Unity supports both GLSL and HLSL shaders with

(usually) minimal modifications, so finding a billboard shader or writing one is relatively easy.

Unity itself uses billboards with its terrain engine and particle engine, so finding a billboard shader is a simple as downloading Unity built-in shader source codes.

1.1.2. Directional Billboards

In regular billboards, the idea is to apply one texture to a plane. In directional billboards, the idea is to use the same simple geometry, but switch between one of several possible texture maps depending on the direction of view. For example, 4 textures could be supplied to represent a tree – a picture of the tree from the front, sides and back. In the application, if the viewer is in front of the tree he would be seeing the front picture. When they move to the side, the picture would change into the side picture. And so forth.

This method makes it possible to see “the tree” from more perspectives, as many as there are textures. One of the problems with this is the visible “pop” when the switching between the textures occurs. To minimize the effect a large amount of pictures should be supplied. If the images are photographic or pre-rendered they would need to be supplied and would increase the size of the application dramatically along with its memory usage. It is impractical, expensive and the results are not satisfactory.

However, this method is still widely used in real time applications, why? Because of the ability to dynamically create textures while application is running.

Most engines today have a method to cope with high polygon counts required to display certain scenes, such as forests. If we are using a polygonal model of a tree to render the forest, that would require the engine to render a large amount of polygons. When a tree is a set distance away from the viewer the engine decides to use a stand in for the real tree. It replaces a real tree with a directional billboard (which can cause “pop” effect if done before reaching adequate distance).

The engine uses the polygonal model of the tree to render textures used to make the directional billboard. Because they are generated on the fly (or on application install) they do not require a large amount of memory or effort. The engine does it subtly and renders them in required directions on the go so the viewer never really notices he is looking at the flat billboards. Some engines (such as UDK which uses Speed Tree for rendering trees), try to minimize the pop effect by blending between real trees and billboards and then slowly fading one out (depending on whether we are moving towards or away from the tree).

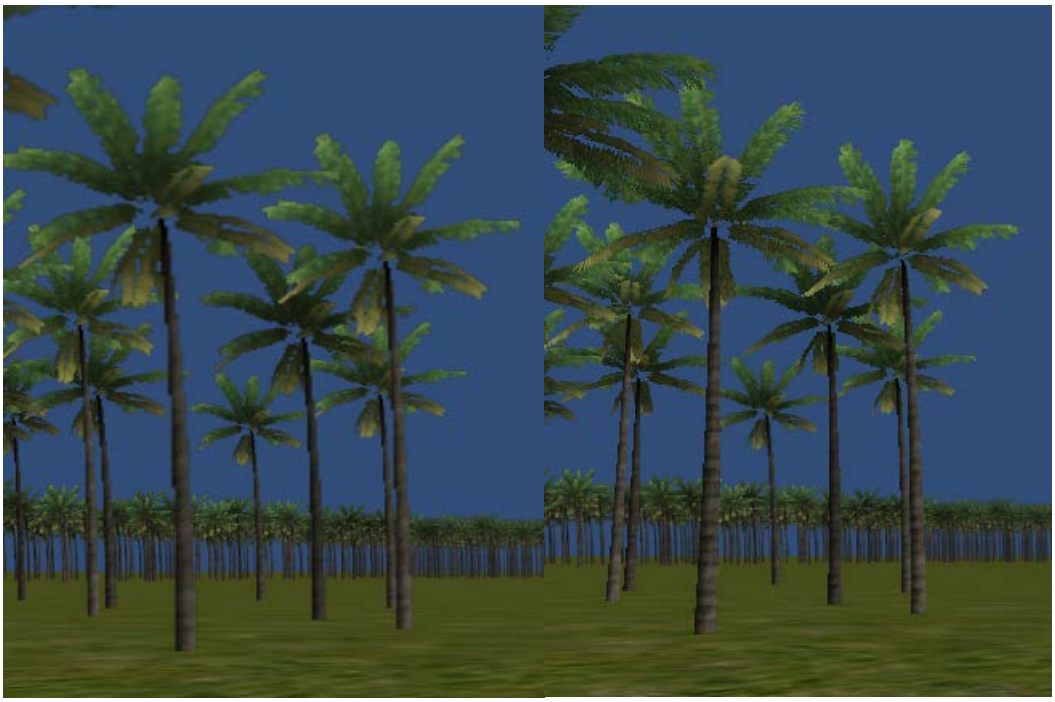

Figure 1.7: Left - Unity Billboarded trees, Right - polygonal models of trees

Table 2: Advantages and disadvantages of directional billboards

Advantages	Disadvantages
+ Low polygon count + Looks beautiful from afar + Makes objects appear more 3D + More variety – same objects can look different depending	- Unlit – does not respond to dynamic lighting. - Textures used usually need transparency (they require an extra channel – alpha channel) - Doesn't work if the object is close

on their rotation	enough to be seen from above or bellow - Visible “pop” effect when changing between textures - Needs a relatively large amount of textures

Implementation in Unity

Directional billboard is a billboard nonetheless, which means it needs to use a billboarding script written in the previous chapter. The only difference between a regular billboard and a directional billboard would be in the script which would change the texture used in the directional billboard according to the angle the viewer is seeing the tree from.

```
public class AngleBasedTextureSwitcher : MonoBehaviour
{
        public Texture2D[] Textures;
        private int currentTextureIndex = 0;
        private Vector3 frontVector;

        void Awake ()
        {
                // we remember the direction the tree is initialy facing as a reference
                vector
                frontVector = transform.forward;
        }
                void Update ()
        {
                float viewAngle = Vector3.Dot(frontVector,
                (Camera.main.transform.position-transform.position).normalized);
                viewAngle = Mathf.Rad2Deg*Mathf.Acos(viewAngle);

                float increment = 360.0f/Textures.Length;
                int interval = (int)(viewAngle/increment);

                if(currentTextureIndex!=interval)
                {
            // change the texture
                        renderer.material.SetTexture("_MainTex", Textures[interval]);
                        currentTextureIndex = interval;
                }

        }
}
```

Figure 1.8: Code that switches textures based on the view angle to the reference point

1.1.3. Slices

The problem with both ordinary and directional billboards is that they are not really the objects they are trying to imitate – they are not 3D. They are just two-dimensional imitations and closer you get to them more obvious it becomes.

That is the problem this method tries to correct.

From moderate distances, our mind is not able to discern the full complexity of the geometric details in the tree crown. The clues to whether we are looking at a 3D object or a 2D picture are parallax – the changes we see in the object as our perspective of it changes. This method tries to achieve that a parallax effect by representing the object with a set of textured parallel layers – slices. The textures used are generated from a detailed polygonal model of the tree. One slice is a textured quad.

One set of slices would be an equivalent to one directional billboard (albeit more 3D). However, like directional billboards, more directions equals more textures and more required memory. And switching between textures would result in the same “pop” effect. This technique tries to prevent the abrupt flipping between different sets of slices and reduce the number of required textures, by using blending between the textures. Tree crown is actually rendered from 2 perspectives (2 different textures) at once and transitions between them are smoothed out.

In the Figure 1.7 tree is rendered using this technique with two pairs of slices, whose textures have been made from different angles. The transition between the planes has been smoothed out and the nature of the trick used is not visible to the viewer. Such tricks are not applicable for the solid elements of the tree (branches and the trunk) so they are rendered as a polygonal mesh. [3]

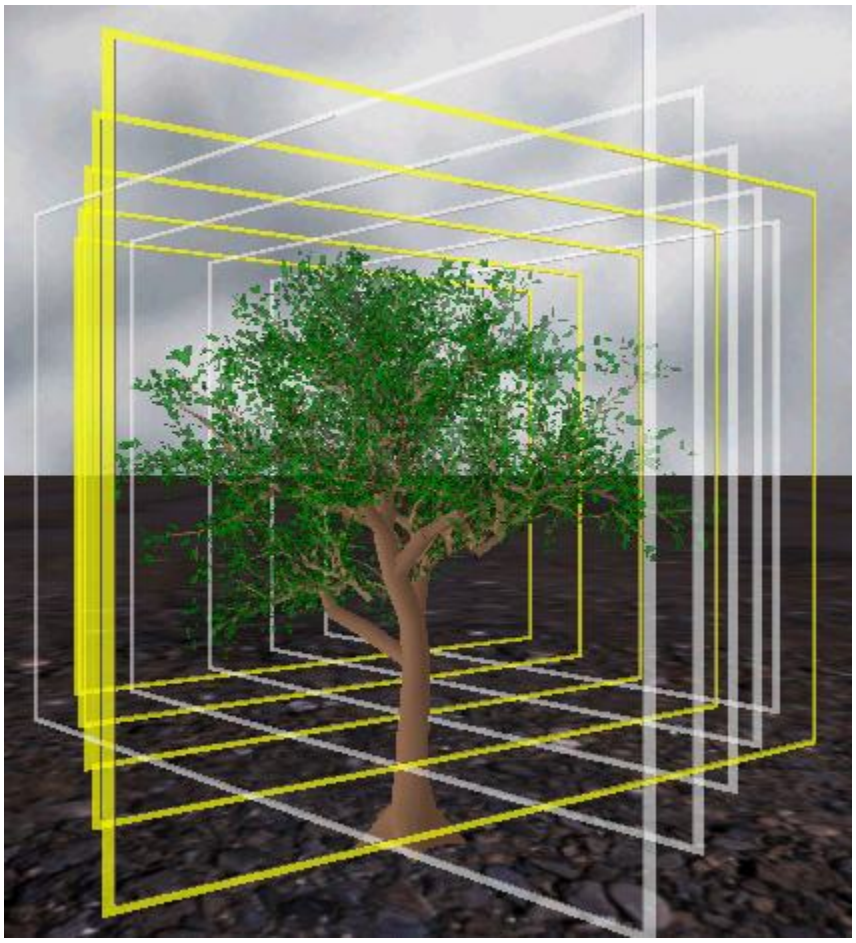

Figure 1.9: Tree is rendered by blending two 60-degree sets of slices, the borders of the quads are rendered yellow for the left set of slices and white for the right set of slices

Table 3: Advantages and disadvantages of slices

Advantages	Disadvantages
Low polygon count Looks beautiful from afar Gives an illusion of depth Minimizes the “pop” effect	Unlit – does not respond to dynamic lighting. Shading in used textures is not accurate. Textures used usually need transparency (they require an extra channel – alpha channel) Doesn’t work if the object is close enough to be seen from above or below Trunk and branches of the tree

	need to be modeled with polygons - Needs a large amount of textures to work

Implementation in Unity

Since only two sets of slices should be visible at any time, a script is needed to make other slices invisible depending on the side the viewer is seeing the tree from:

```
public class AngleBasedDisabler : MonoBehaviour {

	bool childrenActive = true;
	public float permitedViewAngle=60;
	public GameObject firstSlice;


	void Update () {
		Vector3 camerafoot = Camera.main.transform.position;
		camerafoot.y=firstSlice.transform.position.y;
		float viewAngle = Vector3.Dot(firstSlice.transform.forward,
		(camerafoot-firstSlice.transform.position).normalized);
		viewAngle = 180-Mathf.Rad2Deg*Mathf.Acos(viewAngle);

		if(viewAngle >  permitedViewAngle )
		{
			if(childrenActive)
				DisableChildren(true);
		}
		else if(!childrenActive)
			DisableChildren(false);
	}

	void DisableChildren(bool disable)
	{
	    childrenActive = !disable;
		for(int i=0; i< transform.childCount; i++)
			transform.GetChild(i).gameObject.SetActive(!disable);
	}
}
```

Blending between the slices needs to be done in post processing or using a special shader. Neither is simple to implement, so in this paper no blending is used to render a tree.

1.2. Polygonal 3D models

With all of the previously mentioned methods of rendering foliage there is one big problem – they are not 3D. Which means that the viewer, if close enough to the object, can see they are just tricks and not the real thing. Which is why the widest used method of rendering close up foliage in real time 3D applications is to render polygonal foliage – virtual polygonal models of real foliage.

This method is by far the most accurate method of rendering trees, and as the visually most appealing method, it is also the most performance expensive method.

Because of that, rendering full 3D models of foliage is done when the foliage is close enough to the viewer that the other high performance tricks wouldn't fool the viewer.

Table 4: Advantages and disadvantages of using polygonal models

Advantages	Disadvantages
+ As realistic as virtual plants get + Can respond to dynamic and static lighting + Look good from up close and from afar	- Potentially high polygon count - Require modeling and texturing - Need to optimize number of polygons and use LODs for better performance

Hand modeled plants are important, not just because of their use as 3D models in real time applications, but because most textures used by the previously mentioned methods are generated by pre-rendering such modeled trees.

There are many programs today that can be used by 3D artists for 3D modeling– 3D Studio Max, Maya, Blender, Lightwave, Modo, ZBrush and many more. These programs rely on the user's artistic ability to create models so we won't be covering them.

There are also some programs designed specifically for modeling foliage that use procedural techniques to generate foliage without a need for conventional 3D modeling such as XFrog, SpeedTree, Forester Arboretum and others ([4]). There

are even more programs that have the ability of creating procedural foliage as a part of many other things they can do such as Vue, Biosphere 3D and Unity.

This paper will concentrate on only one method, one of the earliest procedural methods of generating foliage, developed by Aristid Lindenmayer in the late 1960's - L-systems.

1.2.1. L-systems

Lindenmayer systems – L-systems for short – were conceived as a mathematical theory of plant development their emphasis on plant topology. The geometric implementations of systems were beyond the scope of the theory.

The central concept of L-systems is based on rewriting. L-systems define complex object by successively replacing parts of a simple initial object using a set of rewriting rules or productions.

The classic example of an object generated by such a system is a snowflake in Figure 1.10.

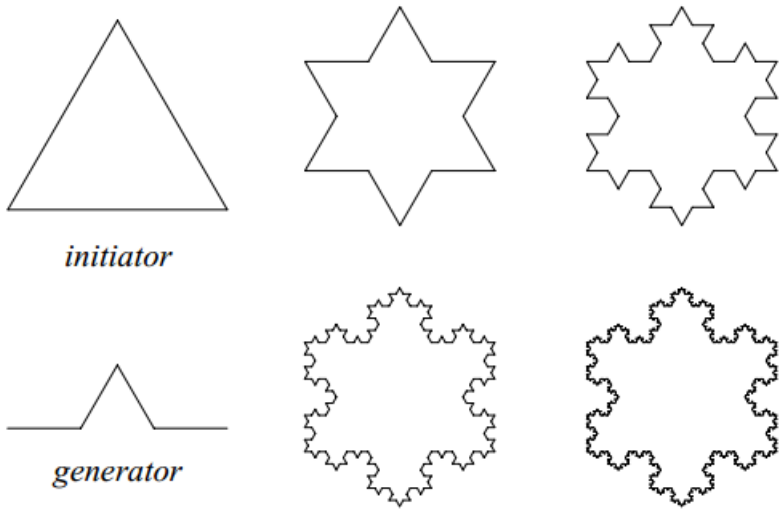

Figure 1.10: Generation of a snowflake
From left to right, from up to down – initiator, iteration 1, iteration 2, generator, iteration 3, iteration 4

One begins with two shapes, an initiator and a generator. Initiator is the starting object. In this example we replace the segments of the triangle with the generator. In every iteration of the procedure, we replace one line segment with the generator shape.

The most often used rewriting systems operate on character strings. The interest in string rewriting was inflamed in late 1950s by Chomsky's work on formal grammars, which was followed by a period of fascination with syntax, grammars and their application to computer science.

The essential difference between Chomsky grammars and L-systems lies in the method of applying productions. In Chomsky grammar's productions are applied sequentially whereas in L-systems they are applied in parallel. L-systems use iterations; every replacement in a single iteration is done simultaneously (in the manner of speaking). In the example of the snowflake, Chomsky grammar's productions would start with one line segment and replace it with the generator, and then they would take the first line segment of the resulting shape and repeat the procedure recursively.

Today there are many different types of L-systems, deterministic and stochastic, context-free and context sensitive, parametric and parameterless.

In this paper the main concentration was on stochastic parametric context free L – systems. Why? Deterministic systems with same parameters would always produce the exact same plants – stochastic systems introduce probability into the mixture and give more possibilities – more possible plants, more variations, which is why they were chosen for this work. In parametric L-systems, almost every symbol has one or more parameters that affect the operation. These parameters are associated with the symbol and the operation the symbol represents operates using them. This allows for more user control of the plant generation. Context-free L-systems were chosen because of their simplicity in comparison to context-sensitive L-systems.

A stochastic OL-system is defined by an ordered quadruplet $G = (V, \omega, P, \pi)$.

V	the alphabet of the system, consisting of variables – characters that can be replaced
ω	initiator, the string of characters defining the initial state
P	production rules (or productions for short) that define ways that variables can be replaced by combinations of constants and other variables
π	the probabilities for each production

In the tree generation, characters in the alphabet denote methods or commands for the tree generator. This L-system is parametric so value of a parameter is supplied along. These characters have special meaning to the tree generator:

F(l)	Create new branch segment the length of l.
B(n)	Go back n segments
f(n)	Create n number of leaves
w(n)	Create w number of flowers
!(r)	Change the radius of the next branch segment to r
&x(a), &y(a), &z(a),	Set new branching angle a in x,y and z axis respectably
/(d)	Set new leaf angle

The tree generation works in respect to certain rules introduced to either simplify the process or make the trees seem more realistic:

- Tree segments are straight
- New segment is always drawn from the current cursor position. When a segment is drawn cursor position moves to the end of the new segment. With B action is used the cursor moves back to a previous position.
- Each new segment is randomly rotated by +-branching divergence angle (a) in all axis
- Lengths of the daughter segments are shortened by constant ratios (lr) with respect to their mother segment.

- Widths of the branches are shortened by a constant ratio (wr) in respect to their mother segment
- The trunk of the tree is always positioned at (0,0,0), (starting cursor position)

Figure 1.11: Tree generated inside Unity using L-system implementation portrayed in Figure 1.12

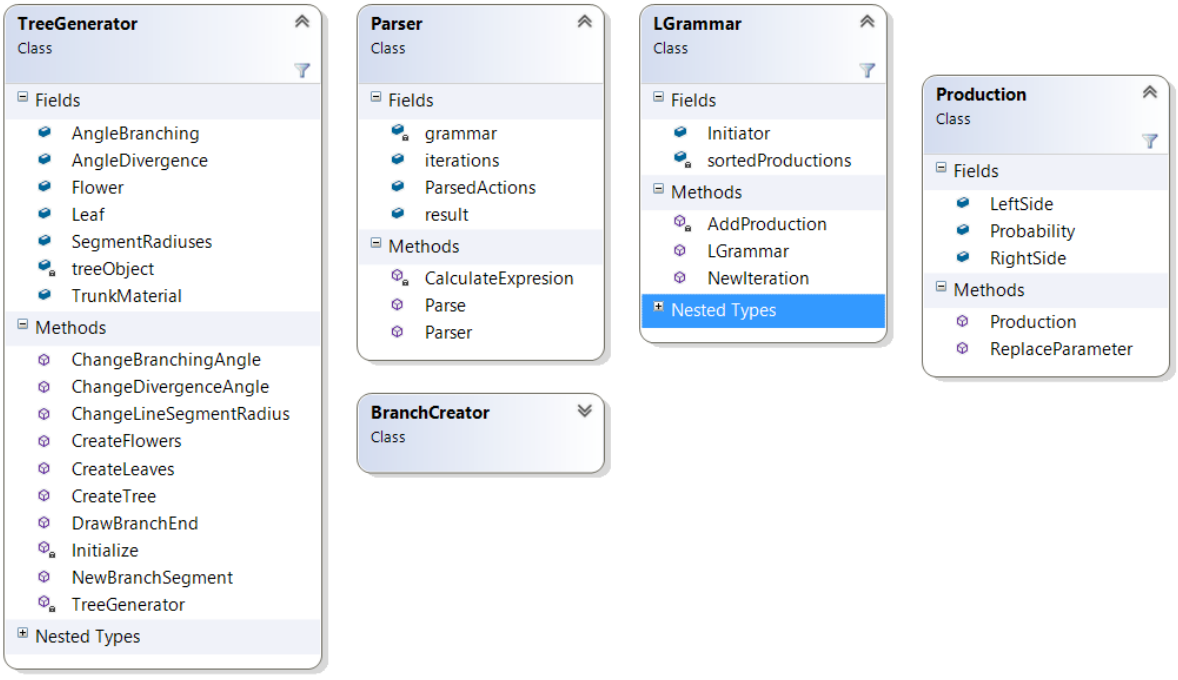

Figure 1.12: L-system implementation class diagram

Implementation in Unity

Implementation in Unity consists of few classes as can be seen in Figure 1.12.

LGrammar class holds all information about our grammar - G = (V, ω, P, π), including a list of all constants such as branch width contraction ratio (wr) and branch length contraction ratios (l1 and l2) which can then be used in the productions of the grammar. When NewIteration is called, Initiator is replaced according to our grammar and in the start of new iteration we have a new Initiator as a result of previous iteration.

It also allows adding new Productions which are described by Production class. ReplaceParameter is a method that replaces parameters in the right side of the production with a concrete number.

Parser is given a grammar and a number of iterations to run. When Parse is called, parser calls LGrammar.NewIteration required number of times and proceeds to parse the result. While parsing the parser reads characters deciding what is an action and what is a parameter. All actions are then connected to corresponding functions in the TreeGenerator as well as their parameters. The result of the parsing is a list of actions. That list of actions is then delegated to TreeGenerator (along with a few more parameters such as Tree Bark Material, leaf and flower mesh if required etc.), which upon calling its method CreateTree calls those actions. Each of the actions calculates required data and when all tree generation data is available methods of BranchCreator are called that deal with VertexCreation, UV positioning and Normals. Previous states of the L-system such as cursor position and branching angles are saved in a stack and popped from the stack when action *back(b)* is called.

The result of the process is a Tree Game Object containing generated tree mesh ready for use in game. The mesh itself is automatically saved to disc under the given name.

To aid with tree creation a Tree Generator Window was also created as a part of Unity Editor Interface as can be seen in Figure 1.13.

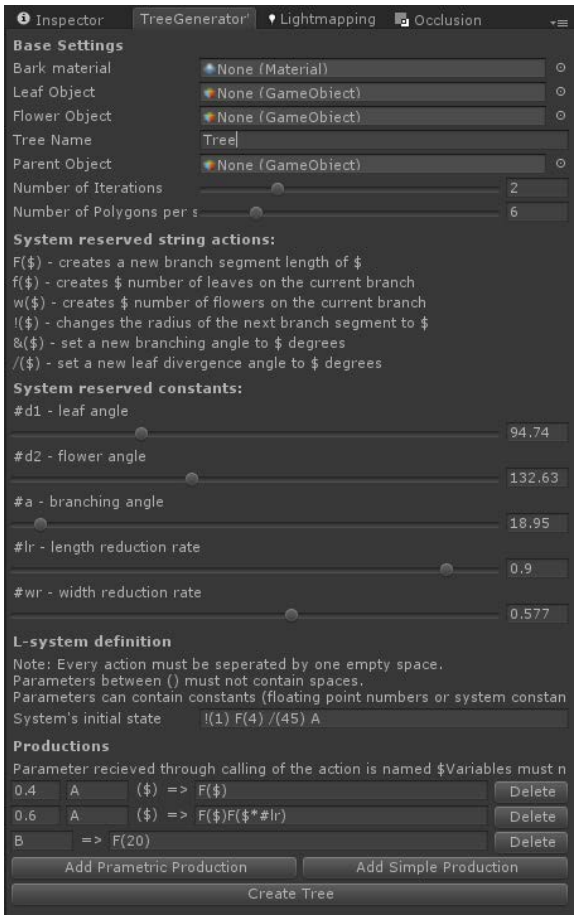

Figure 1.13: Tree Generator Editor Window

2. Performance Optimizations

Developers constantly try to make their applications more visually appealing by adding more eye candy, more objects to achieve realism and so forth. By constantly adding more polygons and textures, application's performance can suddenly drop into unacceptably low frame rates. At that point, they ask themselves, "how can we fix it? How can we improve the application performance and still preserve its visual quality?"

This chapter aims to offer a few techniques that have been developed as an answer to those demands.

2.1. Level of Detail

In CG, term level of details refers to a technique that involves decreasing the complexity of a 3D object representation as it moves away from the camera according to a certain set of rules such as distance from the camera and object importance. Decreasing the complexity of objects as they move away from the camera increases the efficiency of rendering by decreasing the workload on the graphic pipeline. The reduced visual quality of the far away objects is usually not noticeable because of their distance from the camera. [5]

When we use the term Level of Detail – we are usually referring to decreasing the geometric complexity of an object – decreasing its number of polygons. However the basic concept of LOD can be generalized to include some other techniques – such as decreasing shader complexity and texture complexity. In fact, LOD technique for textures has been used for years, only known by a different name – mip-mapping.

Creating simplified geometry to use in LOD algorithms can be done in real time using high complexity objects and then simplifying them – that type of LOD algorithms are known as Continuous LOD (CLOD). The process of creating simplified versions of an object is often complicated and time consuming - not something that can be done in real time so most real time applications use previously created LOD objects and exchange them according to the distance from the camera in discrete intervals – Discrete LOD (DLOD).

In this work, CLOD won't be covered as it is a complex method that would require a thesis of its own and is not used as often as DLOD. Further mentions of LOD will be referring to DLOD.

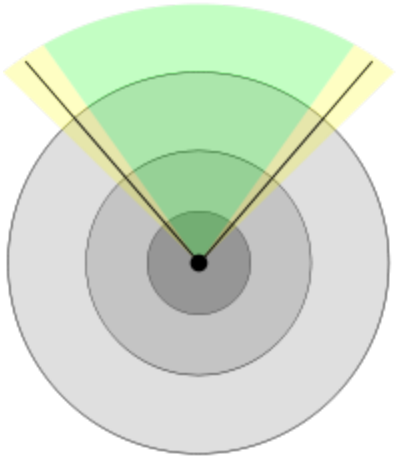

Figure 2.1: An example of various DLOD ranges. The dot in the center is the camera, the green cone representing the visible portion of the space. Darker areas are meant to be rendered with higher detail.

DLOD is based on subdividing the space into finite amount of regions, each assigned with a different LOD. The result is a discrete amount of detail levels. Because of their discrete levels smooth transition of LOD levels is difficult. Many different techniques are used to avoid visual popping such as alpha blending and morphing.

The basic concept of DLOD is to provide various models to represent the same object. LOD models can be obtained by complicated polygon reduction techniques (there are a lot of commercial software solutions for this purpose, such as Polygon Cruncher from Mootools) or created by hand.

Table 5: Visual impact comparisons of different levels of details and their vertex count

~5500	~2880	~1580	~670	140

OC is a common technique used to increase performance when dealing with foliage covered areas. Out of all objects often used in combination with LOD, a surprisingly

large amount of them are, in fact, trees. If application portrays walkable forests containing complex trees, LOD becomes a necessity rather than a convenience.

Implementation in Unity

Unity has three built in ways to utilize LOD functionality.

First, Unity terrain engine has a LOD implementation that turns objects on the terrain (such as trees) into billboards using a billboard shader after reaching a certain distance from the viewer. Parameters for it can be set up within terrain settings.

Second, is not really a true LOD solution, but works similarly. In some applications it may be appropriate to cull small objects more aggressively than large ones to reduce the CPU and GPU load. Small objects such as grass, rocks and small foliage could be made invisible at long distances. This can be achieved by setting manual per layer culling distances on the camera. One could put small objects into a separate layer and setup per-layer cull distances through script using Camera.layerCullDistances.

Third and most often used option is Unity's integrated LOD solution that makes usage of LODs simple and intuitive. It is available only in the Pro version of Unity.

To give an object a LOD functionality one needs to put LOD group (found under Component->Rendering) on the object. LOD group component allows definition of LOD distances and levels per object as well as previewing the adjustments on the go.

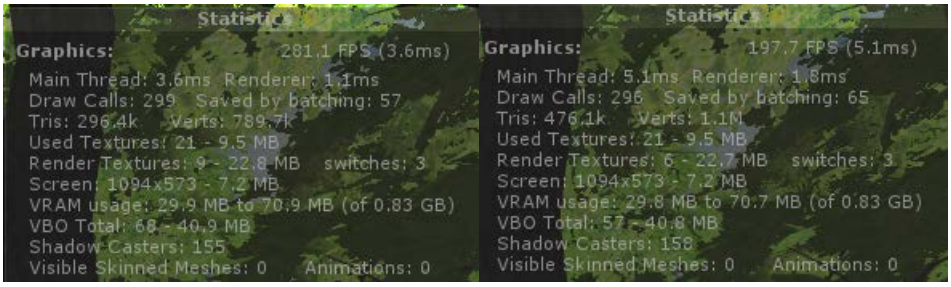

Figure 2.2: Left - forest rendered with 3 LOD levels, Right - forest rendered without usage of LOD FPS is higher while triangle count and render time are lower with the use of LOD. Overall, usage of LOD increases performance (as it should)

2.2. Occlusion Culling

Occlusion Culling (OC), also known as hidden surface removal (HSR) is the process used to determine what surfaces and parts of surfaces are not visible from a certain viewpoint. By knowing what is visible and what is not, the application can ensure only visible objects get sent to be rendered. This reduces the number of draw calls and increases the performance of the application. [6]

Occlusion culling can greatly improve performance when dealing with performance heavy area so it is an often used technique in graphic rich applications. However plants often have materials with transparency for their leaves which makes them excluded from the occlusion computation (more on that later) which means that applying them to just dense forests will not increase performance noticeably unless meshes for the tree trunk and the leaves are separated into two – one with transparency (leaves) that cannot occlude and one completely opaque (trunk) that can. To fully exploit OC to increase performance in foliage heavy areas, occluder objects are needed such as caves, hills, cliffs, boulders, and other objects large enough to occlude the plants and therefore make utilizing OC possible.

There are many different algorithms used for occlusion culling such as Z-buffering, Binary space partitioning and ray tracing. However, Unity's Occlusion Culling algorithms use conservative potentially visible sets (PVS) and Portal rendering. [7]

PVS is an occlusion algorithm that pre-computes candidates for potentially visible polygons. They are then indexed at run time in order to quickly obtain an estimate of the visible geometry. To determine PVS the camera view-space is subdivided into (usually convex) regions and PVS is computed for each region. Benefit of this method is its runtime performance – it is very fast. However pre-computed data needs to be stored to be usable at runtime so it increases the size of the application. Also preprocessing times may be long which may be inconvenient at times. The major flaw of the method being it can't be used for completely dynamic scenes.

Conservative PVS overestimates visibility ensuring that nothing visible will ever be omitted. That means that at time more things than are actually visible might be sent to render. However, this is preferable for graphic consistency rather than aggressive algorithms, which ensure that no invisible polygons are sent to render, though they also might fail to render visible polygons.

Portal rendering, as a sub type of PVS, divides the scene into cells/sectors (rooms) and portals (doors). Each cell is a subdivision of the entire bounding volume of the scene and together they form a binary tree. Portal rendering is able to create portals at runtime and is more accurate than the general PVS but it also requires more processing time.

To use Occlusion Culling in Unity static *Occluders* and *Occludees* must be defined. *Occluders* are objects that are able to occlude other objects; *Occludees* are objects that are able to be occluded by other objects.

Usually, most static game objects will be marked both as *Occluders* and *Occludees*. However in cases where an object is translucent or its material contains transparency it cannot occlude other objects so it should not be marked as an *Occluder*. Also in cases where an object is really small and unlikely to occlude other objects, it shouldn't be marked as an *Occluder*. Not marking such objects as *Occluders* helps reduce the computation.

Unity allows users to choose the type of OC that will be used. Available techniques are "PVS only", "PVS and dynamic objects" and "automatic portal generation".

PVS only culls only static objects, dynamic objects will be culled based only on the view frustum. It is the fastest of the techniques but since dynamic objects are not culled it is only recommended for applications with few moving objects.

PVS and dynamic objects is a method where static objects are culled using pre-computed visibility and dynamic objects are culled using portal culling. This technique is a good balance between runtime overhead and culling efficiency. However it does not allow portals to be opened or closed at runtime.

Automatic Portal Generation is a full Portal occlusion solution. Both static and dynamic objects are culled using portal culling. It is the most accurate method, allows portals to be opened and closed at runtime but also it is the most expensive method.

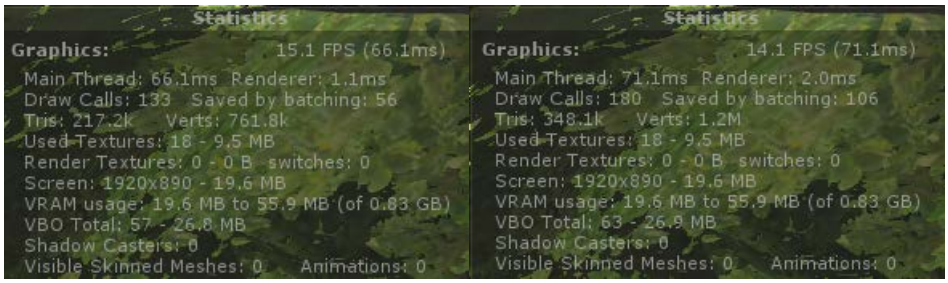

Figure 2.3: Left - forest scene using OC, Right - same forest scene without OC
A small part of the forest is hidden by a hill, OC knows this so it doesn't send those trees to render which we can see by the lowered draw call count, triangle count and render time

2.3. Performance considerations

Application's frame rate is dropping down and making a real time application not really real time. It is a scenario that happens often. To fix the problem one needs to identify it first. What is causing the problem? Is the performance drop caused by CPU or GPU? Optimization strategies for GPU and CPU can be quite different, sometimes even opposite. [8] Naturally, it is important to identify which of the two is causing the problem, typical bottlenecks and ways to check for them are as follows:

GPU

- Exceeding the fill rate limit
- Exceeding memory bandwidth
- Too many vertices to process

CPU

- Too many Draw Calls
- Rendering is not the problem, something else is causing the problem e.g. Physics is taking up too much CPU time
- Too many vertices to process

2.3.1. GPU optimization

Fill rate

Fill rate is the amount of pixels a graphics card can render and write to graphic memory in a second. There is no agreement how to calculate and report fill rates so there isn't a number you can check in your statistics.

If the game's performance increases dramatically in a lower graphics resolution (smaller window size) then fill rate is likely the issue.

Scene complexity increases by overdrawing, which happens when an object is drawn to the frame buffer and then, another object is drawn on top of it. To fix fill rate issues one has to minimize overdraw, with smart occlusion culling algorithms, drawing objects on top of other objects that completely cover them up, is no longer a big issue. Nowadays bigger issue lies in usage of transparent textures and multi-pass shaders.

If you have a fill rate issues:

- Use occlusion culling
- Lower the usage of transparent textures
- Avoid multi-pass shaders

Memory bandwidth

Memory bandwidth is the rate at which data can be read or stored into memory (usually expressed by bytes/second). GPU cannot render data it doesn't have. Which means if GPU is receiving a lot of data to render, saturating the bandwidth, the performance of the application drops.

Memory bandwidth issues aren't as likely today as they once were as GPU stores most if not all persistent data in its own RAM memory. However as there are still ways to saturate the bandwidth, it is best to not overlook the possibility.

Polygon count

A question often asked is “How many polygons?” How many polygons can you render per frame and keep application performance decent? How many polygons should certain objects have? Artists in particular often ask this question. When they are hired to model objects – the effort and time they need to put in is proportional to polygon count the objects should have.

There is only one answer: “It depends.” [9]

The question is too vague leaving out too many important parameters. It depends on visual style of the application, the type, capabilities of the engine, target hardware.

What will be visible per frame? If it is an application simulating forests, how dense will the forests be? How large will the forests be? From which perspectives will the viewer be able to see the trees? If the viewer will not be able to walk around the forest but see it from afar the trees don’t need to be models, they can just be billboards.

For example, if the application was an architectural visualization of a park with dozens of trees visible at any time – or if it was a foresting training program that featured only one tree at a time. In the latter you would end up with one very detailed high poly tree, whilst in the former you would have neither need nor processing power to display that many high detail trees.

Knowing application’s target hardware is the most important parameter. Obtaining the exact count of polygons that the target hardware is able to render per frame in decent performance is possible. Test it!

However, even when presented with a definite amount, determining what objects should have what amount of polygons is hard. There isn’t a math formula that can help. Making visually appealing real time applications is considered an art and as such is not something easy to mathematically express. Usually developers go by reference – how much more important is an object in comparison to another. How many polygons per object does a similar application have?

Judging by existing applications for mobile platforms using between 300 and 1500 polygons per mesh gives good results. For desktop platforms this range rises to 1500-7000. [9] [10]

In the end the best practice is to minimize the amount of polygons while maintaining the adequate graphical quality of the application. If a certain scene is polygon performance heavy either lower the amount of objects or their amount of polygons – don't forget to utilize LODs!

2.3.2. CPU optimizations

Draw Calls

Every time a model is drawn to the screen CPU has to issue a draw call to the GPU. Unfortunately it is not as simple as 1 call per model – rather, it's 1 call per material within the model. But that's not the end of it, if you are using per pixel lighting you get a draw call per light pass, or if you have dynamic reflection (or similar) you'll be rendering the scene at least twice potentially doubling draw calls.

Unity tries to help with this by material batching – it tries to combine models using the same material so they would be drawn with just one draw call. To minimize the draw calls reuse the same materials as often as you can.

Sometimes when dealing with different objects fundamentally the same, such as trees, we would use the same material to render them but the textures used are different. Maple tree has maple leaves, palm tree has palm leaves. But a leaf is a leaf – same material, different texture. If maple trees are often close to the palm trees we could improve the performance by combining the textures used in those two trees into a texture atlas. A Texture atlas is a large texture made from multiple textures. Materials that use that texture atlas render only a part of it – one sub texture of the atlas. However the atlas is treated like one texture which means if two same materials are rendering different parts of the atlas (because of UV positioning), they are treated as the same material which would allow Unity to material batch those objects into one draw call. That is a technique often used for GUI elements but

can be applied to anything as long as it makes sense to group different textures together.

If we never rendered maple and palm trees in the same frame, grouping their textures would not only be pointless but counterproductive.

Processor Time

Sometimes the problem to game performance has nothing to do with graphics. Maybe there are too many complex operations in the application. Maybe the performance of the application is not limited by GPU but CPU instead.

Best way to determine that is to use the built in Profiler. Unity Pro has a powerful profiler that can display exact processing times of all methods in the application. If it takes too much time to calculate physics, too many vertices to send to the GPU or anything else – one look to the profiler and you can tell exactly what method is causing the problem.

3. Tips and Tricks

3.1. Unity supported image formats

Images are used as textures for rendering foliage both for billboards and for polygonal plants. Images are imported through Unity asset pipeline for use in Unity. Unity imports the images according to import settings specified but doesn't change the original images – any adjustments made in the process of importing are nondestructive. Unity supports all of the most often used formats such as PSD, TIFF, JPEG, PNG, GIF, BMP, TGA, IFF and PICT. Photoshop files (PSD) and Tagged Image Files (TIFF) are imported with layers automatically flattened (original files stay unchanged). [11]

Many other image formats are (allegedly) also supported but Unity documentation fails to mention which ones.

For billboarding purposes and for texturing leaves transparent textures are required. Image formats that support an alpha channel are PSD, TIFF, PNG, TGA and BMP. TGA supports an alpha channel when saved in 32 bit color depth and BMP can be stored both at different color depths and using different bitmap file format (some including alpha information) but Unity will recognize neither transparency data from TGA nor BMP images

GIFs do not fully support transparency, resulting in some pixels being completely transparent and others being completely opaque when imported in Unity. (Animated GIFs are not supported in Unity, so any animation data will not be imported)

In conclusion only image formats truly usable for saving and using transparent images within Unity are PSD (which is most often used for artist created layered images), TIFF and PNG.

Unity allows creation of custom asset importers so it is possible to code in support for unsupported image formats if one so desires.

3.2. Unity Default Shaders usable for foliage rendering

When rendering foliage in Unity, whether the plant is geometrically modeled or just a billboard, it cannot be rendered to screen without an assigned material. Material is an instance of a certain shader with supplied required textures and parameters.

Unity comes bundled with many shaders usable for rendering foliage which makes it difficult at times to choose the right one. Information in tables below might be useful when trying to find the right shader. [12]

Self-illuminated shaders, reflective shaders and some others were not covered because of their limited usage with plants. (e.g. Glowing plants and reflective plants are not common on Earth)

Lightmapping

All shaders that come with Unity support lightmapping out of the box. [13] Whether an object casts a shadow and receives shadow is decided in Mesh Renderer properties of each object (only unlit shaders don't receive any shadows even when

marked as receiving shadow). When baking lightmaps Unity takes into account all lightmapping enabled lights and all static objects, and proceeds to bake light maps. This process can be very time consuming, therefore it is run by Beast (Unity's light mapping tool) on a separate thread so one can still continue to work in Unity while Beast does its magic. Lightmaps are sampled per pixel and applied to the final appearance of the object regardless of the shader type used but dependent on the size of the lightmap and UVs used to apply it.

When choosing a shader keep in mind that lighting model the shader uses is only relevant for overall appearance of the object and its response to dynamic lighting – not static lighting a.k.a. baked lighting. (Again, unlit shaders are unlit, including static lighting)

Lighting model

Unity allows users to choose one of three rendering types it supports when rendering the application: Deferred Lighting, Vertex Lit and Forward rendering. In vertex lit rendering each object is generally drawn just once, while in deferred lighting they are drawn twice, no matter the lights affecting it. Difference in shader performance in those modes mostly depend on the textures used and calculations performed.

When we talk about shader lighting models it only makes sense to talk about them in conjunction with forward rendering – which is the default rendering model used in Unity. In forward rendering performance of the shader depends on both the shader itself and lights in the scene. This is where shader lighting model comes into play – there are three types of shader lighting models: unlit, vertex lit and pixel lit.

Unlit shaders are not affected by lighting (not even baked lighting), they are performance wise the cheapest.

Vertex lit shaders calculate light based on the mesh vertices using all lights at once. Because of this, no matter how many lights are affecting the object it will be drawn just once.

Pixel lit shaders are the most expensive because they calculate the final lighting of each pixel that is drawn. This means the object is drawn once to get the ambient and main directional light and once for each additional light shining on the object (lights that are used only for baking lightmaps are not included in the calculations). Pixel lit shaders are popular because they are able to display pixel based rendering effects such as light cookies and normal mapping which makes objects more visually appealing.

When deciding which shader to use it is important to think about the lighting conditions that shader will be used with. If the object will have to respond to multiple dynamic lights it might be wise to use a vertex lit shader instead of a pixel lit shader. However if there won't be more than just a few dynamic lights affecting the object using pixel lit shader most likely won't impact performance too much.

Terminology and abbreviations used in tables

Lighting model	**u** -Unlit, **v** - Vertex lit or **p**- pixel lit (see lighting model for details)
Transparency	Shader can either support transparent textures fully (each pixel can have different levels of transparency), by using clipping (*) – each pixel is either fully transparent or fully opaque, or not support it at all – everything is drawn fully opaque
Normal	Shader has a slot for normal map and supports normal mapping
Specular	Shader supports a specular map either by using alpha information in the diffuse texture or by having a separate slot for a specular map
Extra	Any extra features the shader supports
Performance	Shader overall performance from 1 (best performance) – 10 (worst performance)

3.2.1. Opaque shaders

Shaders in this category are usable for rendering tree trunks, branches and other plant parts which are completely opaque and have no transparency.

Shader Numberings:

1) Vertex-lit
2) Decal
3) Diffuse
4) Diffuse Detail
5) Specular
6) Bumped Diffuse
7) Bumped Specular
8) Parallax Diffuse
9) Parallax Bumped Specular
10) Nature Terrain Diffuse
11) Nature Terrain Bumped Specular
12) Nature Tree Creator Bark
13) Nature Tree Soft Occlusion Bark
14) Unlit Texture

Table 6: Comparison of opaque shaders available in Unity

Shader / Feature	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Lighting Model	v	v	p	p	p	p	p	p	p	p	p	p	p	u
Normal	-	-	-	-	-	+	+	+	+	-	+	+	-	-
Specular	-	-	-	-	+	-	+	-	+	-	+	+	-	-
Performance [14]	2	2	3	3	4	4	6	7	8	4	7	6	?	1
Extra		1		2				3	3	4	4		5	

Note: Mobile versions of shaders behave like their normal counterparts but use simplified calculations (approximate specular, no per material color support, etc.).

? – there is almost no information on Nature Tree Soft Occlusion shaders and how they work

1 – an extra texture needs to be supplied that will be drawn on top the main texture, used to add details such as graffiti, scars etc.

2 – an extra texture needs to be supplied that will become visible blended along with the main texture when an object is up close. This is to avoid seeing only the blurred main texture when the object is close. This adds more detail in variety so the object won't look blurred up close.

3 – height map texture needs to be supplied. Adds an extra dimension of depth. Can be used for tree trunks that have deep gashes to better represent the difference between creases and bumps.

4 – can use multiple main textures and blend between them according to the control texture. Makes it possible for tree trunk texture to change with e.g. tree height.

5 – calculates ambient occlusion which makes trees look more realistically lit

3.2.2. Transparent shaders

Shaders in this category are usable for rendering leaves, flowers and bushes that require usage of transparent textures. They either support texture transparency fully allowing for different levels of transparency, or they support alpha clipping which renders pixels as either completely transparent or completely opaque.

Shaders from the Transparent group are identical to their normal opaque counterparts, the only difference being support for transparency. Shaders from Transparent Cutout group are also identical to their opaque counterparts however, unlike their transparent counterparts; cutout shaders allow no partial transparency. Cutoff threshold allows adjustment of what pixels are rendered as opaque and what are rendered as fully transparent. Cutout shaders have a slightly better performance than transparent shaders.

Unlit transparent and transparent cutout are the same as ordinary diffuse transparent shaders the difference being they are unlit. (Best performance of the transparent shaders)

3.3. Unity Tree Creator

Unity has a built in Tree Creator which allows creation and editing of trees procedurally. The resulting trees are meshes which can be used as normal Game Objects. Trees created with Unity Tree Creator can respond to wind which makes them seem more alive when viewed at runtime.

Tree creator inspector is split into three different panes: Hierarchy, Editing tools and properties. [15]

Hierarchy view shows a schematic representation of the tree where each box represents a group of nodes. Every node has properties that can be changed.

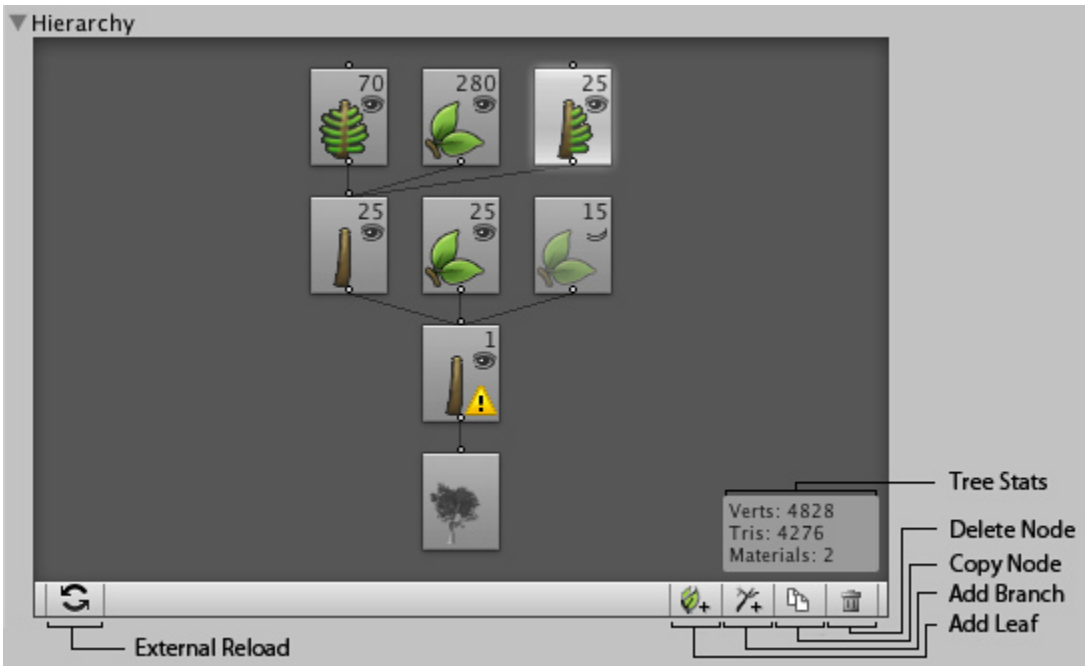

Figure 3.1: This hierarchy represents a tree with one trunk, 25 child branches. Child branches have in total 70 fronds attached, 280 leaves and 25 fronds with branches.

There are 5 different types of nodes:

1. Root node – the starting point of the tree holding the global parameters for the tree
2. Branch node – first branch node represents the trunk, others create child branches
3. Leaf node – it is a final node; no other nodes can be attached to them. It can be used for leaves, flowers, seeds and similar
4. Frond node – similar node to branch node but has specific properties to enable easier creation of fronds
5. Branch + Frond node – a node that has both branch and frond properties

Editing tools allow manual adjustment of the tree – moving and rotating nodes in real time. Editor tools also allow free hand drawing which allows drawing an exact spline for branches to follow.

There are many properties to adjust and play with until creating the desired tree. E.g. in leaf node properties, geometry type can be chosen between plane, billboard, cross, tricross and a custom mesh.

It is a great free alternative to commercial plant generation software and is simple to use. Also it comes in handy for those lacking artistic skills and knowledge in 3D modeling to create good looking hand modeled trees.

Conclusion

Computer graphics is a fast evolving area and every passing year graphic applications are expected to look better than the previous year. New methods of achieving new heights in visual fidelity and improvements of existing methods are introduced every year.

Rendering of plants is important in many areas of computer graphics and because of complexity of their accurate visualization many real time methods of rendering foliage have been developed over the years. Billboarding, directional billboards, slices and polygonal models, are just the most commonly used methods.

Procedural generation of polygonal models is such a vast area that hundreds of papers have been written describing many methods, L-systems being just one of the most famous methods.

Performance improvements are necessary to get the most out of current hardware and software. Special techniques have been developed so to help decrease the cost of visual fidelity – to decrease the work graphics card has to do. Most often used ones being Level of detail and occlusion culling.

This paper touches briefly in all of those areas, concentrating on their implementation in Unity, but not discussing them in depth. Even after covering all these methods and techniques, there are even more left to cover. Lighting foliage and imbedding information such as ambient occlusion lighting data is a big factor in making foliage visually more appealing. So are many different complicated shaders that give life to the plants, containing approximations of phenomena such as sub-surface light scattering.

There is much more to be said both on the subject of rendering and procedural modeling of plants, and implementation of them in Unity. One paper is not enough to cover all of the methods and that is regretful. There is too much left unsaid, and hopefully in the future works some of the holes left by this one will be filled.

List of Figures

List of Tables

References

[1] Information Coding/ Computer Graphics, ISY, LiTH, "Billboards," [Online]. Available: http://www.computer-graphics.se/TSBK07-files/pdf13/8c.pdf. [Accessed 16. 5. 2013.].

[2] A. R. Fernandes, "Billboarding Tutorial," [Online]. Available: http://www.lighthouse3d.com/opengl/billboarding/. [Accessed 16. 5. 2013.].

[3] A. Jakulin, "Interactive Vegetation Rendering with Slicing and Blending," *EUROGRAPHICS,* 2000.

[4] "Plant Generation Software Packages," [Online]. Available: http://vterrain.org/Plants/plantsw.html. [Accessed 15. 5. 2013.].

[5] "Level of detail," [Online]. Available: http://en.wikipedia.org/wiki/Level_of_detail. [Accessed 23. 5. 2013.].

[6] "Hidden surface determination," [Online]. Available: http://en.wikipedia.org/wiki/Hidden_surface_determination. [Accessed 23. 5. 2013.].

[7] "Occlusion Culling," [Online]. Available: http://docs.unity3d.com/Documentation/Manual/OcclusionCulling.html. [Accessed 23. 5. 2013.].

[8] "Optimizing Graphics Performance," [Online]. Available: http://docs.unity3d.com/Documentation/Manual/OptimizingGraphicsPerformance.html. [Accessed 21. 5. 2013.].

[9] "How many polygons is to much?," [Online]. Available: http://answers.unity3d.com/questions/19379/how-many-polygons-is-too-

much.html. [Accessed 20. 5. 2013.].

[10] "Modeling Optimized Characters," [Online]. Available: http://docs.unity3d.com/Documentation/Manual/ModelingOptimizedCharacters.html. [Accessed 22. 5. 2013.].

[11] Unity Technologies, "Asset Workflow," [Online]. Available: http://unity3d.com/unity/workflow/asset-workflow. [Accessed 10. 6. 2013.].

[12] Unity Technologies, "Built In Shader Guide," [Online]. Available: http://docs.unity3d.com/Documentation/Components/Built-inShaderGuide.html. [Accessed 13. 6. 2013.].

[13] Unity Technologies, "Lightmapping In Depth," [Online]. Available: http://docs.unity3d.com/Documentation/Manual/LightmappingInDepth.html. [Accessed 13. 6. 2013.].

[14] Unity Technologies, "Performance of Unity Shaders," [Online]. Available: http://docs.unity3d.com/Documentation/Components/shader-Performance.html. [Accessed 14. 6. 2013.].

[15] Unity Technologies, "Tree Creator Structure," [Online]. Available: http://docs.unity3d.com/Documentation/Components/tree-Structure.html. [Accessed 13. 6. 2013.].

[16] "Fillrate," [Online]. Available: http://en.wikipedia.org/wiki/Fillrate. [Accessed 25. 5. 2013.].

[17] "Geometric modeling of plants," [Online]. Available: http://vterrain.org/Plants/Modelling/. [Accessed 15. 5. 2013.].

[18] "Portal Rendering," [Online]. Available: http://en.wikipedia.org/wiki/Portal_rendering. [Accessed 24. 5. 2013.].

[19] "Potentially visible set," [Online]. Available: http://en.wikipedia.org/wiki/Potentially_visible_set. [Accessed 24. 5. 2013.].

[20] R. Stirling, "Yes, but how many polygons?," [Online]. Available: http://www.rsart.co.uk/2007/08/27/yes-but-how-many-polygons/. [Accessed 26. 5. 2013.].

Sažetak

Generiranje modela biljaka i njihov prikaz u stvarnom vremenu

U radu se obrađuje prikaz biljaka u stvarnom vremenu upotrebom raznih tehnika, generacija modela biljaka proceduralno upotrebom L-sustava te optimizacije prikaza biljaka da bi se postigle što veće performanse. Metode optimizacije uključuju razine složenosti (LOD) i okluziju. Sve tehnike i primjeri su napravljeni i testirani upotrebom grafičkog pogona Unity.

Ključne riječi: *prikaz biljaka, generiranje modela biljaka, optimizacija prikaza, razine složenosti (LOD), L-sustavi, Unity, uklanjanje zaklonjenih objekata, plakatiranje*

Abstract

Real time rendering of plants

Problematic described is real time rendering of plants using different methods (billboarding, directional billboards, slices), generating plant models procedurally using L-systems and render optimization for improving real time performance. Optimization methods include levels of detail (LOD) and occlusion culling. All methods and examples are made and tested in 3D graphics engine Unity.

Keywords: *rendering of plants, plant generation, rendering optimization, Levels of detail (LOD), L-systems, Unity, Occlusion culling, billboarding*