Simulation of Tree Growth

Eric Ekström

2020-12-01

Abstract

Contents

1 Introduction

2 Background

TODO!

2.1 Tree Structure and Terminology

This section covers the terminology used when describing trees. The choice of language is based on the description of trees by (Barthélémy and Caraglio, 2007).

A tree consists of a number of nodes. Each node supports nodes further out in the tree.

- A *node* is the main component of the tree. It supports at least a bud, another node or a leaf. A node combined with an internode is called a *metamer*. (Barthélémy and Caraglio, 2007)
- An *internode* is the section of stem between two nodes.
- The *lateral branch* is a supported node that is connected at an angle to the supporting node.
- The *main branch* is a supported node that has the same angle as the supporting node.
- *Buds* are potential points for expansion of new nodes. A lateral bud is located at the node that already has a main branch. An apical bud is located at a node that has no branches to support.

For this project, a tree grows by shooting a number of new metamers from a bud. This is called rhythmic growth and is the more commonly observed method of growth in nature. (Barthélémy and Caraglio, 2007) The fate of a bud and the number of new metamers depends on a number of factors. For this project, only a simulated light resource was considered.

A node is given an order based on the number of node below it. A node without a supporting node has order 1.

2.2 Volumetric Light Scattering

Volumetric light scattering (also called ”godrays” or ”crepuscular rays”) is a light phenomenon that occurs when the mixture of gas and molecules in the air is just right. This causes light to be rayleigh-scattered and creates visual sun beams through the air. (See Figure 1.)

Figure 1: An example of volumetric light scattering. (Sunshine Kodai Kanal, Manoj K Racherla, 2013)

3 Method

3.1 Tree Generation

TODO: vad är ett träd dåååå

Algorthim outline

The tree generation algorithm used the same basic steps as described by Wojciech Palubicki. (Palubicki et al., 2009)

1. Calculate the amount of light resources available in the tree.
2. Redistribute light resource and create new nodes.
3. Update the branch radii for each node.
4. Calculate which nodes are a liability, and shed these.

Calculation of light resources

Light resources are collected from each node and saved in the base of the tree. Each node contribute with resources if it has either a lateral or apical bud. The size of the contribution is calculated according to Equation 1 as proposed by Yi et al. (2018). T and T_0 is the current time and the creation time of the node respectively, l is the light that hits the current node based on conditions around it and σ is a controllable parameter.

$$Q = l^{(T-T_0)^\sigma} \tag{1}$$

Each node also has a cost associated with it. This is the amount of resources it takes to keep the node alive as discribed by Měch and Prusinkiewicz (1996). The equation for calculating this factor is shown in Equation 2. *BM* and BE are control parameters. This factor is subtracted from the light resources in each node before the result is propagated to the supporting node.

$$BM * \frac{radius}{default\_radius}^{BE} \tag{2}$$

Shadow propogation for node light calculation

The variable l in equation 1 is the amount if light that reach a node and is calculated by considering the nodes already around it. This is done by dividing the space into *voxels* of fixed size. Each voxel saves a float value describing how shadowed the point is. When a new node is added to the tree, it also updated the voxels around it. Specifically, a pyramid of voxels (with the voxel containing the new node at the top) are updated according to Equation 3. a and b are controllable parameters and q is the number of voxels down from the top of the pyramid. (Palubicki et al., 2009)

$$s = s_{prev} + a * b^{-q} \tag{3}$$

Redistribution of light resources

An adapted version of the Borchert-Honda (BH) model (Borchert and Honda, 1984) was used to redistribute resources to each node. The BH model was originally developed to control distribution of growth-inducing resources. Palubicki et al. (2009) adapted the BH model by using the collected light resources to guide the redistribution of resources. Equation 4 and 5 show how the light resources are distributed between the main branch and the lateral branch based on the amount of resources collected by each node. Q_m and Q_l is the amount of light resources collected by each of the branches and λ is a parameter that can be set by the user.

$$v_m = v\frac{\lambda Q_m}{\lambda Q_m + (1-\lambda)Q_l} \tag{4}$$

$$v_l = v\frac{(1-\lambda)Q_l}{\lambda Q_m + (1-\lambda)Q_l} \tag{5}$$

The number of new metamers created from a bud is the integer part of the amount of resources that reach the bud. Three factors are considered when deciding the direction of the new branch; direction of the parent node, the optimal growth direction, and a tropism factor. The tropism factor, τ, simulate how much the branches bend down due to gravity. The optimal growth direction is found as the neighboring shadow voxel with the lowest value. These three factors are combined to the final growth direction according to Equation 6. (Palubicki et al., 2009)

$$growth\_dir = 0.8 * parent\_dir + 0.2 * optimal\_dir + \tau \tag{6}$$

Update branch radii

The radius of each branch is updated to support the weight of the branches above it according to Equation 7, where r_m and r_l are the new radius of the supported branches. If no supported branch is present, the default branch radius is used. (Měch and Prusinkiewicz, 1996)

$$r = \sqrt{r_m^2 + r_l^2} \tag{7}$$

Calculate shedding branches

A branch is shedded if the light resource flowing through it is negative and the number of supported buds are less than a threshold. (Měch and Prusinkiewicz, 1996)

3.2 Volumetric Light Scattering

Volumetric light scattering was done as a post-process shader using the method described by Kenny Mitchell (Mitchell, 2007).

The post-process shader combines the normal render pass with a render pass where the sun is rendered as a single color, and all other objects are then rendered black. This gives an image with color only where the sun is visible from the cameras perspective.

The post-process shader sums samples of the volumetric light scattering pass from the current fragment point to the position of the sun. The samples are exponentially weighted where the samples close to the sun are weighted higher. The equation used for weighting samples is shows in Equation 8.

$$L = exposure * \sum_{i=0}^{n} decay^i * weight * \frac{sample}{n} \tag{8}$$

n is the number of samples between the fragment position and the sun position, *decay* is the exponentially decaying factor that dissipated the contribution from each sample. The *exposure* and *weight* factors are simply scale factors that increase or decrease the final light.

The final factor L for each pixel is then added to the final color of the normal pass for that pixel. The shader code used for volumetric light scattering can be found in appendix A.

3.3 Shadow Mapping

Shadow mapping is a popular technique for generating good shadows in a scene. It works in two steps; first render the scene from the perspective of the light source, then render the scene as normal using the first render to calculate which fragments are in shadow. (Ragnemalm, 2015)

The purpose of the first render pass is to generate a map of depth values for each point that the camera can see. These depth values are then used in the second pass to calculate shadows. If the distance to the camera from the current fragment is greater then the depth value saved in the depth map, there is another fragment somewhere in between the fragment and the sun and the fragment is therefor in shadow. (Ragnemalm, 2015)

Implementation

A framebuffer with only a depth attachment was used to generate the depth map. Since the only part of interest is the depth values, the used fragment shader can be empty and just write the depth values to the z-buffer. The vertex shader is the same as would be used in a normal scene.

The implementation for this project only create shadows from directional light sources. Because of this, a orthographic projection matrix was used. This gives a more realistic result by pretending that all light hits the scene as parallel lines. The position of the light source still have to de specified since it is needed to calculate the distance from a fragment.

This method will generate a number of different artifacts that have to be addressed. The first and most obvious is "shadow acne". This is caused by the fact that the resolution of the shadow map is limited and there is more than one fragment on a surface that sample from the same point in the shadow map. Even though all the fragment on a surface should be lit, only the fragments that happen to have its distance saved in the shadow map are actually lit. This can be solved by adding a small offset when comparing the distance with the depth map.

The next problem that has to be handled is the fragments that are outside the depth map. here we simply set the texture as "clamp to edge" and set the border of the depth texture to 1. The same problem arise when a fragment is beyond the outer limit of the projection matrix. The solution is to set the shadow to 0 if the distance is greater than the furthest allowed distance.
For a full implementation of the shader code used for shadow mapping, see appendix B.

4 Result

Tree Parameters

Table 1 shows the parameters that were used and how they affect the look of the tree.

Name	Effect	Reasonable range
lambda, λ	Branch bias. Larger value gives priority to main branches.	0.4 - 0.5
sigma, σ	Light conversion rate. Larger values increase the amount of light an older node can pick up.	0 - 1
tropism, τ	Larger value bend branches more towards the ground.	0.01 - 0-5
BM	Branch maintenance coefficient	0.01 - 0.5
BE	Branch maintenance exponent	>1
N_{min}	Threshold for branch shedding.	20 - 50

Table 1: Table of used tree parameters.

5 Discussion

shadow map - resolution

5.1 Improvements

References

Daniel Barthélémy and Yves Caraglio. Plant architecture: a dynamic, multilevel and comprehensive approach to plant form, structure and ontogeny. *Annals of botany*, 99(3):375–407, 2007.

Rolf Borchert and Hisao Honda. Control of development in the bifurcating branch system of tabebuia rosea: a computer simulation. *Botanical Gazette*, 145(2):184–195, 1984.

Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, 1996.

Kenny Mitchell. Chapter 13. volumetric light scattering as a post-process, 2007. https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-13-volumetric-light-scattering-post-process.

Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics (TOG)*, 28(3):1–10, 2009.

Ingemar Ragnemalm. *So How Can We Make Them Scream.* Bokakademin, 2015.

Lei Yi, Hongjun Li, Jianwei Guo, Oliver Deussen, and Xiaopeng Zhang. Tree growth modelling constrained by growth equations. In *Computer Graphics Forum*, volume 37, pages 239–253. Wiley Online Library, 2018.

A Shader code for volumetric light scattering

```
vec2 delta_tex_coord = vec2(out_tex_coords - sun_pos);
delta_tex_coord *= 1.0 / float(NUM_SAMPLES) * density;

vec2 tex_coord = out_tex_coords;

float illumination_decay = 1.0;
vec4 godray_color = vec4(0,0,0,0);

for(int i = 0; i < NUM_SAMPLES; i++)
{
    tex_coord -= delta_tex_coord;

    vec4 s = texture(god_ray_pass, tex_coord);

    s *= illumination_decay * weight;

    godray_color += s;

    // Update exponential decay factor.
    illumination_decay *= decay;
}

return godray_color * exposure;
```

B Shader code for shadow map calculation

```
vec3 proj_coords = frag_pos_light_space.xyz / frag_pos_light_space.w;
proj_coords = proj_coords * 0.5 + 0.5;

float current_depth = proj_coords.z;

float bias = max(0.005 * (1.0 - dot(normal, light_dir)), 0.00005);
float shadow = 0.0;
vec2 texel_size = 1.0 / textureSize(shadow_map, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float depth = texture(shadow_map, proj_coords.xy + vec2(x, y) * texel_size).r;
        shadow += current_depth - bias > depth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;

if(proj_coords.z > 1.0)
    shadow = 0.0;

return shadow;
```