Advanced Technologies: DirectX Based Plant Generator

Alexander Stephen Allman
16015338

University of the West of England

April 29, 2020

The aim for this task was to create a piece of software that the user could generate various types of plants and export them out into a format usable in a common engine such as Unity. This solution was built within DirectX and leveraged Dear ImGui to drive interactions.

1 Introduction

The creation of 3D assets for use in video games has been common place since the jump from 2D to 3D graphics, one of the first examples of this being Super Mario 64 from Nintendo ("Super Mario 64, Wikipedia" 1996). Since then the creation of 3D assets has become more accessible with software like Blender being free to all (Roosendaal, 2002). With this being true, it still requires a lot of both creative and technical knowledge to build quality assets. To help shorten the time-consuming process involved in creating 3D assets and allow novices to be able to create assets, a plant generator will be created leveraging DirectX 11.

2 Related Work

Since plant generation is such a time consuming process there are already examples of software that help with just that. SpeedTree ("SpeedTree: Vegetation modeling software" 2017) is an industry standard software that is used for the fast creation of realistic 3D foliage. It does this through building from a base foliage type and then adding different generators with tweakable parameters to reach the desired result. Whilst SpeedTree is the leading standard is foliage generation there are free alternatives such as TreeIt ("TreeIt" 2014) and ngPlant (Prokhorchuk, 2010), although the latter is no longer actively being developed.

3 DirectX 11

DirectX is a library that is used to interface with the GPU, allowing for the creation of high performance hardware accelerated 3D applications. In DirectX, this is done with two main components: Direct3D device, and Direct3D context. The device is used allocate resources and check feature support of the GPU, whereas, the context is used for drawing by setting render states and binding resources to the pipeline (Luna, 2012).

Once these components are initialised the Swap Chain can be created which will be used to present a frame to the screen. In this project a Double buffer Swap Chain is utilised. This is where there are two render views existing at any given time. The front buffer is used for displaying a completed frame to screen, whereas, the back buffer is used to draw the next frame to be presented. This method is used to prevent incomplete frames from being displayed which can result in flickering (Luna, 2012). The idea is illustrated in Figure 1, reading from the bottom of the figure upwards. The Swap Chain can also have more than two buffers but in most cases two is sufficient (Luna, 2012).

To render to these buffers the render pipeline is traversed.

3.1 DirectX Pipeline

Whilst there are two types of the DirectX pipeline, draw and compute (Hudek and Bazan, 2017a), this project will only utilise the draw pipeline as the compute capabilities of the compute pipeline is not required. There-

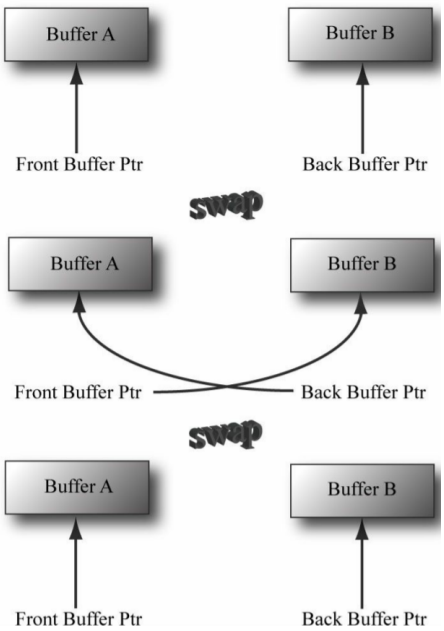

Figure 1: *Process of Swapping the Pointers between the front and back buffers, known as "Presenting" (Luna, 2012).*

fore from now onward the phrase "pipeline" will refer to the draw pipeline.

Each step in the pipeline will take an input (most of which is from the previous step in the pipeline) and generate an output, each step does something different to build the overall frame rendered by the Swap Chain. The full pipeline can be seen in Figure 2.

Not all stages are mandatory for drawing a frame; this project utilises the "Input Assembler", "Vertex Shader", "Rasterizer", "Pixel Shader" and the "Output Merger".

3.1.1 Input Assembler

The Input Assembler is the entry point of the pipeline and it's where you bind vertices and indices to. It does this by using fixed function operations to read the vertices from memory and this results in pipeline work items to be passed on to the Vertex Shader (Hudek and Bazan, 2017b). The vertices and indices are stored in their own buffers respectively. Buffers in DirectX are created in a generic manner where the data is stored on creation of the buffer (Satran, 2018c). The size of the stride in the buffer is determined by the size of each piece of data whether that be an int for an index or a struct containing vertex data. Whilst dynamic buffers are possible in which the CPU can remap data and the GPU reads it. To avoid the performance penalty of synchronising data between the CPU and the GPU; The vertex and index buffers are initialised with their data initially and not changed (Satran, 2018b).

This part of the pipeline also determines how vertices are connected to each other, this is known as the primitive topology. These range from each vertex not

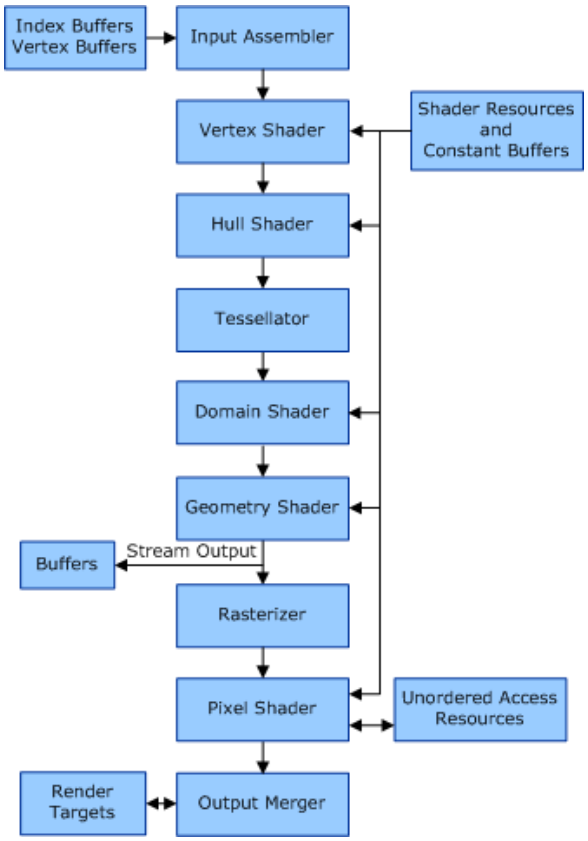

Figure 2: *The DirectX 11 Pipeline (Hudek and Bazan, 2017a).*

being connected at all, known as a point list; to triangle strips where once the first three vertices make a triangle, each vertex afterwards is used to create another triangle joining from two previous vertices (Satran and Jacobs, 2018). A visualisation of this can be seen in Figure 3.

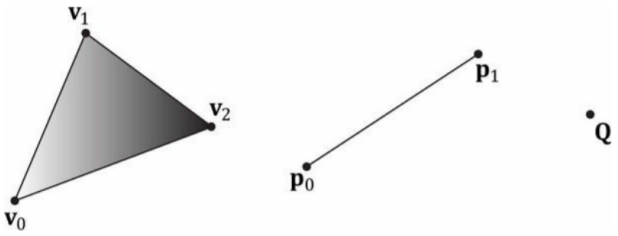

Figure 3: *Triangle, line and point primitive topology differences (Luna, 2012).*

3.1.2 Vertex Shader

The Vertex Shader is the next step in the pipeline. This step takes in a vertex and outputs a vertex. It is responsible for applying transformations to the vertices (Luna, 2012) and in this project, calculating the position and the normal of each vertex. This is done by passing in a model view matrix in a constant buffer to the Vertex Shader. This model world view matrix will built and bound to the pipeline for a specific set of

vertices, generally, a new constant buffer is bound for every new model to be rendered. Firstly, the world matrix of the specific model has to be built with its scale, rotation and position. So we build the scale, rotation and translation matrices respectively (Luna, 2012):

Scale Matrix:
$$\begin{bmatrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & zScale & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix: The rotation matrix is a little more complex as we have to make use of three separate matrices and multiply them in order of Roll, Pitch and Yaw (Jacobs, 2018). These matrices are built around the Left Hand Coordinate system as that is the DirectX default (Satran, 2018a).

Pitch Matrix:
$$\begin{bmatrix} \cos(\theta_x) & 0 & -\sin(\theta_x) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta_x) & 0 & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Yaw Matrix:
$$\begin{bmatrix} \cos(\theta_y) & \sin(\theta_y) & 0 & 0 \\ -\sin(\theta_y) & \cos(\theta_y) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Roll Matrix:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_z) & \sin(\theta_z) & 0 \\ 0 & -\sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation Matrix:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ xPos & yPos & zPos & 1 \end{bmatrix}$$

Building the Model World Matrix To create the final Model World matrix that will be applied to each vertex in a model, the Model's view Matrix must first be created:

$$ModelViewMatrix = ScaleMatrix \times RotationMatrix \times TranslationMatrix \quad (1)$$

Finally to build the Model World Matrix using the previous Model View Matrix and multiply it by the scenes camera matrix:

$$ModelWorldMatrix = ModelViewMatrix \times CameraMatrix \quad (2)$$

Before passing this to the GPU for processing the Model World Matrix must be transposed as these calculations are done in Row-Major order, whereas, HLSL will be expecting a matrix in Column-Major order (Satran and White, 2018).

3.2 Rasterizer

The Rasterizer is the next step in our program. The Rasterizer is used to determine some rules on how objects will be drawn in the scene.

Back Face Culling This is where the winding of triangles determines the direction the triangle is facing and if back face culling is enabled in the Rasterizer, it will avoid drawing the extra triangle on the rear side. This is done as more often than not, these triangles will be on the inside of an enclosed model and drawing them would be a waste as they would never be seen (Luna, 2012). An example of this can be seen in Figure 4.

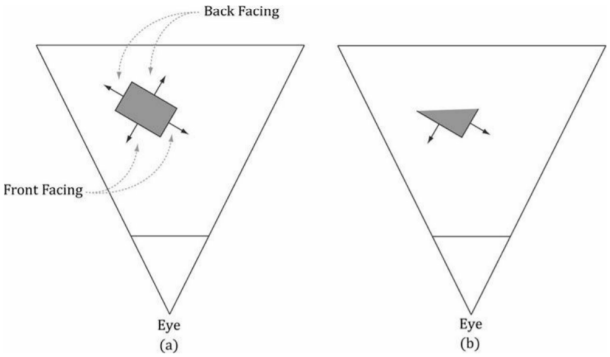

Figure 4: *Before and after back face culling (Luna, 2012).*

Viewport The Viewport determines how much of the window will be used to render in, In most cases this will occupy the entire window in games but in other applications where the rendered scene might only want to take up a section of the full window, it can be done using the Viewport. This can be seen in Figure 5.

3.3 Pixel Shader

The Pixel Shader is used to determine the colour each pixel fragment and utilises the output from the Vertex Shader (Luna, 2012). The Pixel Shader also runs on the GPU in a similar vein to the Vertex Shader and due to taking in interpolated Vertex information, can be used to calculate per-pixel lighting, shadowing or reflections (Luna, 2012). The Pixel Shader can also take extra information for calculations through the constant buffer and in this project are used for material and light information. The main lighting method utilised in this project is Phong.

Figure 5: *Example of Viewport within a Window (Luna, 2012).*

Phong Lighting To achieve Phong lighting three separate calculations must be brought together to get the resultant colour for that pixel fragment (Phong, 1975):

$$OutCol = Ambient + Diffuse + Specular \quad (3)$$

Calculating the Ambient component is the easiest as it is simply multiplying the base diffuse colour of the object whether that be a flat colour or a texture and multiplying it by the light. This is blanket applied to all objects in the scene in order to give everything in the scene some lighting to imitate light scattering without having to trace rays for this as we want the application to be real time.

$$Ambient = i_a * \frac{\rho}{\pi} \quad (4)$$

Both the diffuse and specular have a weighting related to each of their components such that:

$$w_d + w_s = 1 \quad (5)$$

The higher w_s is, the glossier the surface will be as it will reflect more light. The sum of these can never be above 1 however otherwise the surface would be able to reflect more light than is hitting it.

Calculating the diffuse component takes into account the positions of lights in a scene relative to a point on the surface of an object. The intensity taken from this is dependant on the light striking point face on. If the light is perpendicular to the point then intensity is at it max; if the light is parallel to the point then intensity is at its minimum. A max function is used to ensure that if the point is facing away from the light that it will also be zero.

$$Diffuse = w_d * \frac{\rho}{\pi}(L_i * \frac{max(\omega_i \cdot N, 0)}{r^2}) \quad (6)$$

Calculating the specular component requires the use of the reflected direction vector to the light alongside the the direction vector to the camera from the point. The size of the specular is determined by the exponent e.

$$Specular = w_s * \frac{e+2}{2\pi}(\omega_o \cdot R(\omega_i, N)) * (L_i * \frac{max(\omega_i \cdot N, 0)}{r^2}) \quad (7)$$

The final equation can be refactored as such:

$$L = i_a\frac{\rho}{\pi} + [L_i\frac{max(\omega_i \cdot N, 0)}{r^2}(w_d\frac{\rho}{\pi} + w_s(\frac{e+2}{2\pi}(\omega_o \cdot R(\omega_i, N)^e)))] \quad (8)$$

3.4 Output Merger

This is the final step of the pipeline and is responsible for depth / stencil culling and writing the outputs of the Pixel Shader to the render targets stored on the swap chain.

4 Plant Types

The project has three different base plant types that have their own customisation options relating to that plant type, they are as follows:

4.1 Flower

The flower consists of two main elements. The first is the stem, the stem acts as the center of the plant in which petals and leaves are built around. The stem is built from a low poly cylinder.

The second component are the petals and leaves. These are both built from the same base system in which many small planes are grouped together and manipulated together. These planes have an additional translation before the usual calculations in order to offset them so that their pivot point is at the base of the plane rather than the center. The calculation is as follows:

$$PlaneModelViewMatrix = PivotOffsetTranslationMatrix * ScaleMatrix * RotationMatrix * TranslationMatrix \quad (9)$$

Each plane is then rotated on the X-Axis so that the spacing between each plane is equal:

Where n is the number of planes and p is the current plane:

$$\theta_p = (\pi/n) * p_{index} \quad (10)$$

These planes can then take a TGA file to be textured to whatever the user may want. Whether that be a leaf texture or a petal texture.

The planes can both be manipulated in a group so that rotations and translations affect all planes at the same time; or their rotations can be manipulated on a per plane basis.

An example of the Flowers and its customisation options can be seen in Figure 6.

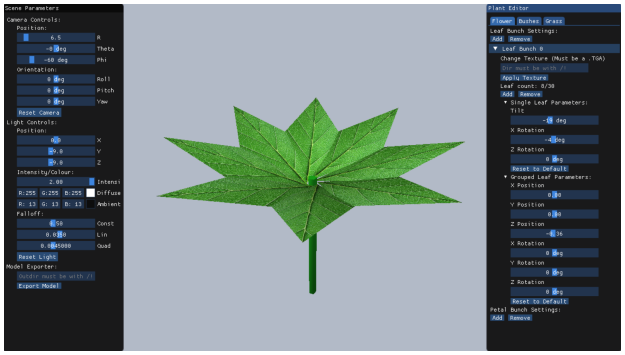

Figure 6: *The Flower and its customisation options*

4.2 Bushes

In the bushes editor the user can instantiate multiple bushes. Each bush consists of a set of double sided semi circles planes equally spaced. Each bush can be translated, rotated and scaled within world space. The amount of semi circle planes on each bush can also be increased to make the bush have a more full appearance.

An example of the Bushes customisation and its options can be seen in Figure 7.

Figure 7: *The Bushes and their customisation options*

4.3 Grass

The grass editor is in a similar vein to Bushes; in which the user can instantiate multiple blades of grass. These blades can be translated, rotated and scaled in the same manner as the Bushes but the key difference is the grass has a curvature modifier.

The blades each have their own dynamic vertex buffer, in which the vertices can be edited in the buffer during runtime. This allows the user to customise the base curvature by setting an offset and at every new segment of the blade the offset is added:

Where S is the X-Axis offset of the current segment:

$$S = S_{previous} + BaseOffset \tag{11}$$

An example of the Grass customisation and its options can be seen in Figure 8.

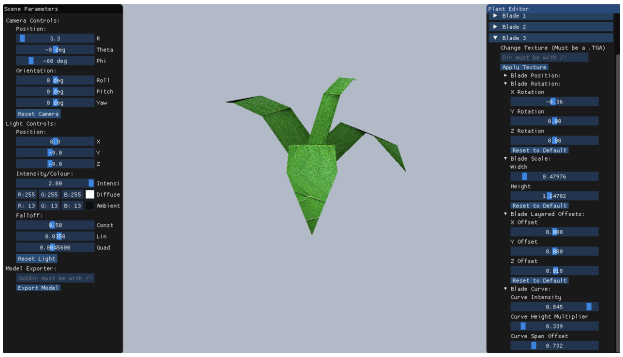

Figure 8: *The Grass and its customisation options*

5 Evaluation

The results of the software whilst are significantly lower quality than real plants due to utilising primitives, still represent the intended plant type.

Examples of all the plants and comparisons to real life counterparts can be seen in Figures 9, 10 and 11.

Figure 9: *The software's flower compared to a flower of similar nature*

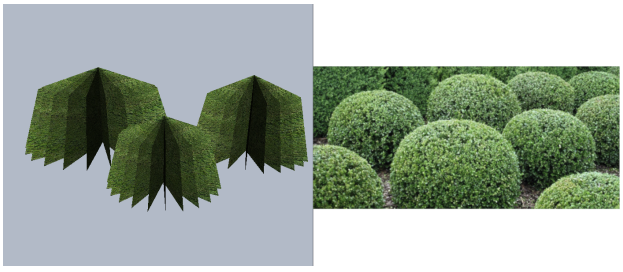

Figure 10: *The software's bushes compared to bushes of similar nature*

Figure 11: *The software's tuft of grass compared to a tuft similar in nature*

Once the user is finished creating their plant, it can be exported out into an OBJ and its accompanying MTL file. Examples of the plants within the Unity Engine can be seen in Figures 12, 13 and 14.

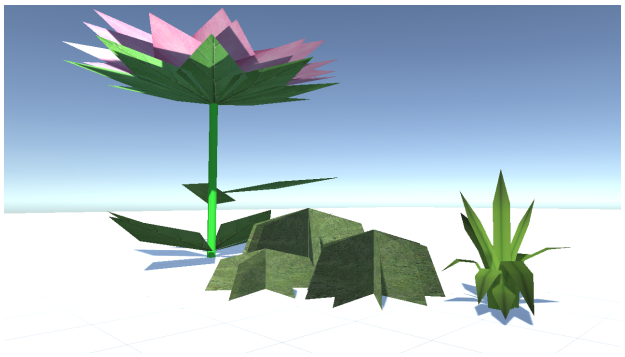

Figure 12: *The software's plants outputted into a scene in Unity*

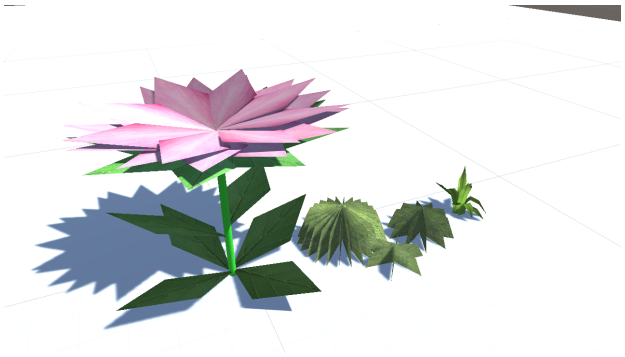

Figure 13: *The software's plants outputted into a scene in Unity*

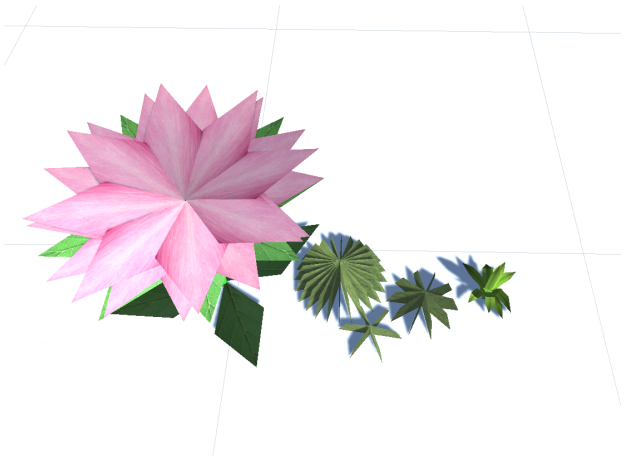

Figure 14: *The software's plants outputted into a scene in Unity*

curve short and shallow. Given a directory the software can then export the models and related texture data to be used in other 3D software, whether that be something more heavy duty like Blender (Roosendaal, 2002) or just as assets in a Game Engine.

6 Conclusion

In conclusion the project accomplished what was intended. It allows a user to create a small variety of plant types without any deep knowledge related to graphics and the anatomy of plants. Editing the parameters of the plants is done in real time with a simple interface. Since it is real time it gives the user instant feedback on what changed in relation to the adjusted parameter, which helps massively to keep the learning

Bibliography

Hudek, Ted and Nathan Bazan (2017a). "Pipelines for Direct3D Version 11". In: *Windows Drivers*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/display/pipelines-for-direct3d-version-11.

– (2017b). "Rendering Pipeline". In: *Windows Drivers*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/display/rendering-pipeline.

Jacobs, Mike (2018). "Math Library". In: *DirectX 11 Math*. URL: https://docs.microsoft.com/en-us/windows/win32/api/directxmath/nf-directxmath-xmmatrixrotationrollpitchyaw.

Luna, Frank D (2012). In: *Introduction to 3D Game Programming with DirectX 11*. Stylus Publishing, LLC.

Phong, Bui Tuong (1975). "Illumination for computer generated pictures". In: *Graphics and Image processing*. URL: https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf.

Prokhorchuk, Sergey (2010). "ngPlant". In: URL: https://github.com/stager13/ngplant.

Roosendaal, Ton (2002). "Blender". In: *Blender: The free and open source 3D creation suite.* Blender Foundation. URL: https://www.blender.org/about/.

Satran, Michael (2018a). "Coordinate Systems". In: *DirectX 9 Graphics*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d9/coordinate-systems.

– (2018b). "D3D11$_U SAGE Enumeration$". In: *Direct3D 11 Graphics*. URL: https://docs.microsoft.

com/en-us/windows/win32/api/d3d11/ne-d3d11-d3d11_usage.

Satran, Michael (2018c). "ID3D11 Buffer Interface". In: *DirectX 11 Graphics*. URL: https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nn-d3d11-id3d11buffer.

Satran, Michael and Mike Jacobs (2018). "Rendering Pipeline". In: *DirectX 11 Graphics*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-primitive-topologies.

Satran, Michael and Steven White (2018). "Per-Component Math Operations". In: *Reference for HLSL*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-per-component-math?redirectedfrom=MSDN#Matrix_Ordering.

"SpeedTree: Vegetation modeling software" (2017). In: IDV. URL: https://store.speedtree.com/.

"Super Mario 64, Wikipedia" (1996). In: Nintendo. URL: https://en.wikipedia.org/wiki/Super_Mario_64.

"TreeIt" (2014). In: Evolved Software. URL: http://www.evolved-software.com/treeit/treeit.