EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

GPU-Based Wind Animation of Trees

Jo Skjermo †

Norwegian University of Science and Technology. Department of Computer And Information Science.

Abstract

This paper present a simplified approach to wind animation of natural looking tree stems and branches. The presented approach is composed from several earlier works by a number of authors, each adapted to increase its suitability for processing on a Graphic Processing Unit (GPU). The outlined approach uses two passes through the GPU. The first pass samples from a simple wind force simulator based on sine sums. It then animates the parameters and the control points defining each branch using the sampled force, taking advantage of the parallel nature of GPU's. The second pass uses a previously presented GPU-based deformer to generate and render actual models of each branch, using the animated control points.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism

1. Introduction

When rendering trees for use in real-time computer applications an approximation using billboards has often been used [JAK00] [MNP01]. However, this approach will not produce adequate results when the virtual observer is close to the viewed object. In [CCH05] and in the commercial product SpeedTree [Spe], polygon meshes are exchanged by texture billboards based on the distance to the virtual observer. Thus, interactive rates can be achieved even when using polygon meshes when close to the observer.

Historically, there has been a number of proposed methods for generating both static and animated polygon meshes for visualization of tree stems and branches. When using a tree generator such as L-systems [LP90], polygon cylinders were often used for each segment of a branch, sometimes stitched together with spheres in branching points. In [Mai02], a coarse polygon mesh model for a tree was generated by the rules of an L-systems, that was further refined using Catmul-Clark subdivision [CC78]. A similar approach was developed to be used with a predetermined centerline definition in [FFKW02], and further refined in [SE05].

In [Blo85], Bezier curves were used to define a centerline of a branch. A polygon mesh was then generated by sweeping a reference frame along this centerline, while changing the radius. Parametric tree descriptions were first introduced in [Hon71], and further developed in [WP95] where a polygon mesh representing a branch was generated by stitching together cylinder segments whose appearance was defined by parameters. In [Skj06], the tree generation methods in [WP95] was used together with a GPU-based sweep-like deformer loosely based on the Bezier curves approach of [Blo85].

To animate wind in both L-systems and parametric tree generators, constantly regenerating the whole tree model for each change in the wind force field has usually been needed. One approach to decouple the wind animation from the tree generator is to animate a tree as if each tree was a rigid body system [GCF01]. Such methods are however computationally expensive and more complex then needed for simple animations.

The presented approach simulates wind using simple sine sums. Historically, sine sums have mostly been used to simulate waves on water surfaces as shown in [IVB02]. However, as generating forces from sine sums maps easily to the parallel approach used on modern GPU's, it has lately also been used to simulate wind forces in grass animation as seen in chapter 1 of [Pha05]. The approach also makes heavy use of methods developed for General-Purpose computation on the GPU (GPGPU). In these methods, the the parallel nature of GPU computation is exploited to gain huge speedup at a low cost, as shown in [OLG*05].

† Jo.Skjermo@idi.ntnu.no

© The Eurographics Association 2007.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

2. Overall Approach

The proposed approach generates fully shaded and textured tree stems and branches with wind animation using the GPU. The approach uses two passes through the graphic pipeline, one pass to update the position of the branches control points, and one pass to generate, deform and finally shade the branches. Figure 1 shows the approach from an overall perspective.

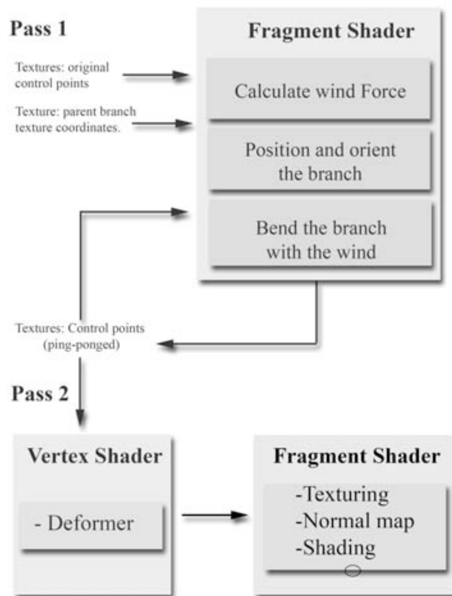

Figure 1: *The overall approach.*

2.1. Tree Description and Data Storage

The basis for the aproach is a parametric description of each branch of a tree based on the method of Weber and Penn [WP95]. The individual branches of a tree are positioned, oriented and fully described by a generated set of parameters. A tree consists of several *levels* of branches, where the main stem is *level 0*, the branches growing out of the stem is of *level 1*, and so on.

The tree generator defines a cubic Bezier curve to describe the overall shape of each branch. But it also uses a set of other parameters that describes different aspects such as radius, flare, bulges and taper. The generation of the control points for the Bezier curve defining a branch are simple translations and rotations using angles and lengths from the tree generator (as if generating a branch with only 3 segments and no radius starting at the origin with the first segment along the x-axis).

The control points of the branch before it is positioned and oriented into the tree is then stored for use in our wind animation algorithm, before finally being transformed and rotated into its initial position in the tree model. This final location and orientation is used as the starting point of the animation.

All the Bezier control points generated by the tree generator (also the control points of the unpositioned branches) are stored in textures. For a single branch, control point P0 is stored in texture 0, control point P1 is stored in texture 1, and so on for all 4 control points. The branches are sorted by their parent branch and stored in a swizzle pattern, to increase locality when doing dependent texture lookup of parent branches control points. The texture address of the parent of a branch is also stored in a texture, to enable dependent texture lookups.

3. The Deformer

The secound part of the approach is the deformer. The deformer is however presented first, as methods developed for the deformer are used in the wind animation shader. The deformer shader used here was first presented in [Skj06].

The deformer is basically a vertex program *deformer*, as described in [FK03]. The deformer basically generates branches using the parametric method of Weber and Penn [WP95], from where the branch defining equations were adapted. The deformer works on each input vertex in three steps - input (simple grid mesh) to unbent branch, unbent branch to bendt branch and finally finding the needed surface coordinate system for the present vertex. The overall approach of the deformer can be seen in figure 2.

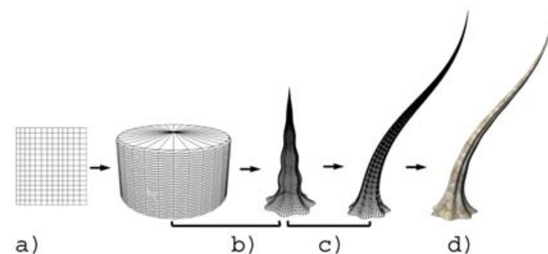

Figure 2: *a) Input vertices. b)Deform to unbent branch. c) Deform to bent branch. d) Shaded branch. From [Skj06].*

In addition to the values of the input grid mesh (υ, θ and the texture coordinates for the color texture), the deformer also requires some other input. The values for these inputs are the same for all vertices of a specific branch, and they therefore only need to be set once per branch (as setting input values is the same as setting states in the OpenGL pipeline).

© The Eurographics Association 2007.

3.1. Deformer - Input Polygon Mesh to Cylinder

The first step is to deform the vertex parametric grid given as a Vertex Buffer Object (VBO) into a cylinder as shown in equation 1.

$$\begin{aligned} x &= \upsilon \\ y &= \sin(\theta)radius \\ z &= \cos(\theta)radius \end{aligned} \quad (1)$$

Where υ defines the present position along the length of the cylinder, θ defines the present position's angular position on the cylinder, while *radius* define the radius of the cylinder. Using the υ and θ values given as input from our vertex buffer object, one can find the corresponding position on the cylinder's surface, given the cylinder's *radius*.

3.2. Deformer - Cylinder to Branch

The cylinder is further deformed by varying the *radius* as υ and θ changes, to give the cylinder the shape of an idealized unbent branch.

This part of the deformer is defined by four functions, $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$, giving the radius of an unbent branch.

The formulas used here are derived from [WP95]. For all the formulas, θ is the angular position, while υ is the position along the centerline defined to be between 0 and 1, as shown in equation 1

3.2.1. Taper, Flare, Bulges and Lobes

The f_{taper} function (equation 2) controls how fast the final radius goes toward 0. Here t is the amount of taper (how fast the branch end goes toward 0). r is the original radius of the branch. Some examples can be seen in figure 3 a), with $t = 1$, $t = 3$ and $t = 10$.

$$f_{taper}(\upsilon) = r(1 - \upsilon^t) \quad (2)$$

The f_{flare} function (equation 3) adds an increased radius near the base of a branch, and is mostly used when the branch is in fact the main stem of a tree. Here f is the $flare$ parameter, defining the amount the branch should flare out near the base. Some examples can be seen in figure 3 b), with a flare of 0, 0.1 and 0.33 (using f_{taper} with $t = 1$)

$$f_{flare}(\upsilon) = \frac{f}{100}(100^{(1-8\upsilon)}) + 1 \quad (3)$$

The f_{bulges} function (equation 4) defines a sin-curve added to the surface radius, along the direction of the branch centerline. This can be used to generate bulges based the defined frequency and amplitude. Here b_n is the number of bulges (frequency), and b_d is the depth of the bulges (amplitude). Some examples can be seen in figure 3 c), with 5 bulges of depth 0, 0.05 and 0.1 (using f_{taper} with $t = 1$).

$$f_{bulges}(\upsilon) = 1 + b_d \sin(b_n 2\pi\upsilon) \quad (4)$$

The f_{lobes} function (equation 5) basically does the same as the f_{bulges} function, but on the radius, as defined by the angle around the branch's centerline. Here l_n is the number of lobes (frequency), l_d is the depth of the lobes (amplitude), and l_t defines how fast the lobes fades away (0 for no lobes near the tip, 1 for full magnitude). Some examples can be seen in figure 3 d), with 7 lobes of depth 0 and taper 0, depth 0.2 and taper 0, depth 0.2 and taper 1 (using f_{taper} with $t = 1$).

$$f_{lobes}(\upsilon, \theta) = 1 + l_d \sin(l_n\theta)(1 - \upsilon l_t) \quad (5)$$

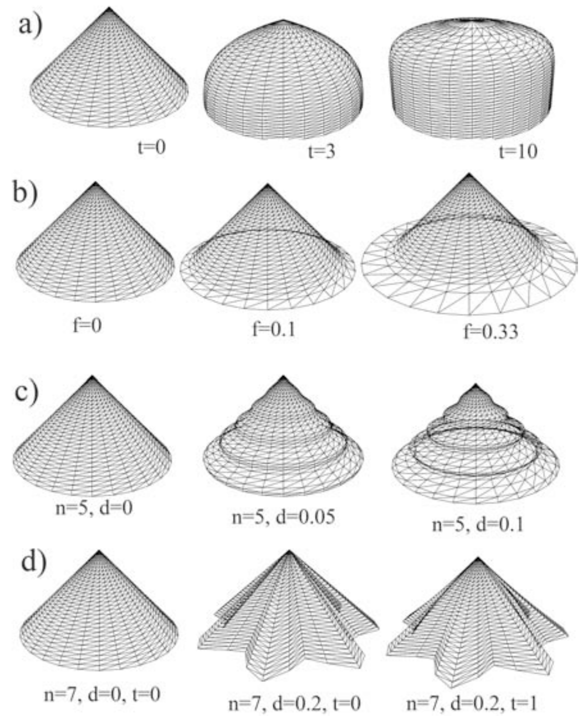

Figure 3: *a)Taper, b)Flare, c)Bulges, d)Lobes, showing the effect of different control values*

3.2.2. Deformed Cylinder

Using $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$, to define the *radius* in equation 1, the deformer that generates an unbent branch is defined as the result of three functions, one for each axis, as shown in equation 6.

© The Eurographics Association 2007.

$$\begin{aligned} f_u &= \upsilon \\ f_v &= \sin(\theta) f_{taper}(\upsilon) f_{flare}(\upsilon) f_{bulges}(\upsilon) f_{lobes}(\upsilon,\theta) \\ f_w &= \cos(\theta) f_{taper}(\upsilon) f_{flare}(\upsilon) f_{bulges}(\upsilon) f_{lobes}(\upsilon,\theta) \end{aligned} \quad (6)$$

3.3. Deformer - Branch to Bent Branch

After applying the $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$ functions, a surface of an unbent branch of length 1 is described.

The last step is to deform this surface along the cubic Bezier curve given by the Bezier control points defining a branch. This is done using the generalized de Casteljau approach to 3D free form deformation defined by Chang and Rockwood [CR94].

3.3.1. Generalized de-Casteljau Approach to Deformation

The generalized de-Casteljau approach is a function $\phi[p,q] : R^3 \rightarrow R^3$ defining an affine transformation from parametric space into affine space, as defined in equation 7.

$$\phi[p,q]\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} q_x - p_x & s_x & t_x & p_x \\ q_y - p_y & s_y & t_y & p_y \\ q_z - p_z & s_z & t_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (7)$$

Where p and q are two control points (for the first iteration), while s and t are *handles* defined for the line segment between p and q. The generalized de Casteljau approach can be seen as an iterative approach to deform space around a Bezier curve. Also, if the *handles* are unit vectors orthogonal to the line defined by P_i and P_{i-1}, and each other, for the first level of iteraton, and zero vectors for all following levels, the space will be warped with the most natural bending. This is the method used in the presented branch deformer.

3.3.2. Handle Definition and Surface point

As the control points for a cubic Bezier curve are given to the GPU vertex program, only *handles* are needed to use the generalized de Casteljau approach. Handles can be generated using Ken Sloan's approach for a moving frame on a Bezier curve as described by Jules Bloomenthal in [Gla90].

Basically, using the four control points P_0, P_1, P_2 and P_3, together with a given unit vector O in the parent branch's direction (at the branching point), Sloans approach gives us the reference frames at position P_0, P_1 and P_2 as seen in equation 8.

$$\begin{aligned} L_0 &= normalize(P_1 - P_0) \\ L_1 &= normalize(P_2 - P_1) \\ L_2 &= normalize(P_3 - P_2) \\[1ex] S_0 &= L_0 \chi O, T_0 = S_0 \chi L_0 \\ T_1 &= S_0 \chi L_1, S_1 = L_1 \chi T_1 \\ T_2 &= S_1 \chi L_2, S_2 = L_2 \chi T_2 \end{aligned} \quad (8)$$

Applying the generalized de Casteljau deformation using the f_u, f_v and f_w from equation 6 as u,v and w in equation 7, gives us the final deformed branch that follows the given Bezier curve. Calulating a point on the surface for a given υ and θ can be done using equations 9.

$$\begin{aligned} R^0_0 &= lerp(P_0, P_1, u) + S_0 v + T_0 w \\ R^0_1 &= lerp(P_1, P_2, u) + S_1 v + T_1 w \\ R^0_2 &= lerp(P_2, P_3, u) + S_2 v + T_2 w \\[1ex] R^1_0 &= lerp(R^0_0, R^0_1, u) \\ R^1_1 &= lerp(R^0_1, R^0_2, u) \\[1ex] R^2_0 &= lerp(R^1_0, R^1_1, u) \end{aligned} \quad (9)$$

R^2_0 gives the final world position for the present vertex defined by a u and θ value.

3.4. Shading and Texturing

To add lighting and textures to the deformed branch, the tangent and binormal (and a normal) at each surface point must be found. This also enables us to use texture normal mapping to add even further detail to a branch surface. A method to generate a tangent and binormal (for normal generation) on the parametricly defined surface, suitable for use in GPU programs, was presented in [Fer04].

Given $f(x,y,z) = (f_x, f_y, f_z)$ the Jacobian matrix is defined as shown in equation 10

$$J(x,y,z) = \begin{bmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} & \frac{\partial f_x}{\partial z} \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} & \frac{\partial f_y}{\partial z} \\ \frac{\partial f_z}{\partial x} & \frac{\partial f_z}{\partial y} & \frac{\partial f_z}{\partial z} \end{bmatrix} \quad (10)$$

If the unit tangent and binormal vectors T and B are given on the surface before deformation, multiplying these vectors with the Jacobian will give the deformed tangent and binormal. Also, one can calculate a deformed unit normal N by using:

$$n` = normalize[(J(x,y,z) t \chi (J(x,y,z)) b)]$$

Using this approach, one first takes the partial derivatives

© The Eurographics Association 2007.

with respect to u and θ of the equations for each axis in equation 6. This gives a tangent and a (estimated) binormal on the surface of the cylinder deformed to an unbent branch. Multiplying the calulated tangent and binormal with the Jacobian of the de-Casteljau deformation, will give the tangent and binormal on the final deformed surface, and can then be used to generate a TBN matrix to be used for further fragment computations. How to calulate the Jacobian for the deformer is shown in [Skj06].

4. Sums of Sine Based Wind Animation

In this section the approach for using sums of sine to animate the branches of our tree models is presented. A set of parameters defines (at most four) sine waves that simulates a wind environment. One can sample the sine-waves using the position of the tip of each branch as basis for calculating the results of the sine-wave equations. The values found are summed, and used as a force to bend a branch from its original shape by moving the control points of a branch.

The control points being moved by the wind force are the original control points of a branch, before the branch was positioned and oriented into its poisition in a tree. Usually, one would calculate the position to attach a branch into the tree by first calculating for level 0, then for level 1 and so on. However, for each branch being positioned, the method proposed simply position and orient the branch onto its parent branch from the last animation step. With this simplification, all the updating of the branches control points can be calculated in parallel on the GPU on a per branch basis.

From a implementation based point of view, the whole animation process, including the sine-sum based wind force sampling, is implemented as a single fragment shader. The fragment shader needs the control points of the present branch, the parameters to attach the branch to its parent, the sine-wave control parameters, and finally the control points of the parent branch as input values. The output is the four new control points of a branch (and a orientation vector O).

The fragment program uses the OpenGL extension for Multiple Render Targets (MRT) to write four textures in the same execution. This makes it possible to update all four control points of a branch in a single fragment program. The results are written to 4 textures in a ping-pong fashion, meaning we are switching between two groups of four textures for reading and writing (as the output from last animation step is used as input to the present step).

4.1. The Sum of Sine Force Approximation

The animation uses forces generated by using sums of sine. As in chapter 1 of [Fer04], the state of a single sine-wave i is a function of the position (x,y), and time (t), as shown in equation 11.

$$W_i(x,y,t) = A_i \sin(D_i \cdot (x,y) w_i + t\varphi_i), \qquad (11)$$

where A_i is the amplitude, D_i is the waves direction on the horizontal plane, w_i is the frequency (that relates to wavelength as $w = \frac{2\pi}{wavelength}$, and φ_i is the phase constand describing the waves speed as $\varphi = speed \times \frac{2\pi}{wavelength}$.

When simulating waves on water, one usually sums the set of sine-waves for a given position to generate a height of the position one is interested in, as shown in the sum of sine equation (equation 12).

$$H(x,y,t) = \sum(A_i \sin(D_i \cdot (x,y) w_i + t\varphi_i)). \qquad (12)$$

However, wind is a directional force and height is not needed. The force can be obtained by multiplying the state $W_i(x,y,t)$ of a wave i with its given (normalized) direction D_i, and then sum the resulting vectors using equation 13.

$$F_{wind}(x,y,t) = \sum(D_i(A_i \sin(D_i \cdot (x,y) w_i + t\varphi_i))). \qquad (13)$$

The actual implementation does however use an improved version of the force-generating sine-sum equation, that gives more control over the sharpness of the peaks and width of the troughs of each wave, as seen in equation 14.

$$F_{wind}(x,y,t) = \sum(D_i(\frac{A_i}{l}\left(\frac{\sin(D_i \cdot (x,y) w_i + t\varphi_i)}{2}\right)^k)), \qquad (14)$$

where l is the length between two control points, and the power constant k adds extra control over the wave as described in chapter one of [Fer04]. As the exponent k is raised above 1, the peaks of a sine sharpens, and the valleys flatten.

Using the sums of sine as presented means that we are sampling from a (horisontal) 2-dimentional wind field. The generated force is also in this plane. The shape of the landscape or any objects in the scene does not influence the wind in any way.

4.2. Positioning and Orientating the Branch

The first step is to get the original control points of a branch without any wind influence and before it has been positioned and oriented into the tree. These values are stored in texture memory, and does not change during any step of the animation process.

The original control points for the branch without any wind influence must be positioned and oriented onto its parent branch. This position and orientation is described by a set of parameters from the tree generator (and the parameters describing the parent branch itself). It is important to

© The Eurographics Association 2007.

note that one uses the control points of the parent branch from last animation step, to calculate the position and orientation of a branch for the next animation step:

P0p, P1p, P2p, P3p are the control points of the parent branch, and O is the given parents orientation vector of the parent branch (used to generate handles for the parent). υ is the parametric position for P0, on the parent branch, θ is the angle around the centerline and α is the angle to rotate "down". P0, P1, P2 and P3 are then the original unbent Bezier control points of the present branch in its final position and orientation.

To find the attachment position of a branch (basically, where P0 should be positioned), we simply find the position on the Bezier curve defined by its parent branch control points, using the position parameter υ along the curve.

The rotation itself is done by first finding the derivative of the parent branch Bezier curve, at the present branches attachment point υ, giving a vector U along the parent's direction. By using θ with the generalized de Casteljau approach, as described in 3.3.2, a point on the parent branch sweep surface in the horisontal direction the branch should grow is found. The point found is used together with position $P0$ of the branch to define a vector V.

Using cross products of the (normalized) vectors U and V, enables us to find 3 perpendicular vectors. These vectors defines a coordinate system on the parent branch, that is used when attaching the branch to it. The unbent original branch is attached into the coordinate system, then rotated down by the angle α , as seen in figure 4.

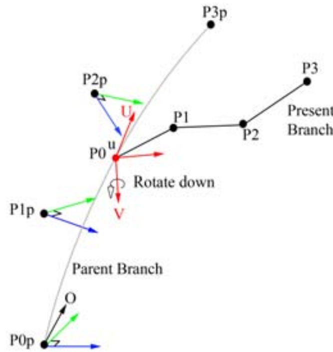

Figure 4: *Position and orientate a branch onto its parent branch*

4.3. Bending the Branch

When the original control points defining the overall shape of a branch are positioned and oriented to theire positions in the tree, we can move individual control points to bend the branch.

As a branch's bending is defined by the four control points of a Bezier curve, we can bend it using the approach developed for bending branches in the movie Shrek [Pet]. The approach first calculates a *bend factor* for each control point of a branch, and then move each control point from its original position. A simple version of a *bend factor* is calculated for each control point as shown in equation 15.

$$b_i = b\left(\frac{i}{n-1}\right). \tag{15}$$

Where b is a vector representing the *bend factor* for the whole branch, i is the present control point, and n is the total number of control points of a branch.

The force calculated from the sine waves is used as the *bend factor b*. Also, to ensure that a branch looks like growing out of it's parent branch, the start of a branch should be static compared to the parent (at control point P0), making that only control points P2 and P3 are actually updated.

This means that control points P2 and P3 of the original unbent branch, after it has been positioned and oriented into it's position, can be updated to simulate bending from the simulated wind. From visual inspection, equation 15 has been tuned as shown in equation 16, to produce the *bend factors* for control points P2 and P3.

$$\begin{aligned} b_3 &= F^{1.8} \\ b_2 &= F\left(\frac{1}{3}\right)^{1.8} \end{aligned} \tag{16}$$

Control points P2 and P3 can now be bent using the calculated *bend factors* as shown in equation 17, where b_i is the *bend factor* of control point P_i. Figure 5 shows the calculation for one control point.

$$P_i' = P_{i-1}' + \frac{(P_i - P_{i-1}) + b_i}{\mid P_i - P_{i-1} \mid}. \tag{17}$$

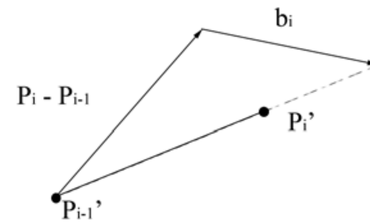

Figure 5: *a) Position and orientate a branch onto it's parent branch ,b) Updating the position of a control point, figure recreated from [Pet]*

© The Eurographics Association 2007.

The resulting control points after applying the animation are written to four textures using MRT. These textures are then used as input to the Deformer GPU program (but also used as input to the next animation step).

5. Results

Some examples showing the branch deformer using normal textures and lighting can be seen in figure 6, while figure 7 shows a snapshot from a test application animating a scene with over a 100 highly detailed trees with wind animation.

Figure 6: *From left: shaded branches with texturing and lightning, shaded branch with different parameters, the stem of a cacti, the start of a stem. From [Skj06]*

Figure 7: *Screenshot from test application, animating wind in forrest scene*

Some performance results of drawing trees using both the deformer and wind animation GPU shaders can be seen in figure 8, for different number of branches (and vertices).

For the wind animation shader step, the size of the textures used, fully determines the update speed. For a Nvidia

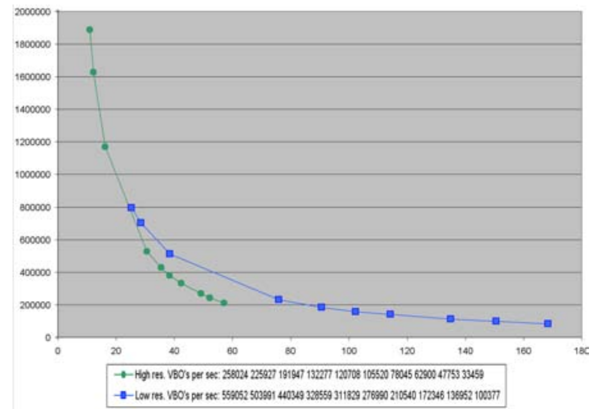

Figure 8: *Frames per secound for different number of VBO branchs per secound*

7900GTX GPU, using 256x256 textures the wind animation shader calculates at 600 iterations per secound, applying wind animation to more then 65k branches in parallel (when not drawing to the screen with the deformer shader pass). When using 512x512 textures, the wind animation calculates at 180 iterations per secound.

When calculating the position and orientation of a branch, the original control points of a branch (before they where positioned into the tree) are used for the present branch. For the parent branch (the branch one attaches the present branch onto), one uses the control points from the last animation step. This means that this method for positioning and orienting a branch introduces an error, as the animation actually lags behind one animation step for each structural level we move out in the tree structure.

As the branches usually gets smaller the further out in the tree structure one goes, the error one gets using the parent branch from the last animation step, could easily become unacceptable. However, when the generated wind forces are kept relatively small, the movement between each animation step is not large enough for a observer to actually notice this error, especially when keeping the branch oscillation speed low.

Defining the parameters for a sine wave that gives a good wind effect in the animation can be hard, especially as the parameters influence each other. One should also note that changing a parameters defining a sine wave during runtime will introduce an abrupt change in the wind animation. These problems can be handled by defining a set of parameters that yields good wind animation effects, and blend the impact of a wave out before blending in a new sett of wave parameters.

The wind animation can handle a lot more branches then the deformer part. This stems mostly from the fact that there is more surface vertice positions beeing calulated

© The Eurographics Association 2007.

then branches animated. Also, texture fetches in the vertex shaders are not optimized on earlier graphic cards, and there are more fragment shader computation units then vertex fragment computation units. On the newest graphic cards, like the Nvidia 8800 series, there is however no real difference between fragment and vertex computation units, so the load between the vertex deformer and the fragment wind animation shades should in theory be a lot more balanced. This will be tested in the near future.

References

[Blo85] BLOOMENTHAL J.: Modeling the mighty maple. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 305–311.

[CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 6 (1978), 350–355.

[CCH05] CANDUSSI A., CANDUSS N., , HÖLLERER T.: Rendering realistic trees and forests in real time. *Eurographics journal, Computer Graphics Forum 24* (2005), 73–76. EG Short Presentations.

[CR94] CHANG Y. K., ROCKWOOD A.: A generalized de casteljau approach to 3d free-form deformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994* (1994), SIGGRPAH, ACM Press, pp. 257–260.

[Fer04] FERNANDO R. (Ed.): *GPU Gems*. Addison-Wesley Profesional, 2004.

[FFKW02] FELKEL P., FUHRMANN A., KANITSAR A., WEGENKITTL R.: Surface reconstruction of the branching vessels for augmented reality aided surgery. *Analysis of Biomedical Signals and Images 16* (2002), 252–254. (Proc. BIOSIGNAL 2002).

[FK03] FERNANDO R., KILGARD M. J. (Eds.): *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley Professional, 2003, pp. 218–226.

[GCF01] GIACOMO T. D., CAPO S., FAURE F.: An interactive forest. In *Eurographics Workshop on Computer Animation and Simulation (EGCAS)* (sept. 2001), Cani M.-P., Magnenat-Thalmann N., Thalmann D., (Eds.), Springer, pp. 65–74. Manchester.

[Gla90] GLASSNER A. (Ed.): *Calculation of Reference Frames along a Space Curve*. Academic Press, 1990, pp. 567–574. By Jules Bloomenthal.

[Hon71] HONDA H.: Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology* (1971), 331–338.

[IVB02] ISIDORO J., VLACHOS A., BRENNAN C.: Rendering ocean water. In *ShaderX* (2002), Wordware Publishing, pp. 347–Ű356.

[JAK00] JAKULIN A.: Interactive vegetation rendering with slicing and blending. In *In Proceedings of Eurographics 2000* (2000). Short Presentations.

[JPM00] JIRASEK C., PRUSINKIEWICZ P., MOULIA B.: Integrating biomechanics into developmental plant models expressed using l-systems. *Plant biomechanics* (2000), 615–624.

[LP90] LINDENMAYER A., PRUSINKIEWICZ P.: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[Mai02] MAIERHOFER S.: *Rule-Based Mesh Growing and Generalized Subdivision Meshes*. PhD thesis, Technische Universitaet Wien, Technisch-Naturwissenschaftliche Fakultaet, Institut fuer Computergraphik, 2002.

[MNP01] MEYER A., NEYRET F., POULIN P.: Interactive rendering of trees with shading and shadows. In *In Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (2001), Springer-Verlag, pp. 183–Ű190.

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (Aug. 2005), pp. 21–51.

[Pet] PETERSON S.: Visual effects in shrek. Silicon Valley ACM SIGGRAPH, http://silicon-valley.siggraph.org/MeetingNotes/Shrek.html.

[Pha05] PHARR M. (Ed.): *GPU Gems 2*. Addison-Wesley Profesional, 2005.

[SE05] SKJERMO J., EIDHEIM O. C.: Polygon mesh generation of branching structures. In *SCIA, Image Analysis, 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005, Proceedings* (2005), Kälviäinen H., Parkkinen J., Kaarna A., (Eds.), vol. 3540 of *Lecture Notes in Computer Science*, Springer.

[Skj06] SKJERMO J.: A gpu-based branch deformer. In *Proceedings of the 22nd spring conference on Computer graphics* (2006), Comenius University, Comenius University, Bratislava, pp. 120–127.

[Spe] Tree modeling and rendering middleware. Commersial product, http://www.speedtree.com.

[WP95] WEBER J., PENN J.: Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), ACM Press, pp. 119–128.

© The Eurographics Association 2007.