EG UK Theory and Practice of Computer Graphics (2008)
Ik Soo Lim, Wen Tang (Editors)

Generation and Tessellation of Tree Stems

Jo Skjermo [†]

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

Abstract

When visualizing tree stems and branches for use in interactive applications, the polygon models resolution are usually as low as possible to achieve a high frame rate. Also, to ease animation and mesh generation, each branch of a tree model is often considered as a distinct mesh. However, by using a single watertight mesh for a tree, together with (GPU-based) tessellation, both the resolution and appearance of a tree can be greatly improved while maintaining a high frame rate. This paper presents concepts, ideas and early work on generating watertight polygon meshes of animated trees stems suitable for refinement and tessellation of such meshes.

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

As of today, a complete tree is often rendered by using distinct branches, where a child branch is drawn as starting inside its parent branch. However, this approach gives a visual error where the polygons of a child branch penetrate the polygons of the parent branch. This is especially noticeable when animating a tree. To reduce this problem, one can generate a single mesh for the whole tree (only the main branches of a tree is considered, as texture fronds can easily be used to flush out a tree with details). The generated mesh can then be tessellated for improved appearance.

1.1. Overall Approach

The work presented here considers two approaches, depending on the method used for animating the trees. In the first approach a structure of rigid bodies and joints is constructed. Then a low resolution mesh is generated at each time step, which is used for tessellation.

In the other approach, a low resolution polygon mesh is generated directly from the trees description. The vertices of this mesh are considered to be mass points, while the edges are considered to be length constraints. After each time step the mesh is tessellated. For all physic simulations, the AGEIA PhysX physics engine [Phy] is used.

2. Tree Generation

For the generation of the polygon mesh of a tree model, a tree is considered to be a set of connected segments which has a start and end point, with a given length and radius. In addition, at an end-point of a segment with more than a single attach segment, the attached segment with the largest radius is considered to be the direct continuation of the segment with the end-point (the branching points). In other words, the child segment with the largest radius is considered to be continuation of the present branch, while any other segments are considered new branches.

2.1. Branching Rules

It is assumed that the radii, and the angles of branching at the branching points follow the rules for branching of vascular transportation systems as set forth by DaVinci, and refined by Murray (see [SE05]). The first rule states that the area just before branching is equal to the sum of the areas after a branching. The second rule describe the angles that the child branches are rotated away from the parent. If the branches follow these rules, the mesh generation approach will mostly generate a well-behaved low polygon mesh, al-

[†] Jo.Skjermo@idi.ntnu.no

though no throughout verification has been done to the validity of this assumption yet.

2.2. Rigid Body Tree

In the case of a tree consisting of a set of rigid bodies, a branch consists of a set of capsules (a swept sphere) attached to each other in succession. At branching points, the parent capsule is shortened by an amount equal its radius, and a capsule of height zero is added (a sphere). This approach can be seen to the left in Figure 1.

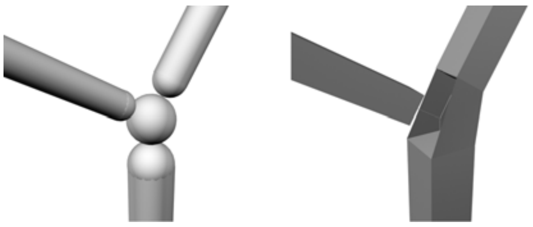
Figure 1: *Left: Capsules, Right: partly generated mesh*

Joints with springs can be used to connect segments at any point. However, note that if one does not add a joint between two capsules, these are considered to be part of the same rigid body. In this context, the rigid body generation can be seen as similar of the approach presented in [Zio08]. If several capsules are used as part of the same rigid body, one has to take special care when handling destruction; by finding what segments to remove from the body. These segments can then be added to a new rigid body for further animation. This approach enables full collision detection and destructible trees, but is computational costly, so it should be considered only when required.

2.3. Mesh Generation

The generation of the low polygon mesh is in concept similar to the approach presented in [SE05] (recommended for clarification). For a capsule/segment four quads is generated, by using the direction, the start points, the end point and the radii (basically, a matrix skinning). A mesh of a branch is generated by following the capsules for that branch, starting at its root.

However, if a capsule has more then one attached (child) capsule, the first child capsule is added by generating a new slice at a length equal the segments radius. For the spherical capsule, only the sides that have no additional child segments in its direction are closed, as seen in Figure 1. For any child capsules, a start slice for the segment is found by translating along its direction by a distance equal its radius. This slice and the open part of the spherical capsule can then be closed with quads.

Special care has to be taken when generating normal vectors, as the normal vectors are used to generate control points for tessellation. Generating normal vectors using all the quads sharing a vertex will result in a deformation of the parent branch around branching points. However, this is easily fixed by ignoring any quads not part of the present branch when generating the normal vectors for a branch, and assuming that all directions are closed off at branching points. That is, only the vertices considered to be part of the same branch are used when generating the normal vectors.

2.4. Tree Mesh as Cloth

In this approach a low resolution polygon mesh of a tree is generated directly from the tree description. The vertices are considered mass points, while the edges are considered to be length constraints. One approach for animating such structures could be to use Verlet integration as described in [Jak03], or even animate on the GPU as described in [Nvia]. However, as the PhysX physics engine is used, the tree meshes are handled as cloths, as described in [MHHR06].

Using this approach one can also in theory have destructable trees, as the physics engine supports tearing of cloth along defined seams. However, only the vertices are considered during collision detection so it is best suited for effect physics like wind animation.

3. Tessellation

To get a better quality when rendering, the low polygon mesh is tessellated on the GPU. Tessellation also gives natural scalability (level of detail.) The original mesh generation approach made meshes targeted for Catmul-Clark subdivision. Catmul-Clark subdivision can be done on the GPU, for instance as seen in [ASK08], However, GPU-based subdivision approaches usually requires several passes, one for each level of subdivision. Therefore, in this work a local tessellation approach is considered.

3.1. PN-Triangles and PN-Quads

The approach used for the mesh generation produces a mesh consisting of quads, however, the PhysX cloth animation step works on a triangulated mesh. For tessellation one can use triangles or quads, depending on the approach selected.

For triangles, curved PN-Triangles as described in [VPBM01] can be used. For a PN-Triangle, the original vertices are used directly as control points, as seen in Figure 2 a). Six new edge control points are also generated using the vertices and normal vectors for that edge, as seen in 2 b). To calculate an edge control point, the points and normal vectors of that edge are used. For instance, in Equation 1, the calculation of b_{210} is shown. Finally, an internal control point is generated as the weighted average of the edge and vertices control points.

$$b_{210} = (2P_1 + P_2 - w_{12}N_1)/3, where: w_{ij} = (P_j - P_i) \cdot N_i \quad (1)$$

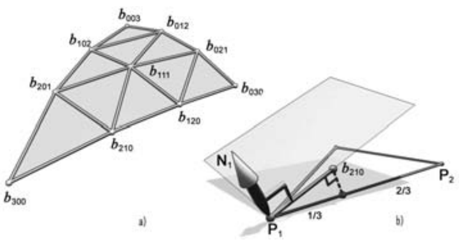

Figure 2: *From [VPBM01], a) PN-Triangle, b) Edge control point*

For quads, PN-Quads as described in [Lan03] can be used. For PN-Quads the given vertices are used directly as control points. The same approach as for PN-Triangles is used to generate new edge control points. However, eight new edge control point is generated instead of six. Four internal control points are generated by addition of line segments, found by using a corner point and the two closest new edge control points.

When generating edge control points, $1/3$ of the length between P_i and P_j is normally used to produce a control point, as seen in Figure 3. However, by using $1/2$, the generated surface will be more rounded, as seen in figure 3. An example of a tessellated PN-Quad mesh with different tessellation levels can be seen in Figure 4.

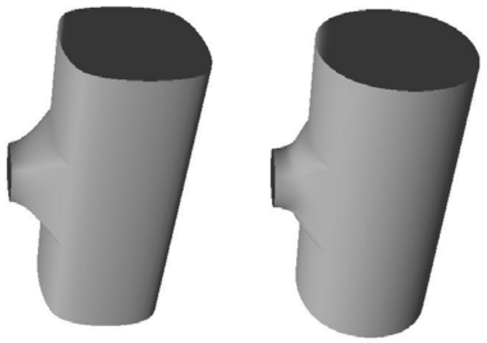

Figure 3: *Left: normal (PN-Quad) approach, Right: Rounded (PN-Quad) approach*

To do the tessellation of the PN-Quads on the GPU, a (OpenGL) vertex shader is used. First, control points (the vertices of the tree mesh) and indices are sent to the GPU as bindable uniform buffers. For all quads (or triangles) with the same tessellation level, a tessellation pattern of the required resolution is drawn once for each polygon using instancing, giving u and v parameter values to the vertex shader. In the vertex shader program, the control points are read from the buffers using the instance id of the drawn quad, into the instance buffer. The control points are then used to calculate a point on the surface for the given u and v value, using PN-Triangles or PN-Quads depending on the input polygons type.

For PN-Quads, the vertex shader first calculates the control points. Then it uses the approach of the tessellation shader in Nvidia's GPU SDK version 10 [Nvib], to find the position on the Bezier surface for the given u and v position in the tessellation pattern. For PN-Triangles, a vertex shader as proposed in [BS05] is used. Hopefully, both of these approaches will easily map to future tessellation capable GPU's, as described in [Tat08].

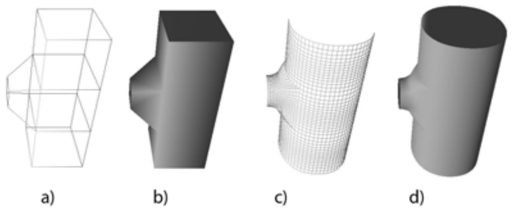

Figure 4: *a) Low Resolution Mesh, b) Same as a); but shaded, c) Low resolution mesh tessellated by 20 levels, d) Same as c), but shaded*

4. Results, Present focus and Future Work

The overall approach presented here has only been partially tested and verified. Future integration and testing is needed to fully verify the approach. However, some preliminary results can be seen in Figure 5 and Figure 6.

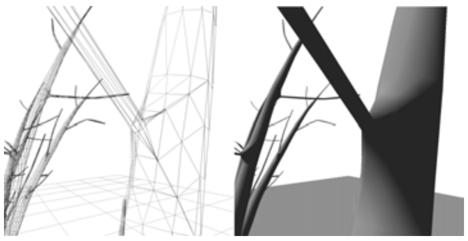

Figure 5: *Left: Trees (PN-Quad) tessellated 5 levels, Right: Shaded*

4.1. Present focus

As seen in figure 3, using PN-Quads (and PN-Triangles) for the tessellation of a mesh that is as coarse as the ones generated, do introduce some degeneration, as a result of limitations in PN-Triangles and PN-Quads approach. A possible

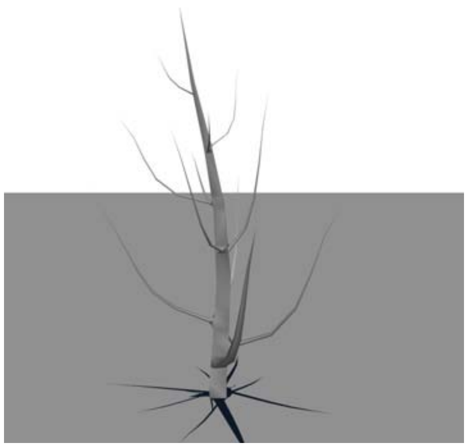

Figure 6: *A low resolution mesh animated as cloth with stiff length constraints*

solution is to do a single level of subdivision on the generated mesh before tessellation, instead of handling the normals per branch separately as descibed in section 2.3. For quads, Catmull-Clark subdivision can be used. For triangles, Loop subdivision can be used. One level of subdivision can be done on the CPU, or on the GPU as a separate pass before PN-Quad/PN-Triangle tessellation.

For Catmull-Clark subdivision, the geometry shader proposed in [ASK08] could be used. For triangle meshes, a similar approach for a one level Loop subdivision on the GPU, has been implemented for further testing. For the Loop geometry shader, the triangles are drawn using the *Triangles with Adjacency* primitive, with index and valence as per vertex data. The vertex positions and normals for the tree mesh together with indices into the vertices buffer for any additional edge endpoint (that is not already part of the adjacency data) is attached as bindable uniform buffers. A single level of Loop subdivision can then be done and output into the GPU memory for use as input to the tessellation step (using Nvidia's *NV_transform_feedback* extension to OpenGL). Present focus is on problems with this extended approach (such as limited valence at vertices, and limited size of bindable uniform buffers on present hardware).

4.2. Future work

When tessellating a mesh, height-mapping or procedural displacement could be used to add additional detail. This could be used to add more details to a tree, than details typically added by texture methods like parallax mapping. For instance large ridged, bulges and some types of fungi could be considered. Texturing has only loosely been looked into. Cylindrical texture mapping along the length of branches, with texture layer blending at branching points (calculating continuous texture space for each branch) looks promising, however some work still remains.

Adaptive tessellation is when one uses different tessellation level for each edge of the given polygon (quad or triangle). This gives the opportunity to tessellate differently by for instance curvature, area, distance or a mixture of these. A closer look at approaches for (GPU-based) adaptive tessellation could be considered.

References

[ASK08] ANTAL G., SZIRMAY-KALOS L.: Fast evaluation of subdivision surfaces on direct3d 10 graphics hardware. In *ShaderX6* (2008), Engel W., (Ed.).

[BS05] BOUBEKEUR T., SCHLICK C.: Generic mesh refinement on gpu. In *ACM SIGGRAPH/Eurographics Graphics Hardware* (2005). http://www.labri.fr/publications/is/2005/BS05.

[Jak03] JAKOBSEN T.: Advanced character physics, 2003. Online Article., http://www.gamasutra.com.

[Lan03] LANDER J. (Ed.): *Graphics Programming Methods*. Charles River Media, 2003.

[MHHR06] MÜLLER M., HEIDELBERGER B., HENNIX M., RATCLIFF J.: Position based dynamics. In *Proceedings of Virtual Reality Interactions and Physical Simulations (VRIPhys)* (2006), pp. 71–80.

[Nvia] NVIDIA: *Cloth Simulation*. Direct3D SDG samples, WP-03018-001 v01., http://developer.nvidia.com/object/sdk_home.html.

[Nvib] NVIDIA: *Tessellation*. OpenGL SDK Samples, v. 10., http://developer.nvidia.com/object/sdk_home.html.

[Phy] Ageia physx engine sdk. Physic Engine and Software Developer Kit. http://www.ageia.com/developers/.

[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.

[Tat08] TATARINOV A.: Instanced tessellation in directx10, feb 2008. At Game Developer Conference 2008., http://developer.download.nvidia.com/ presentations/2008/GDC/Inst_Tess_Compatible.pdf.

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved pn triangles. In *SI3D* (2001), pp. 159–166.

[Zio08] ZIOMA R.: Gpu-generated procedural wind animation for trees. In *GPU Gems 3* (2008), Nguyen H., (Ed.), Addison-Wesley, pp. 105–123.