TECHNICAL UNIVERSITY OF DENMARK

Conversion of Graphs to Polygonal Meshes

M.Sc. Thesis

Jacob Nielsen
IMM-M.Sc.-2009-64
10/1/2009

Conversion of Graphs to Polygonal Meshes

Jacob Nielsen

Technical University of Denmark

Abstract

I present a real-time rendering method for converting directed acyclic graphs to polygonal meshes directly on the GPU. The method is composed of two steps. First a graph, where each node has information about its direction, radius and length, is used to construct a non-intersecting, provided well-defined input, topologically correct, closed polygon mesh. This produces a coarse mesh in real-time and the second step of the method then improves the visual quality of the mesh, by automatically tessellating it. This smoothes the mesh and provides automatic level of detail. Both steps are carried out on the GPU in the geometry shader stage, which carries certain restrictions on the techniques available and the quality of the mesh that can be produced.
This method can be used to take a graph of a branching structure, such as plants, trees, character and creature skeletons, and produce, in real-time, a 3D polygonal mesh.

Keywords: polygonal mesh, directed acyclic graph, GPU, real-time, geometry shader

1 Introduction

In recent years many advances have been made in the field of computer graphics. First vertex- and pixel shaders were included in the graphics pipeline; the former making it possible to perform operations on each vertex such as transformations, skinning and lighting, and the latter making per-pixel operations possible, allowing for much more realistic lighting and advanced texturing effects among other things.

Fast forward a few years to the introduction of the geometry shader stage to the graphics pipeline. The geometry shader processes entire primitives, in the form of points (one vertex), lines (two vertices) or triangles (three vertices). Additionally, the primitives can include edge-adjacent vertex data, limited to two vertices for a line and three vertices for a triangle. The geometry shader stage makes it possible not only to entirely discard the primitive but also to emit one or more new primitives. This opens up for a lot of new possibilities that were previously not possible to do on the GPU.

With the increasing complexity of current real-time applications, and especially of games, the possibility of freeing up CPU resources by delegating work to the GPU is vitally important. The goal of this paper is to examine some of the techniques which would normally have taken place offline that might now be carried out on the GPU in the geometry shader stage, namely polygon mesh creation and refinement.

Converting a graph to a polygonal mesh is attractive for a number of reasons. Many organisms and objects can be described by a graph. Plants and trees are often described by Lindenmayer Systems (1) (or simply L-systems), where each variable in the L-system is bound by a set of rules. This can essentially be thought of as a graph where each variable in the L-System translates to a node in the graph and determines the properties of that node. The skeleton structure of humans and animals can be described as a graph where each joint (node) in the skeleton (graph) has a connected bone (edge)

and information about the joints it is connected to. In the medical imaging industry you often produce 2D slices of the body and its organs, which are then used to diagnose the patient. The information from these 2D slices could be used to produce a graph of, for example, the cardiovascular system and visualize it in 3D, which would help in both diagnosing and planning the procedures of a surgical operation. Interactive 3D applications such as games are another obvious area in which the method might be applied. In the game development industry the time from conception of an idea to the release of the final product is often very long, and one of the reasons for this is that the games are getting increasingly complex and require so many models in form of objects, creatures et cetera which have to be modelled by hand. If some of these models could be created automatically simply from the description of a graph, perhaps with some random element, a lot of work could be offloaded from artists and modellers. Another huge benefit of creating models based on a graph is that we get the ability to modify or add parts to the model, as it is simply a matter of adding, modifying or removing nodes from the graph. Therefore, being able to take an arbitrary graph - albeit the graph has to adhere to certain definitions - and convert it to a polygon mesh is obviously compelling. An example of this could be plants that are growing, trees sprouting new branches or characters (or creatures) loosing limbs or growing bigger limbs or even new limbs.

Another technique that will be explored in this paper is how the visual quality of a polygonal mesh can be automatically improved by tessellating the mesh. In some cases it might not be necessary for artists and modellers to create a very detailed mesh, if a comparable visual quality can be achieved by automatically refining the mesh. This will not only cut down on the time needed for development, it will also induce less overhead on the CPU-GPU communication as a coarse mesh can be sent to the GPU which then produces a more detailed mesh with natural adjustment for level of detail.

In this paper I explore how these techniques might be implemented in the geometry shader stage and look at the limitations and challenges we are faced with. The main contribution of this article is:

- A solution for generating a closed non-self-intersecting polygonal mesh with support for branching directly on the GPU.
- A general tessellation technique which will improve the visual quality of almost any given mesh implemented entirely on the GPU.

2 Related Work

There are many ways to represent a 3D model and consequently many methods to consider when converting a graph to a mesh. Several parameterized geometry methods have been proposed to construct a surface mesh, e.g. Bezier surfaces (2) , NURBS (3), and subdivision surfaces. Subdivision surfaces were first introduced by Catmull and Clark (4) and Doo and Sabin (5) as an answer to the problem that B-spline surfaces are not able to handle arbitrary topology. Doo-Sabin adapted Chaikin's (6) polygon refinement technique for the bi-quadratic uniform B-spline and introduced a new surface generation method. The scheme developed by Catmull-Clark was designed to generalize the uniform B-spline knot insertion to meshes of arbitrary topology. Apart from Doo-Sabin surfaces and Catmull-Clark surfaces, many other subdivision methods have been proposed. Some of the more notable ones are Loop subdivision (7) which applies to arbitrary triangle meshes, Butterfly subdivision (8) which also applies to triangle meshes. Butterfly subdivision would originally introduce artefacts in the mesh, but the method was revised by Zorin (9) who overcame this problem and also

improved the smoothing factor. Finally, the Kobbelt (or √3) subdivision scheme (10) which applies to quad meshes is also worth mentioning. Some practical approaches of using subdivision surfaces have been proposed by (11) for creating meshes with support for branching and (12) whom presents a method allowing different subdivision rules at each level of the subdivision and a method for growing the mesh based on rules. Today's GPUs are optimized, and indeed designed, to work with triangles, so even though there are many advantages of parameterized geometry (e.g. subdivision surfaces) over triangle meshes, such as less memory storage needed, lower bus bandwidth needed and fewer calculations needed for example for animation and collision detection, working with parameterized geometry would presently have to be dealt with in software and thus is not a good fit in real-time rendering scenarios.

Another approach is using implicit surfaces which simply put are isosurfaces (or contours) through a scalar field in 3D. Implicit surfaces are often used in the medical imaging industry for visualizing volume data and are well suited for visualizing organic forms and fluids. One popular technique based on implicit surfaces is Metaballs (13), which "melt" spheres and is very well suited for visualizing organic objects. Metaballs as a technique has also found its uses in real-time applications such as the game Spore (14). Much work has been done in the field of polygonization of implicit surfaces, especially by Bloomenthal (15) on tree modelling, and (16) which allows for non-manifold implicit surfaces. One of the more popular, if not the most popular, technique for extracting a polygonal mesh from an isosurface is the marching cubes algorithm (17). While using implicit surfaces to create meshes from a graph is possible, it is not an ideal solution since the accuracy varies with the resolution of the grid used to subdivide the volume data. In low resolutions parts of the mesh produced might even be missing.

Considering the current hardware and the restrictions that working with graphs impose, it is clear that some form of polygonal visualization must be used. L-systems, as introduced by the biologist Aristid Lindenmayer (1), is a string rewriting system useful to describe biological processes of plants, other organisms and branching structures. A lot of work has been done in the visualization of L-systems. Previously mentioned (12) proposes a subdivision surface technique for visualization of parametric L-Systems, building on the work of Prusinkiewicz et al who in (18) proposed the use of parameterized geometry for L-systems visualization. The introduction of PL-systems by Hannan (19) was followed up by (20) on visualizing plants interacting with their environment. Although much work has been done in visualizing L-systems and it at first hand seems as a great fit for visualizing branching structures, they consider rule based mesh growing and not working in finite data sets like a graph. Secondly, many L-system implementations do not consider creating a topologically correct mesh that does not intersect at branching points.

The method most relevant for this paper is the work done by (21) and (22). P. Felkel et al (21) propose an algorithm they call SMART (Surface Models from by-Axis-and-Radius-defined-Tubes) for generation of a topologically correct 2-manifold mesh of liver vascular structures. The method is able to handle n-furcations and constructs a coarse mesh which can then be smoothed using Catmull-Clark subdivision (4). However, the mesh created by the SMART algorithm may self-intersect unless manually adjusted. The work done by Skjermo et al (22) is largely based on the SMART algorithm and aim to solve the problems with the SMART algorithm.

Skjermo uses the Da Vinci rule (23) to ensure that the mesh will not self-intersect, which they base on the fact that in a natural branching system the child segment with the largest diameter in a branching point will also have the smallest angle compared to its parent segment. The algorithm takes as input a number of nodes, each defining their direction, length and radius which defines a segment in the mesh. The input nodes are sorted in a directed acyclic graph, and the algorithm recursively adds nodes to the graph. Starting with the root node, a cross section (i.e. a quad surface) defined by four vertices is created, and another cross-section is created along the direction and length of the root node. These two cross-sections, defined by eight vertices, define the top and bottom sides of a box that can then be closed, unless the node being processed has more children. In this case a special branching structure is created and the remaining children are sorted and added to the appropriate sides of the box. For each node added, its children are recursively added before returning to the parent node.

The algorithm developed by Skjermo et al assumes that at any given point one has access to the entire mesh, which is required in order to determine connection points for a new node in the mesh. For a GPU implementation this is not feasible since geometry is processed on a per-vertex, per-primitive or per-pixel basis. Furthermore, while high level shading languages such as GLSL and HLSL provide many of the same features as C and C++, one important omission is the pointers data-type. This is important to bear in mind since representing a graph as a data structure requires some sort of pointers.

3 Method

The method presented here is based on the work done by Skjermo et al in (22), but with the important distinction that it is a GPU implementation. The input data is sorted in a directed acyclic graph where each node has information about its direction, diameter and length.

At first, the graph is parsed by the CPU in a pre-processing step. In this step the graph is recursively processed, and for each node the position in world-space, the half-direction vector and up vector, is calculated, see 3.1.

Next the graph is sent to the GPU which, in a single pass, converts the graph to a coarse polygonal mesh. Starting with the root node a cross-section defined by four vertices is created. Next an in-memory cross-section representation of the first child node is created. These two cross-sections define the top and bottom sides of a box (in this case a cube where the edge lengths and angles are not necessarily equal) and in cases with no other children, the four sides of the box are closed as described in 3.2.2. In cases where there are more children, only the sides of the box with no child connections are closed as described in 13.2.1. Finally, special cases, where the angle between the direction of the first child of a node and the nodes direction is more than 90 degrees, are described in 3.1.3 and handled as in 3.2.1.

3.1 Pre-processing

In order to be able to convert a graph to a polygonal mesh it is necessary to determine the position of each node relative to other nodes in the graph. This is not feasible on the GPU as for each node to process we would need to send the entire graph in order to determine the position of that node. Instead we perform a pre-processing step which parses the graph input data and outputs it in such a way that it can be read and used by the GPU. First, however, we need to determine which data is required in order to construct the geometry for a single node. Any node in the graph, except the root node, will have one parent and all nodes will have {0, .., n} children. Since the geometry shader can only

output 1024 32 byte values (or 4096 bytes in total) for each primitive, and because we cannot work with any arbitrary primitive, we only process one node at a time in the geometry shader. This ensures that we can process a graph of any size in terms of nodes, whereas it complicates matters as we might need information about the parent and/or children nodes in order to construct the geometry for the current node. Consequently for any node in the graph we need to be able to retrieve the data for the parent and children nodes directly on the GPU. Traditionally, when working with vertices in shaders, the position of the vertex along with information such as its normal vector and texture coordinate are stored in the vertex buffer. However, this approach would not work here as one cannot index arbitrarily into the vertex buffer. Instead, for each node in the graph we create one vertex, store the information listed in Table 1, and add this vertex to the vertex buffer which is later sent to the GPU.

Vertex data	Description
Node index	Index for this node
First child node index	Index for the first child of this node
Parent node index	Index for the parent node
Parents first child node index	Index for the first child of the parent node
Number of children	Number of children for this node. If the node has more than one child, this is used to determine how many children to look up in the buffers.
Branching child	Whether this is a node connected to the side of a box or not. False for first child of a node, and true for the remaining children.

Table 1 Vertex buffer structure for graph nodes

To create the geometry for a node we need the following data:

- Position in world space
- Direction vector
- Diameter
- Up vector
- All of the above mentioned points for any parent and/or children

To make this data accessible to the GPU we create five 1D textures, the four most obvious being one texture that stores the positions, one that stores the direction vectors, one that stores the diameters and finally one that stores the up vectors. For every node in the graph, one value is stored in each of these four textures. The node index of each vertex can then be used to look up the respective values on the GPU. The fifth texture is used to store the indices for the children of each node. In this way, each node can look up its children and their data. Figure 1 shows the vertex values of three nodes in a graph, and how they can be used to look up the position and the indices for each child.

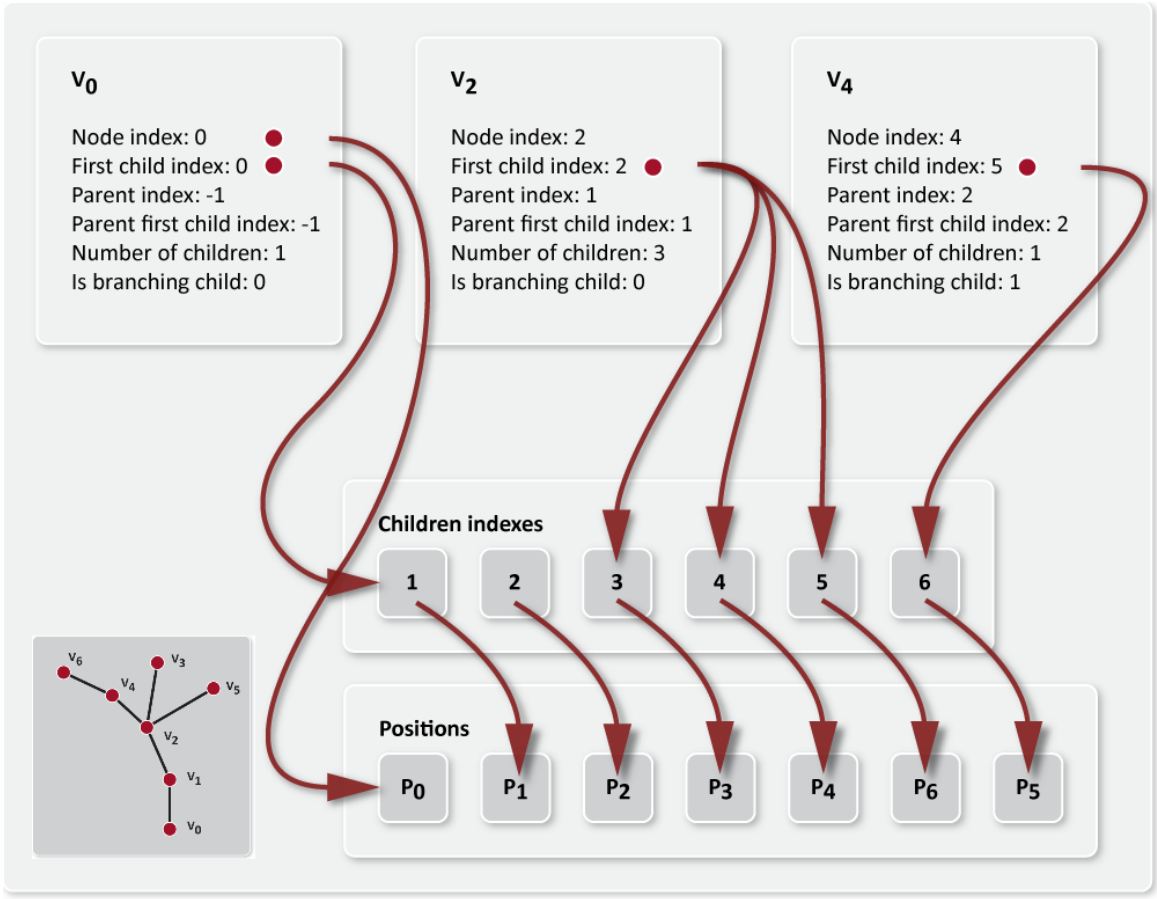

Figure 1 Simple graph and example of values stored in buffers

The actual parsing of the graph is done recursively in a number of steps as outlined in pseudo code in Table 2:

```
1. Store current node data in textures
2. Make room for child indices
3. Calculate half-direction for children
4. if node has one child then
        SingleChild (See Table 3)
5. else if node is a backwards connection
node then
        BackwardsConnection (See Table 4)
6. else if node has no children and is
not a dead end then
        DeadEnd (See Table 5)
7. else if node has more than one child
then
        Branching (See Table 6)
8. else if node is a branch connection
node then
        ChildBranch (See Table 7)
9. return node index
```

Table 2 Recursive pre-processing pseudo code

3.1.1 Step 1 - 3

The first step taken when pre-processing the graph is storing the position, direction, diameter and up-vector. The root node is positioned at the origin, but for all subsequent nodes the position is accumulated based on the previous node position and its direction. Next, one entry for each child of the node is added to the buffer containing child indices, but since we do not know their index yet their value is not yet set. This is done so we know that e.g. if a node has three children and the index of the first child is stored at index 2 in the child index buffer, the index of the second and third child will be stored at index 3 and 4 respectively (see Figure 1) in the child index buffer. Finally, the half direction for each child is calculated by taking the average of the direction of the current node and the direction of the child node.

3.1.2 Step 4 - SingleChild

```
4a. Check angle between nodes
4b. if angle is less than 90 degress then
4c.  Process child node (go to 1)
4d.  Save child node index in texture
4e. else if angle is >= 90 degrees then
4f.  Create middle and end node
4g.  Add end node and the child node as
child nodes of the middle node
4h.  Process the middle (a backwards
connection) node (go to 1)
4i.  Save middle node index in texture
```

Table 3 Pre-processing step 4

If the current node only has one child, the angle between the current node direction and the child node direction is evaluated. If the angle is less than 90 degrees, the up vector and the position is calculated for the child node. The position is simply calculated by adding the current nodes half-direction times its length to the current position. The child up vector is calculated by projecting the current nodes up vector on the half-direction vector of the child node and then projecting it onto the plane. The up-vector of the child node will now be perpendicular to its half-direction vector and point forward. Crossing the half-direction and up-vector will yield a third vector, the right vector, and using these three vectors it will be possible to determine which sides of a node a new child should be added to. See Figure 2.

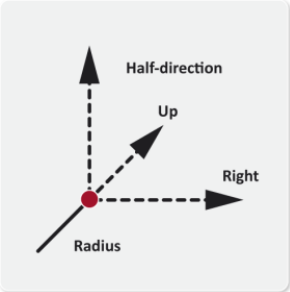

Figure 2 Graph node

In case the angle between the child node direction and the current node direction is greater than or equal to 90 degrees, two new nodes are added to the graph. This is done to ensure a closed mesh that does not self-intersect, see Figure 3 which shows an example of how the final mesh might look like, which on the left shows the mesh as it would have looked if there were no handling of backward connections, and on the right shows how it is handled:

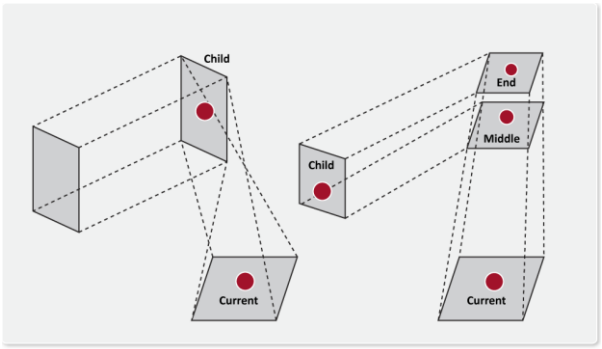

Figure 3 Backwards connection – Left: no handling of children pointing backwards. Right: Adding new nodes to graph to handle backward connections.

The end node is inserted along the direction of the current node times the length of the current node plus the radius of the child node. The middle node is inserted at the position of the end node minus the diameter of the child along the direction of the current node. The child node is removed as a child of the current node and added as a child of the middle node as is the end node. The middle node is flagged as a backwards connection mode and then processed.

3.1.3 Step 5 - BackwardsConnection

```
5a. Process end node (go to 1)
5b. Save end node index in texture
5c. Process child node (go to 1)
5d. Save child node index in texture
```

Table 4 Pre-processing step 5

Nodes flagged as backwards connection nodes first process the end node just as a normal connection node (see step 4), and then the child node is processed. However, since the child node has to be connected to one of the sides of the box formed by the middle and the end node, special care has to be taken. A new node is added to the graph as a child node of the middle node which is illustrated in Figure 4.

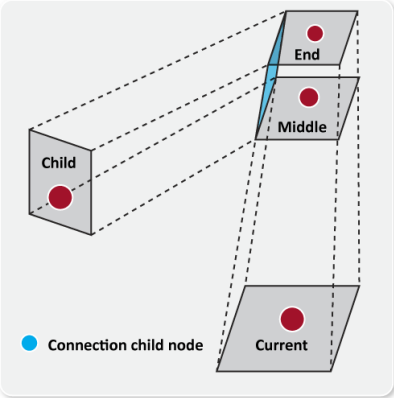

Figure 4 Backwards connecting child node

The position of the connecting child node is calculated by finding the middle point of the middle and end node, and then moving in the direction of the child node times the radius of the child node. The child node is then removed as a child from the middle node and added as a child of the connecting child node. Since the connecting child node will be positioned approximately in the centre of the box side the child node should be connected to, this makes it much easier to determine on the GPU what vertices of the middle and end node the child node should connect to. The connecting child node is flagged as a branching child (see Table 1) and then processed.

3.1.4 Step 6 - DeadEnd

```
6a. Create dead end node
6b. Process dead end node (go to 1)
6c. Save dead end node index in texture
```

Table 5 Pre-processing step 6

For nodes with no children, i.e. the end nodes of the graph, an extra node is added to the graph and this node is flagged as a dead end. The extra node will have the same direction- and up-vector as well as the same diameter and is handled just as in step 4 for nodes with angles less than 90 degrees.

3.1.5 Step 7 - Branching

```
7a. Find first child (largest diameter)
7b. Create middle and end node
7c. Make the first child a child of the
end node
7d. Add end node and the remaining
children as children of the middle node
7e. Process the middle node (a branch
connection node) (go to 1)
```

Table 6 Pre-processing step 7

In case the node has more than one child, and is not marked as a backwards or branching node, the method first determines which child node has the largest diameter. Then, as in step 5, a middle and end node is added to the graph and the first child node is added as a child of the end node. The remaining children are then added as children of the middle node and removed as children of the current node. Next, the middle node is flagged as a branching node and is then processed.

3.1.6 Step 8 - ChildBranch

```
8a. Process the end child node (go to 1)
8b. Save end node index in texture
8c. Sort remaining children in 4 sets, by
comparing their direction to direction of
current node
8d. Sort each child set by diameter
8e. for each child set
8f.   Make all but first child children
of the first child
8g.   Process first child (go to 1)
8h.   Save first child index in texture
```

Table 7 Pre-processing step 8

For nodes marked as a branching node their first child, the end node, is processed first just as in step 4. Next, the remaining children are sorted into four sets by comparing the angle between their direction and the direction of the current nodes up vector, and the current nodes right vector. The sets with children are then sorted by diameter, and are then processed one at a time.

When a set is processed a new node is created at the side of the box of which this set of nodes should be connected. This node is created with the diameter of the first child in the set (the one with the largest diameter) and is created parallel to the side it is created at; see 13.2.1 for an

explanation of why this is done. Finally, the nodes in the set are added as children to the newly created node.

3.2 Shader implementation

Once the entire graph has been pre-processed the vertices (one vertex for each node in the graph) are sent to the GPU in a vertex buffer which is described in Table 1. The geometry shader processes primitives and since we only want to process one node in the graph at a time, we will be working with point primitives (i.e. a single vertex) in the geometry shader. The geometry output for each point varies with the data for that point. The point for the root node in the graph will output a cross-section and so will points for dead end nodes in the graph. All other points will only output the vertices needed to connect to its children.

3.2.1 Non-branching nodes

All non-branching nodes are dealt with in a similar fashion. A cross-section, defined by four vertices, with the radius one is created around the origin and then translated by the radius of the node as illustrated in Figure 5.

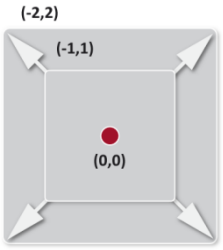

Figure 5 Four vertices, forming a cross-section with radius 1, created around the origin and then adjusted according to the radius (2 in this case) of the vertex being process

To position and rotate the cross-section correctly, the four vertices must be transformed. First the following matrix is created:

$$
\begin{aligned}
zaxis &= |At - Eye| \\
xaxis &= |Up \times zaxis| \\
yaxis &= zaxis \times xaxis
\end{aligned}
$$

$$
\begin{bmatrix}
xaxis.x & yaxis.x & zaxis.x & 0 \\
xaxis.y & yaxis.y & zaxis.y & 0 \\
xaxis.z & yaxis.z & zaxis.z & 0 \\
-(xaxis \bullet eye) & -(yaxis \bullet eye) & -(zaxis \bullet eye) & 1
\end{bmatrix}
$$

Equation 1 Left-handed look-at matrix definition

Referring to Equation 1, the Eye vector is the position of the vertex, the At vector is the position of the vertex plus the half-direction and the Up vector is the up-vector for the vertex as calculated in the pre-processing step. The reason why the half-direction- and up-vector are not calculated in the shader is that they are based on the half-direction and up-vector of all parent nodes in the graph. While we could traverse the entire graph and look up all parent nodes in the texture buffers and calculate the half-direction and up-vector directly in the shader, this would require at least n x 3 texture look-ups for the n'th level in the graph (a look-up to get the parent index, direction and up-vector). While doable, it is much faster to do this offline in a pre-processing step.

Once the look-at matrix has been created, it is then inverted. Using a camera analogy, this is done as the look-at matrix transforms from where we are looking from to the origin, but we are interested in a matrix that transforms from the origin to where we are looking from.

Before the cross-section can be output, it is necessary to determine the normal vector at each vertex. This is done as shown in Equation 2:

$$V_{AB} = V_B - V_A$$
$$V_{AC} = V_C - V_A$$
$$V_{normal} = \hat{(V_{AB} \times V_{AC})}$$

Equation 2 Vertex normal

Here A, B and C are vertices of the cross-section. To ensure the normal is not pointing inwards, the parent or child cross-section (depending on whether the vertex being processed is a dead end or any other node) is also created. The centre position between the current cross-section and the parent or child cross-section is found, and by examining the dot product between the normal and a vector pointing from the centre of the current cross-section to the centre position of the box, it can be determined whether the normal is pointing inwards or outwards.

Once the vertices of the two cross-sections have been transformed, and the normals of each vertex found, the cross-section of the current node may be output to the stream-output stage of the geometry shader. However, this is only done if the node is a dead end node or a root node. Finally, for all other nodes than dead end nodes, the sides between the box formed by the current cross-section and its children are closed as described in 3.2.2.

3.2.2 Closing the sides between two cross-sections

Whenever a cross-section is created, either in memory (for nodes in the graph with a parent and one or more children) or by actually outputting it (for the root node and dead end nodes) it is determined which, if any, sides of the box formed by the cross-section and the cross-section of its first child, should be closed. For nodes with only one child all sides are closed, but for nodes with multiple children the following equation is solved for each child:

$$V_{right} = |Up \times HalfDirection|$$
$$a = ChildHalfDirection \bullet Up$$
$$b = ChildHalfDirection \bullet V_{right}$$

Equation 3 Child direction angle compared to parent

Examining the a and b values for each child enables us to determine which sides should be closed.

To actually close the sides of the box, the eight vertices defining respectively the current nodes cross-section and the cross-section of the first child node are reused. The normal for each vertex of each side is calculated as explained in 3.2.1. Figure 6 shows how a very simple graph is processed.

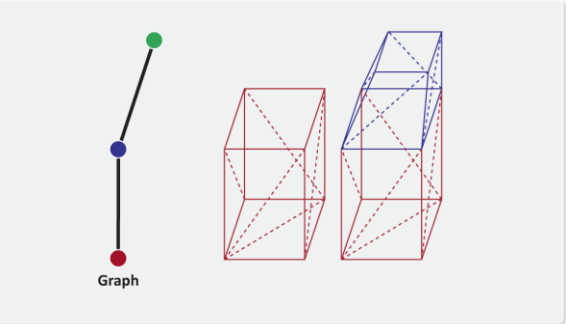

Figure 6 – A simple graph and an illustration of how the first two nodes are processed. 1) Root node processed. 2) Child of root node processed. Dashed lines denote closed sides. Note how the cross-sections between the root- and child-node, and the cross-section at the end of the child node are not closed.

To ensure a minimum amount of twist, and to avoid self-intersection, it is necessary to determine which vertices should be paired. Although all vertices are created around the origin, they are later translated and more importantly also rotated which could cause the mesh to self-intersect. Since the four vertices of the current nodes cross-section and the four vertices of the child nodes cross-section can only be paired in 24 different ways, we simply find the solution with the lowest total distance between

the paired points and use that to determine which vertices are connected to each other.

3.2.1 Branching nodes

Branching nodes, i.e. nodes created with the branching child flag (see Table 1), needs to be handled separately since they are not connected directly to their parent node, but to one of the sides formed by the box defined by the parent nodes cross-section and the cross-section of the first child of the parent node. Furthermore, multiple children might have to be connected to the same side as illustrated in Figure 7.

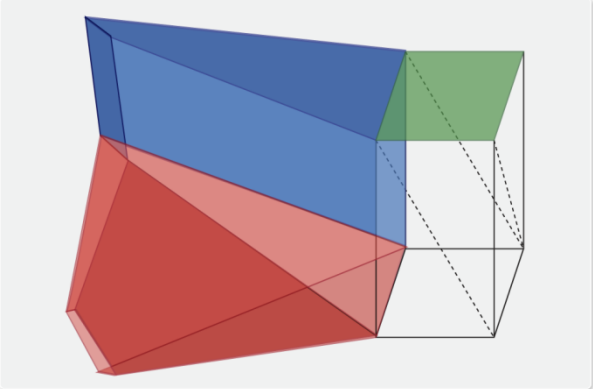

Figure 7 - Two branching nodes on same side. First, the blue child (with largest diameter) is connected to the cross-section of the parent (in black) and the cross-section of the first child (in green). Next, the red child is connected.

In the example illustrated in Figure 7, the second branching child (in red) is connected to the first branching child (in blue) and the parent cross-section (in black). However, had the second branching child pointed in another direction, it might have had to be connected to the first branching child (in blue) and the parents first child (in green) instead. If we knew we never had to consider more than two branching children in each direction, this would be possible to handle, but with no limit to the number of branching children in one direction we would soon have to process a lot of nodes and recreate all their cross-sections in order to determine connection points. Instead, a much simpler solution is proposed here in this paper. As explained in 3.1.6, a new node is added to the graph with the diameter of the first branching child, and the direction of the side the branching node is connected to. Figure 8 illustrates this branching scheme.

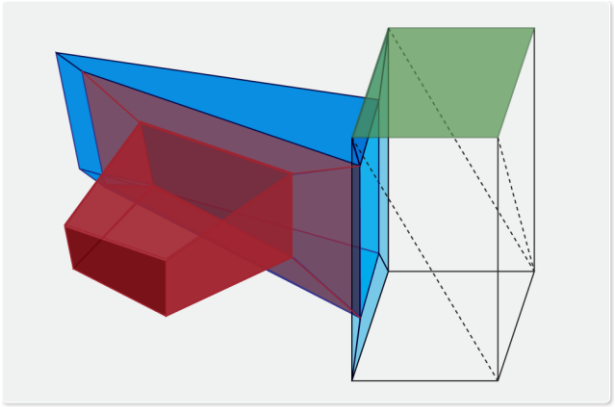

Figure 8 - Modified branching solution

The first step in doing this is finding the vertices of the parent cross-section and the parent's first child cross-section the branching child should connect to. This is done by recreating the cross-sections for the parent and the parent's first child, and then calculating the centre points of each of the four sides. The side from which the distance between its centre point and the centre point of the branching child's cross-section is lowest is used as the side to connect the branching child to. Finally, the vertices of the branching child's cross-section and the four vertices of the side it is supposed to connect to are paired as described in 3.2.2.

4 Tessellation

As an extension of the conversion of a graph to a polygonal mesh method, the Curved PN Triangles (24) method was implemented in a separate geometry shader pass. Unlike subdivision schemes like Catmull-Clark (4), the PN Triangles scheme work on a local flat triangle requiring only three vertices and three vertex normals. The method creates a cubic Bezier patch for each triangle in the original mesh and can in theory tessellate it into as many triangles as we want. Implementing the method is fairly

straightforward. It does not require any changes to the original mesh and it provides a much smoother mesh and thereby also better lighting. Since it works on a per-triangle basis, meaning it does not require any adjacency information, it cannot provide unlimited curvature to the mesh and it does not provide as “correct” results as e.g. Catmull-Clark. See 8.1 for a further discussion on this topic. Since the geometry shader stage is limited to outputting 4096 bytes per primitive, we cannot provide an infinite tessellation level in a single pass. A common output scenario for each vertex is shown in Table 8:

	Vertex data	Data Type	Bit Size
	Position	Vector 4	128
	Normal	Vector 3	96
	Texture coordinate	Vector 2	64
Total size (bits/bytes)			**288 / 36**

Table 8 PN-Triangles method geometry shader output

In the case depicted in Table 8, we would only be able to output a maximum of 113 vertices per primitive. One way of overcoming this is by doing multiple tessellation passes and outputting the intermediate results to the stream-output stage of the graphics pipeline. In this implementation we only perform a single pass though. We can provide automatic level-of-detail by increasing or decreasing the level of tessellation by examining the distance between the viewer and the mesh. Figure 9 shows a graph converted to a mesh and then the maximum achievable tessellation level in a single pass.

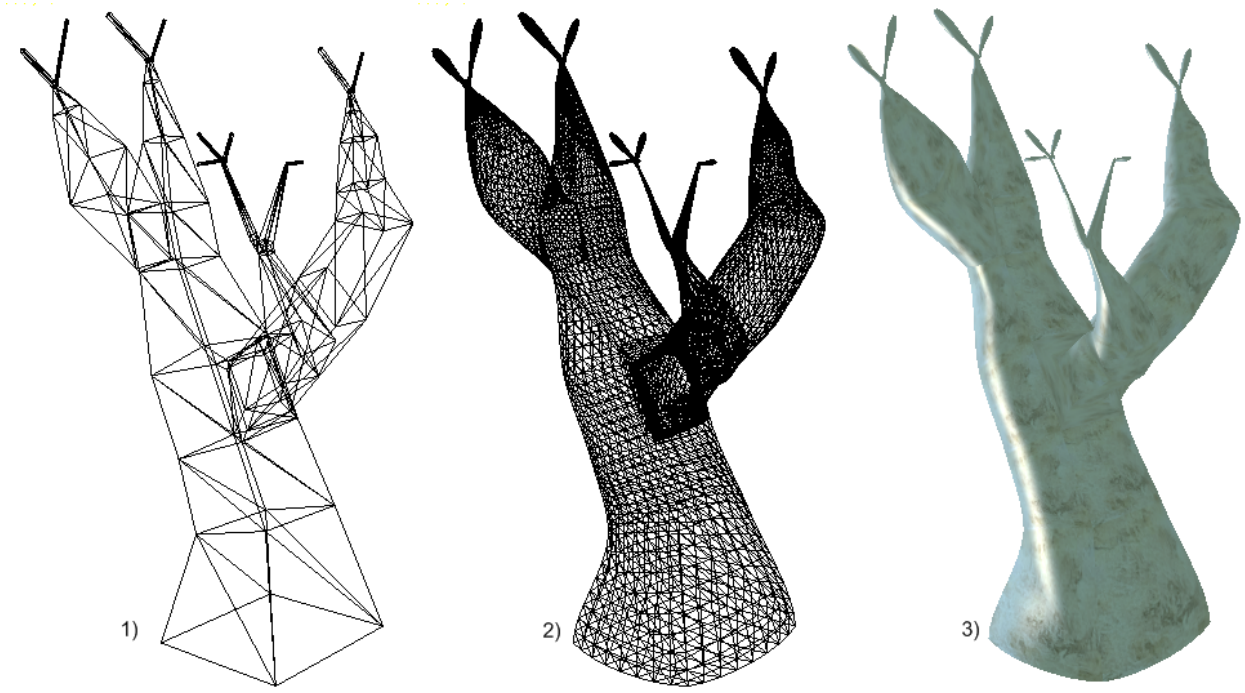

Figure 9 Tree visualized after: 1) Initial conversion 2) After 6 levels of tessellation 3) 6 levels of tessellation, textured and shaded

5 Results

The goal of this paper was to be able to convert a well defined graph to a polygonal mesh in the geometry shader stage in real-time scenarios. The results presented here were carried out on a notebook equipped with a 512MB GeForce 9600M GT graphics card and a Intel Core 2 Duo 2.53GHz processor, running the DirectX March 2009 SDK in a release configuration and profiled using PIX.

The graph data is converted to a polygonal mesh in the first draw event in a single pass. The mesh is then streamed out to video memory, allowing for it to be either read back into the pipeline in a subsequent rendering pass or read by the CPU. To make the mesh readable by the CPU, the constructed mesh must be copied to a staging buffer and then read by the CPU, which then fills a vertex and index buffer. Although this approach is not necessary it has some benefits. It allows the mesh to be exported to a file, making it possible, for instance, to use the technique in a 3D modelling application. Furthermore, it makes geometry instancing possible which may greatly improve performance in certain scenarios where you want to reuse the same geometry to draw multiple instances of the same object in a scene. However, storing the mesh in memory on the graphics card means much less bus communication between the CPU and GPU.

After the mesh has been created in the first pass of the first draw event, another pass then renders the mesh. Two methods have been implemented for this approach. The first method simply binds the mesh to the vertex buffer and then in a single pass tessellates and renders the mesh. The other method tessellates the mesh in a single pass if, and only if, there has been a change in the tessellation level and then streams the output to video memory. In a second pass, the mesh stored in memory on the graphics card is rendered. In theory this makes infinite tessellation possible as opposed to the first method in which the tessellated mesh is in effect discarded after every frame. However, it also means that a lot of memory may be required to store the mesh. Table 9 shows the polygon count per tessellation level for a simple cube using respectively the single pass method (method one) which stores the mesh in a vertex buffer, and the two-pass method (method two) which stores the mesh in video memory:

	Tessellation level	Method one	Method two
Polygons	1	12	12
	2	48	48
	3	108	192
	4	192	768
	5	300	3072
	6	432	12288
	7	588	49152
	8	768	196608

Table 9 Polygon count per tessellation level for a cube

Figure 10 shows a graph of a tree who has been converted to a polygonal mesh (2a and 2b) and afterwards tessellated 6 levels using method two. The conversion from the pre-processed graph to a polygonal mesh in this case takes approximately 0.87 ms creating a total of 438 triangles from an input of 65 vertices (or nodes in the graph). The time taken to pre-process the mesh is not included here as it could be done offline. Table 10 shows the profiled timing values for various tessellation levels of Figure 10:

Tessellation level	1	4	7
Method one	3 ms	17.7 ms	56.9 ms
Method two	0.13 ms	0.86ms	5.11 ms
Method one (no tessellation)	0.15 ms	-	-

Table 10 Performance of the two rendering methods

Method one is noticeably slower, and while some of the reason behind this can be explained by the CPU-GPU communication, the real bottleneck is

the tessellation routine which is applied on every pass. As shown in Table 10, removing the tessellation from method one yields almost the same results as method two does.

Figure 10 through Figure 13 show various graphs converted to polygonal meshes. They illustrate both possible applications as well as how branching is handled. For each figure, 1) shows the graph after pre-processing, 2a) and 2b) show the polygonal mesh constructed from the graph respectively shown as a wireframe and a solid model, and 3a) and 3b) show the model after a number of tessellation steps. Each figure also shows the time required to convert the graph to a mesh.

6 Conclusion and Future Work

A real-time solution for converting graphs to polygonal meshes has been presented. Provided well-defined input, the solution creates a closed non-intersecting polygon mesh on the GPU, and it has been shown that the process of converting a graph to a polygonal mesh can be done in real-time. For certain graphs with multiple branching nodes in the same direction, the method will perform certain adjustments to the graph in order to produce a closed non-intersecting mesh. This is not an ideal solution, and could be handled more gracefully at the cost of a more complex routine. An extension to the solution that increases the visual quality of the polygonal mesh by tessellating the mesh was also shown. This extension did not require any changes to the original work, and was also implemented directly on the GPU. It was shown that the tessellation procedure can also be done in real-time, but depending on how complex the scene is, high tessellation levels might not be possible to reach. That being said, this process of refining the mesh by tessellation is something that can be improved upon in the future. It would be obvious to look at the new hardware being introduced with DirectX 11, and especially how the new hardware tessellation stage can be utilized to provide true subdivision schemes like Catmull-Clark. Another area that would be obvious to cover is skinning (25) of the mesh, either through traditionally linear blend skinning or dual quaternion blending (26). Although true subdivision and skinning would go a long way to make this method more useful, a whole other area also needs to be considered; namely the creation of the graphs which we want to convert. Currently these have to be hardcoded, and although this can be tolerated in an environment like this, it would not be applicable in a working environment. This is why it would be very beneficial to be able to e.g. import skeletons of existing 3D models or integrate this method in a modelling tool. Finally, one area that has not been covered here is modifying the mesh by adding nodes, removing nodes or changing properties of nodes. Handling this is somewhat trivial however, as the graph would simple have to be converted once again. It could also be done be recreating only the parts of the mesh that are affected by the modification to the graph.

While there are many areas to look into, a method for creating a polygonal mesh from a graph has been implemented on the GPU and it has been shown that it is a feasible solution even in real-time rendering scenarios.

7 References

1. *Mathematical models for cellular interactions in development I and II.* **Lindenmayer, Aristid.** 3, s.l. : Journal of Theoretical Biology, March 1968, Vol. 18, pp. 280-315.

2. **Farin, Gerald.** *Curves and surfaces for computer aided geometric design: a practical guide.* San Diego : Academic Press Professinal, Inc., 1988. ISDBN: 0-12-249050-9 .

3. **Piegl, Les and Tiller, Wayne.** *The NURBS Book (2nd ed.).* New York : Springer-Verlag, 1997. ISBN:3-540-61545-8 .

4. **Catmull, E. and Clark, J.** Recursively generated B-spline surfaces on arbitrary topological meshes. 1978, Vol. 10, 6, pp. 350 - 355.

5. **Doo, Daniel and Sabin, Malcolm.** Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design.* 1978, 10.

6. **Chaikin, G.** An Algorithm for High-Speed Curve Generation. *Computer Graphics and Image Processing.* 1974, 3, pp. 346-349.

7. **Loop, Charles T.** *Smooth Subdivision Surfaces Based on Triangles.* s.l. : Master's Thesis, University of Utah, Department of Mathematics, 1987.

8. **Nira, Dyn, Leven, David and Gregory, John.** A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control. *ACM Transactions on Graphics.* April 1990, Vol. 9, 2, pp. 160-169.

9. **Zorin, Dennis, Schröder, Peter and Sweldens, Wim.** Interpolating Subdivision for Meshes with Arbitrary topology. *Proceedings of ACM SIGGRAPH.* 1996, pp. 189-192.

10. **Kobbelt, Leif.** A Subdivision Scheme for Smooth Interpolation of Quad-Mesh Data. *Eurographics.* 1998.

11. **Ou, Shiqi and Bin, Hongzan.** Subdivision method to create furcating object with multibranches. *The Visual Computer: International Journal of Computer Graphics.* 2005, Vol. 21, 3, pp. 170-187 .

12. *A Multiresolution Mesh Generation Approach for Procedural Definition of Complex Geometry.* **Tobler, Robert F., Maierhofer, Stefan and Wilkie, Alexander.** 2002 : IEEE Computer Society, 2002. Proceedings of the Shape Modeling International. p. 271. ISBN:0-7695-1546-0.

13. *A Generalization of Algebraic Surface Drawing.* **Blinn, James F.** 1982, ACM Transactions on Graphics.

14. **Hecker, Chris.** Chris Hecker's Website. [Online] Maxis/Electronic Arts. [Cited: 27 May 2009.] http://chrishecker.com/My_Liner_Notes_for_Spore#Creature_Skin_Mesh.

15. **Bloomenthal, Jules.** *Skeletal design of natural forms.* Calgary : University of Calgary, 1996. Doctoral Thesis.

16. **Bloomenthal, Jules.** An Implicit Surface Polygonizer. [book auth.] Jules Bloomenthal. *Graphics gems IV.* San Diego, CA, USA : Academic Press Professional, Inc., 1994, pp. 324 - 349.

17. *Marching cubes: A high resolution 3D surface construction algorithm.* **Lorensen, William E. and Cline, Harvey E.** New York : ACM, 1987. Proceedings of the 14th annual conference on Computer graphics and interactive techniques. pp. 163 - 169. SBN:0-89791-227-6 .

18. *Development models of herbaceous plants for computer imagery purposes.* **Przemyslaw, Prusinkiewicz, Lindenmayer, Aristid and Hanan, James.** New York, NY, USA : ACM, 1988. Proceedings of the 15th annual conference on Computer graphics and interactive techniques. pp. 141 - 150. ISBN:0-89791-275-6.

19. **Hanan, James Scott.** *Parametric l-systems and their application to the modelling and visualization of plants.* s.l. : The University of Regina, Canada, June 1992. p. 192, PhD thesis. ISBN:0-315-83871-X.

20. *Visual models of plants interacting with their environment.* **Mech, Radomir and Prusinkiewicz, Przemyslaw.** New York, NY, USA : ACM, 1996.

Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. pp. 397 - 410. ISBN:0-89791-746-4.

21. *SMART - Surface Models From by-Axis-and-Radius-Defined Tubes.* **Felkel, P., Fuhrmann, A.L. and Wegenkittl, R.** Vienna : VRVis Center, 2001.

22. *Polygon Mesh Generation of Branching Structures.* **Skjermo, Jo and Eidheim, Ole Christian.** Trondheim : s.n., 2005, Vol. 3540.

23. *The Notebooks of Leonardo da Vinci.* **Richter, J.P.** s.l. : Dover Pulications, 1970, Vol. 1.

24. *Curved PN triangles.* **Vlachos, Alex, et al.** New York, NY, USA : ACM, 2001. Proceedings of the 2001 symposium on Interactive 3D graphic. pp. 159 - 166 . ISBN:1-58113-292-1 .

25. *Joint-dependent local deformations for hand animation and object grasping.* **Magnenat-Thalmann, N., Laperrière, R. and Thalmann, D.** Edmonton, Alberta, Canada : Canadian Information Processing Society, 1989. Proceedings on Graphics interface. pp. 26 - 33 . ISSN:0713-5424.

26. *Geometric skinning with approximate dual quaternion blending.* **Kavan, Ladislav, et al.** 4, New York, NY, USA : ACM, October 2008, ACM Transactions on Graphics, Vol. 27, p. Article No. 105. ISSN:0730-0301.

27. *The Direct3D 10 System.* **Blythe, David.** Boston, Massachusetts : ACM, 2006. ACM SIGGRAPH 2006 Paper. pp. 724 - 734. ISBN:1-59593-364-6.

28. *Approximating Catmull-Clark subdivision surfaces with bicubic patches.* **Loop, Charles and Schaefer, Scott.** 1, New York, NY, USA : ACM, 2008, Vol. 27. ISSN:0730-0301 .

29. **Klein, Allison.** Gamefest. *Gamefest presentations.* [Online] Microsoft, 22 July 2008. http://www.microsoft.com/downloads/details.aspx?FamilyID=E410716F-12BF-4E8F-AC41-97B4440C3B90&displaylang=en.

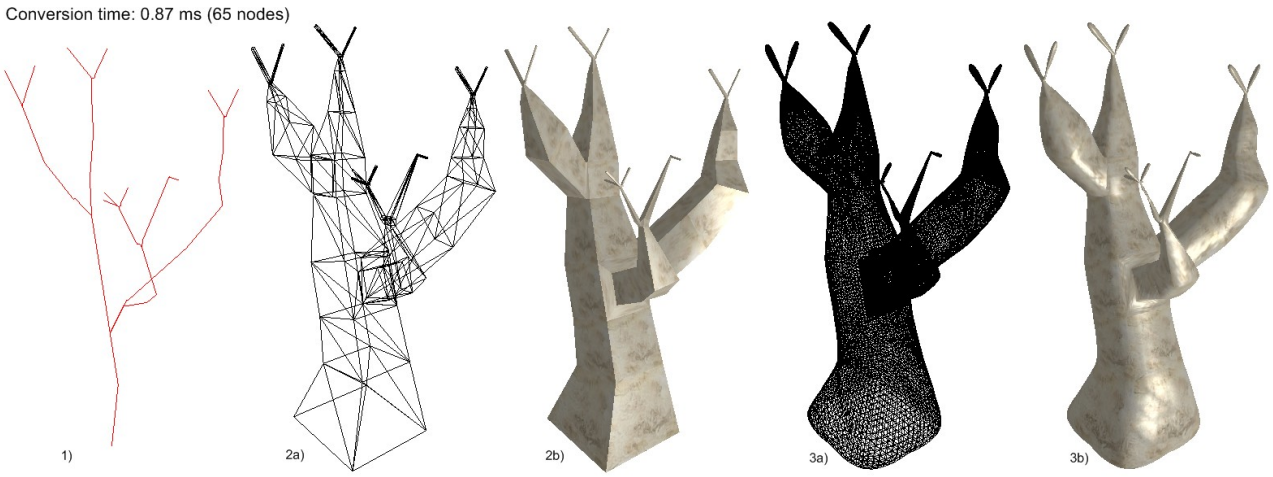

Figure 10 – Tree graph converted to polygonal mesh. 1a) Wireframe and 1b) Solid mesh as converted. 2a) Wireframe and 2b) solid after 6 tessellation steps

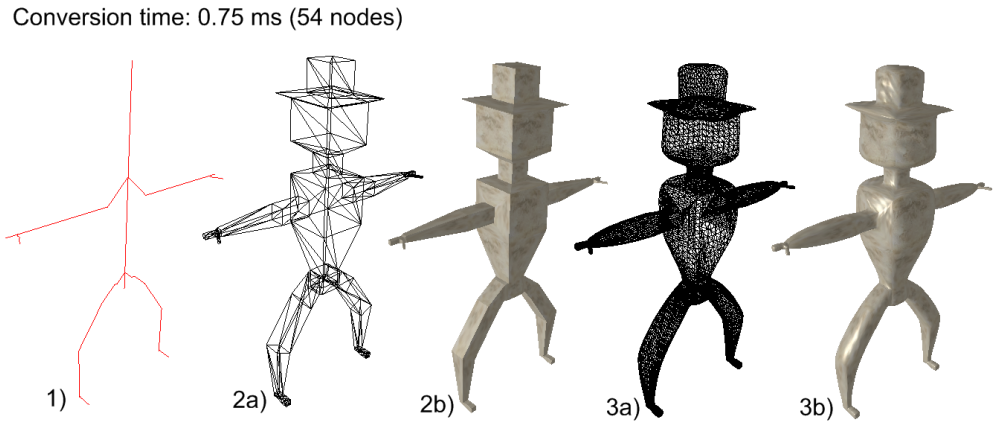

Figure 11 - Stick man graph

Figure 12 - Multiple branching in a single node

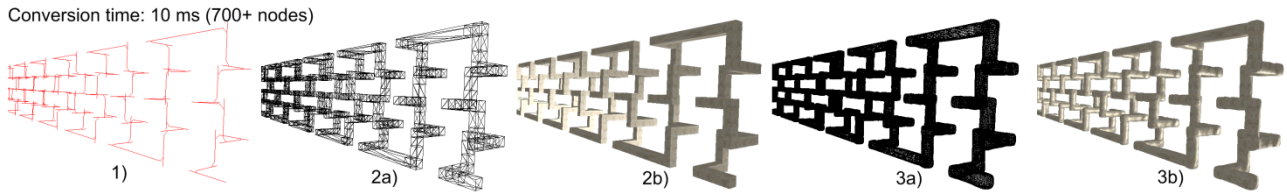

Figure 13 - Part of a large graph

8 Appendices

8.1 Appendix A – Direct3D 10 and 11 Graphics Pipeline

One of the major changes going from previous releases of DirectX and working with Direct3D 10 is that the fixed-function pipeline is now history and has been replaced by a unified programmable graphics pipeline, meaning the entire scene has to be rendered using vertex and pixel shaders. Figure 14 shows an outline of the new graphics pipeline.

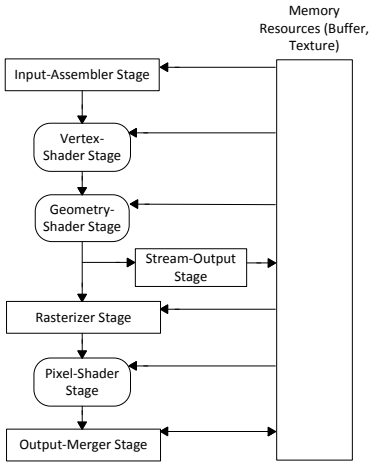

Figure 14 - Outline of the Direct3D 10 pipeline

A fixed-function pipeline architecture is limited by what the hardware supports and which extensions are available. Some stages, like transformation and shading, are either fixed or only partly configurable, but in a programmable pipeline architecture these stages can be directly controlled in shaders. For obvious reasons this is far more flexible and makes implementing certain effects that were previously impossible to do without extensions to the hardware, possible. However, you are still limited by e.g. the available memory and the number of instructions. One example is the animation of a character with a skeleton. Previously the skinning transformations had to be done on the CPU, but now these can be done – and much more efficiently - in the vertex shader stage, freeing up CPU resources to do AI, for example. Another major new feature to Direct3D 10 is the new shader model 4.0, which has more than 100 times the number of instruction slots for pixel shaders, integer operations and support for geometry shaders to name a few (27). Geometry shaders are new to Direct3D 10 and where vertex shaders work on vertices, pixel shaders on pixels, the geometry shader processes primitives: points, lines or triangles, and possible with adjacency information. The geometry shader can modify the primitives, output new primitives or discard them or parts of them. It is limited to outputting 1024 32 bit values per primitive and in addition to outputting to the rasterizer stage the output data can also be streamed to the stream-output stage. The data streamed to the stream-output stage is written to a buffer in memory on the graphics card, and apart from the ability to copy it back to the CPU it can also be fed back in the input-assembler stage or read by shaders using newly introduced load functions. There are many applications of the geometry shader, including implementation of algorithms such as:

- Shadow volumes
- Fur/Fins generation
- Dynamic particle systems

The geometry shader, in theory, is also a good fit for subdivision or tessellation. However, given that it is limited to outputting 4096 bytes per primitive it would only be able to provide a fixed amount of tessellation in a single pass. While running more passes is possible, the performance of the geometry shader also severely diminishes as the output increases. In recent years, as hardware has become more powerful, meshes of objects and characters have also become much denser. This causes a number of problems as animations on polygonal meshes become more costly and I/O issues as they leave a much larger memory footprint. With the limitations of the geometry shader this unfortunately does not

offer a good solution to this problem, which leads us on the next section on Direct3D 11.

Direct3D 11 builds on Direct3D 10, but adds three new stages for tessellation plus a compute shader to the pipeline. Figure 15 shows the outline of the Direct3D pipeline with the new tessellation stages highlighted.

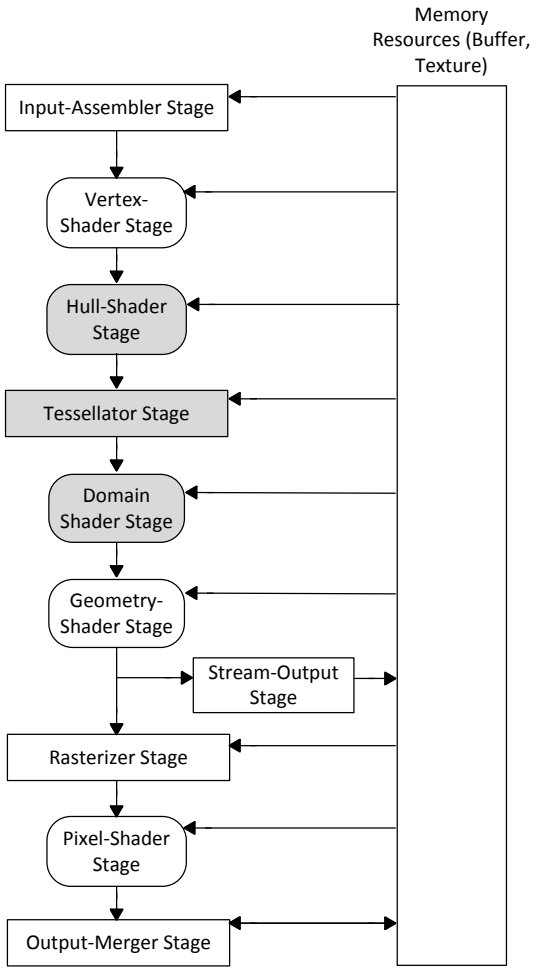

Figure 15 Outline of the Direct3D 11 pipeline

The new pipeline operates on meshes represented by surface patches, and both triangle and quad surface patches are primitives. Each patch is defined by a number of control points, which are transformed, skinned and/or morphed in the vertex shader. The hull shader is invocated once per patch, outputs the tessellation factor to the tessellator stage and optionally passes the control points after basis conversion directly to the domain shader, bypassing the tessellator. The tessellator operates per patch and uses the tessellation factors to tessellate the patch into multiple triangle or quad primitives, and outputs the UV {W} domain points resulting from this to the domain shader. The tessellator also outputs the topology for primitive assembly. Finally, the domain shader is invocated once per point from the tessellator, and outputs one vertex using the control points and tessellation factors received from the hull shader, and the U V {W} domain points received from the tessellator.

Both the hull shader and domain shader are programmable, whereas the tessellator is configurable. These new stages make it possible to implement techniques such as Catmull-Clark subdivision surfaces as described by (28). Implementing this technique could be done by inputting a Catmull-Clark control mesh, consisting of quads and adjacency information, to the pipeline and then going through the hull shader, tessellator and domain shader stages to do the subdivision. This would allow for a coarse mesh to be input and a very finely tessellated mesh to be output. Adding a displacement map to this method and a much more detailed mesh could even be produced as show in Figure 16:

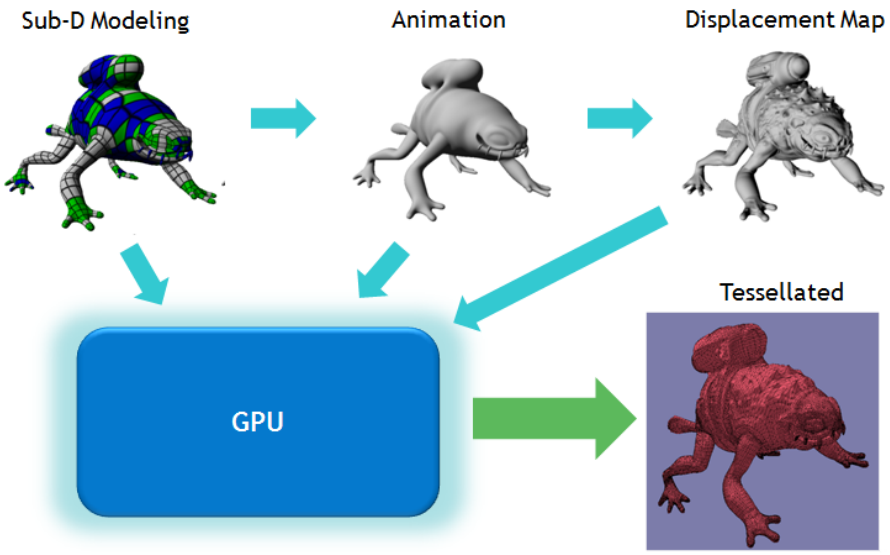

Figure 16 Taken from (29)

The advantages of these new stages for tessellation are plentiful, including reduced memory footprints, cheap /free level-of-detail (and thereby also reduced asset creation time) and reduced skinning costs.

There are many new features introduced with Direct3D 11, including the compute shader which enables much more general algorithms to be implemented on the GPU (also known as data parallel computing or GPGPU) and for example could be used for post processing effects, ray-tracing, physics or AI. Also worth mentioning is multi-threading and dynamic shader linkage.