Plant Generator

Matt Filer
17021871

University of the West of England

May 6, 2020

This report outlines the creation of a plant generator in DirectX11, and the background research undertaken to implement it.

1 Introduction

Graphical toolkits are a useful component of any asset pipeline, with tools such as Substance Designer and Maya forming the industry standards for most artists. To produce these toolkits, rendering engines must be used, similar to that of the games themselves. This project focussed on a toolkit for producing plants through the use of pre-made model files which could be altered by artists to suit a requirement. DirectX11 was chosen for the rendering engine behind this tool as although version 12 supersedes it, 11 is still considered the baseline standard for most implementations due to memory handling changes which were implemented into 12.

This report will detail a look into the DirectX pipeline, along with the engine produced for the toolkit, capable of loading high-poly models in a modular format for editing live and saving as new assets.

2 DirectX Setup

DirectX is a rendering solution produced by Microsoft, first released in 1995. The technology inspired the release of the Xbox console, and has shipped with every Windows release since Windows 98. As of Windows 8, the DirectX SDK also shipped with default Windows installations (Walbourn, 2015). As with most APIs, DirectX requires a set of standard boilerplate code to be implemented in order to initialise its features. While some components of this boilerplate code are remnants of old Windows APIs which haven't been modernised, this case is still present in newer APIs such as Vulkan, and not an issue directly linked to DirectX itself.

Before initialising DirectX you must create a Windows window instance. This is done in three stages; first by calling `RegisterClassEx` which registers a window class object with the operating system (OS). This window class configures information such as icon, cursor, style, name, and global namespace wrapper for collecting events from the OS. Next, `CreateWindow` can be called which produces the window itself using the registered class instance, along with the window's title, style, and dimensions. Finally, `ShowWindow` can be called which shows the created window, with a given default state (for example: minimised, or maximised) (*ShowWindow Documentation*).

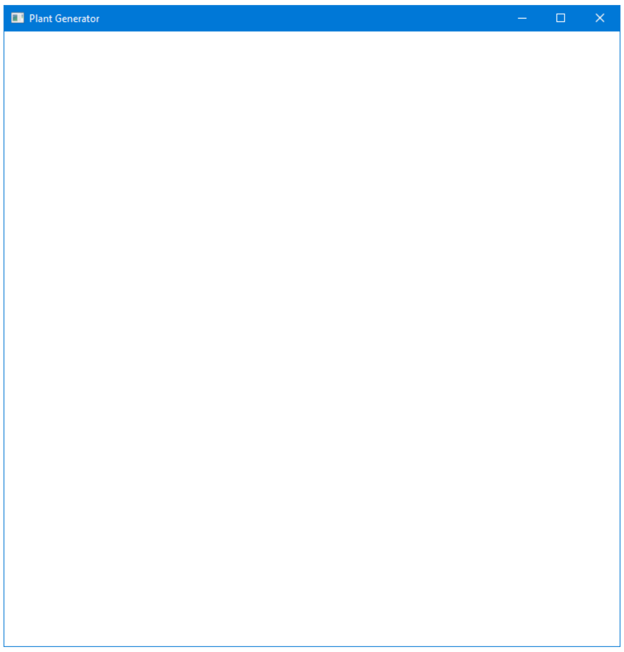


Figure 1: *An example window created with the CreateWindow/ShowWindow process.*

With the window created and shown, the first stage of DirectX initialisation can begin, which is to call `D3D11CreateDeviceAndSwapChain`. This function populates objects for device, device context, and swap chain. The device is created dependant on the given driver type, while the swap chain is created based on a given swap chain description object; both of which must be passed to the creation function. The swap chain description specifies important information such as the buffer size, format, count, and usage, as well as sampling parameters and the Windows window instance to target. The job of the swap chain is to handle surfaces that store data which has been rendered, ready to be displayed as the output in the window. After the swap chain and device have been populated, the device's render target can be set, which is usually configured as the back buffer from the swap chain: this gives a surface to draw to, and a way to render that out to the window using the system's rendering device (*What is a Swap Chain?*).

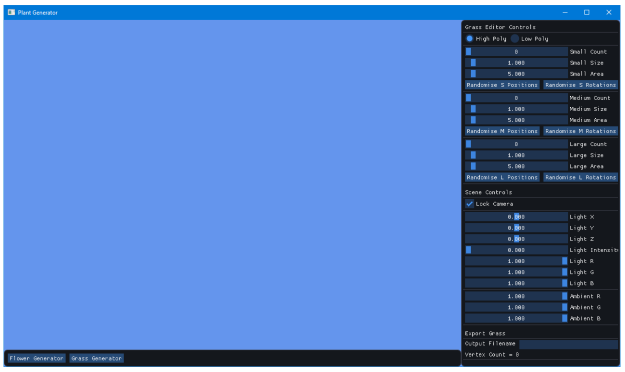
To allow for depth testing, next a depth stencil texture should be created. This is a Texture2D object which can be created with the device, given a descriptor with the binding flag of `D3D11_BIND_DEPTH_STENCIL`. Once created, this texture can then be used to create a DepthStencilView object, along with another descriptor which sets properties for the view such as dimension and format. Using this created DepthStencilView, the render target can then be set on the device context, using `OMSetRenderTargets`. This function supports setting multiple render targets, allowing for split-screen functionality if required. Similarly, you must set your viewport(s) up using `RSSetViewports` which defines the size of these render targets – important to match if using multiple, both in order and size.

A rasteriser state should be created and set to configure the properties of the rasteriser within the device context. To do this, similar to other setups, a descriptor must be created which defines how culling and filling is performed by the rasteriser. Selecting a fill mode of `D3D11_FILL_SOLID` will fill triangles created by vertices in geometry as expected to create surfaces, while `D3D11_FILL_WIREFRAME` will simply draw connections between the vertices to produce outlines of geometry. Once the rasteriser state has been created through the device using `CreateRasterizerState`, it can be applied to the device context using `RSSetState`.

DirectX11 is now set up for use, however the update/render loop must be configured. Essentially the update loop of any rendering app is an infinite while loop, interrupted by the Windows event generated by the user attempting to close the program (`WM_QUIT`). Within this infinite loop, typically operations are split into two functions: update and render; with update performing all logic such as moving objects around, and render clearing the frame, submitting all current draw calls to DirectX, and then presenting the frame from the back buffer to the front buffer for displaying. The render target can be cleared using the device context's `ClearRenderTargetView` function, and the back buffer can be presented using the swap chain's Present function (*ID3D11DeviceContext::ClearRenderTargetView method*).

3 Implementation

To create this toolkit, a simple engine was developed which could handle renderable objects of varying child types, able to be positioned and scaled within an environment. Using the DirectX11 setup as a base, numerous systems were developed on top to provide this functionality, as well as additional functionality for loading and saving models from object files (OBJ). A graphical user interface (GUI) was implemented utilising the open source library imGUI.



```
Grass Editor Controls
(●) High Poly  ( ) Low Poly
0        Small Count
1.000    Small Size
5.000    Small Area
Randomise S Positions  Randomise S Rotations
0        Medium Count
1.000    Medium Size
5.000    Medium Area
Randomise M Positions  Randomise M Rotations
0        Large Count
1.000    Large Size
5.000    Large Area
Randomise L Positions  Randomise L Rotations
Scene Controls
[✓] Lock Camera
0.000    Light X
0.000    Light Y
0.000    Light Z
0.000    Light Intensit
1.000    Light R
1.000    Light G
1.000    Light B
1.000    Ambient R
1.000    Ambient G
1.000    Ambient B
Export Grass
Output Filename
Vertex Count = 0
```

Figure 2: *The finalised imGUI controls.*

A scene manager was the key first step for developing this engine system. Scenes can be created within the project, inheriting from the parent scene object, and stored within a scenes array within the scene manager. The scene manager can then load and unload content within individual scenes at runtime, collecting resources together in an easy to load and dispose format. This ensures that not only are there no memory leaks through the whole scope of the application, but also that data that is deleted in an update call is not then called to render. A current scene index and requested scene index within the manager handled this, allowing the system to know the current state of operations and perform whatever is necessary. Clearing of the back buffer and presentation of back buffer to front buffer is also handled here to centralise operations.

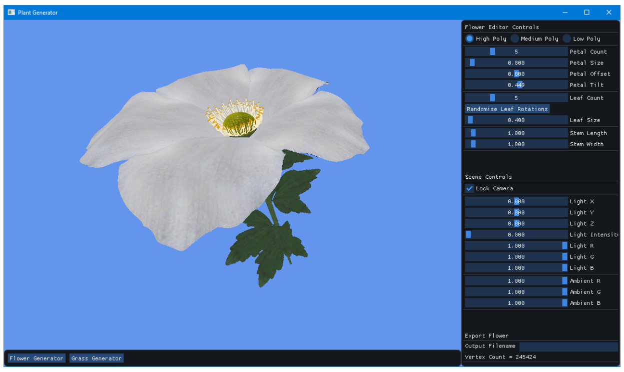
Renderable objects in the system, referred to as GameObjects, contain basic data in the child class for position, rotation, and scale. These values are translated (using the DirectXMath library) into a world matrix, which is interpreted by DirectX itself to calculate what to render from the camera's perspective. As well as this, in debug mode GameObjects also contain buffers for geometry: rendering a simple shaded cube. This can be shown or hidden when in a debug build in order to display the current position of all GameObjects, to test if loading or rendering is failing. GameObjects can be inherited from to create variants; with variants implemented for cubes, lights, models, and cameras. Each inherited GameObject takes a slightly different functionality on top of the base GameObject functionality, for example, cameras have no render call, can have their movement locked, and move as default based on user input. In the final implementation, the active camera GameObject's world matrix is used to render the scene from.

The model GameObject's buffers are populated from an OBJ file on the user's filesystem. To do this, the OBJ is parsed to pull its faces, which are each made up of three (if triangulated) vertices. Each vertex in the face has three array indexes, relating to data for the vertices position, texture coordinate, and normal vector. These indexes link to arrays stored within the OBJ for each set of data, with the arrays starting at index 1. This way of creating arrays for data and referencing indexes is a way of compressing the OBJ, as data does not have to be duplicated within the file. Although this saves filesize, OBJs are significantly larger than other model formats such as FBX due to them being plain text, rather than stored in binary values. This gives OBJs the benefit of readability, but can impact read speeds: every digit in a number is stored as a single byte char meaning, for example, a vertex with X position of 2.42564 would take up 7 bytes, compared to 4 in a float value within a binary file. That's almost half the file size for just one number – let alone more extreme cases, and other formatting within text files such as line breaks and indentation.

With the OBJ parsed into memory, this data can then be used to construct buffers in memory for DirectX to draw with. Vertexes go into the vertex buffer, and indexes of vertexes making up triangles for models go in the index buffer. This is essentially a direct transfer of data from the OBJ into DirectX's memory, however must be ordered depending on the device context's topology configuration: to set this, IASetPrimitiveTopology can be called, with `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST` matching the native way OBJ files are read; a linear list of triangle sets.

As OBJ files support submeshes within a model allowing for multiple materials (one per submesh), the model GameObject was created as a parent model object which transformations were applied to and render calls were performed from, however the object was made up of a number of submesh objects which each contained a segment of the model with its assigned material (pulled from the OBJ's MTL file). This allows for shader resources to be set per mesh part (and subsequently material), giving a single model multiple textures and colours. Working with native OBJ and MTL standards allowed the system to support a wide range of existing content.

As well as OBJ loading, the system was also required to write OBJ files back out from the editor. To achieve this, the same LoadedModel object was utilised which OBJ files were loaded into – taking model data for each part of the plant, transforming the vertices by the transforms applied by the user in-editor, and adding that data to a core LoadedModel object. Once all transformed parts of the plant were added to the same LoadedModel object, this could simply then be traversed to write out the vertex positions, texture coordinates, and normals for each face.



```
Flower Editor Controls
(●) High Poly  ( ) Medium Poly  ( ) Low Poly
5        Petal Count
0.800    Petal Size
0.000    Petal Offset
0.449    Petal Tilt
5        Leaf Count
Randomise Leaf Rotations
0.400    Leaf Size
1.000    Stem Length
1.000    Stem Width

Scene Controls
[✓] Lock Camera
0.000    Light X
0.000    Light Y
0.000    Light Z
0.000    Light Intensit
1.000    Light R
1.000    Light G
1.000    Light B
1.000    Ambient R
1.000    Ambient G
1.000    Ambient B

Export Flower
Output Filename
Vertex Count = 245424
```

Figure 3: *A flower being edited inside the tool.*

The editor interface was created using imGUI, an open source library compatible with DirectX11, which allowed implementation of simple checkboxes and sliders after connecting the imGUI render and update calls to the correct stages in the DirectX process, and giving it access to the input data. Using imGUI controls allowed for the positioning of models within the editor, creating the core of the plant generator: positioning loaded models to create a new design.

Models that could be loaded into the tool to be positioned by the imGUI UI were configured through a JSON file, parsed by the open source JSON for Modern C++ library. This allows the tool to be customised for different asset configurations if required; with custom offsets and model paths being configurable outside of code and a recompile.

To save loading new models every time something is added (for example, a new petal on a flower), only one GameObject is loaded for each segment of the plant, which is then updated and rendered multiple times depending on how many are required. This means that although only one GameObject is loaded it can be shown on screen an infinite number of times, without using more memory. This is particularly useful for high poly models, and it removes the need for runtime loading and also reduces the memory footprint of the application.

Another optimisation, mainly for the output content, was to provide options for high/medium/low poly geometry. These asset variants are specified in the content JSON file and allow models to be exported at lower detail levels if required. If the content being produced by the tool was intended for game use, this could serve as a way of producing different level of detail (LOD) variants for assets.

4 Conclusion

The final editor tool is simple to use and produces as good of an output as the assets loaded into it. Being able to customise the assets being loaded is useful as it means that it could be repurposed for different content without needing to touch the original code. The DirectX implementation is neat, the engine is tidy, and the code is performant – loading models in milliseconds and handling high poly density due to intelligent management of duplicated assets.

Although the final result is useful and performs well, it would have been nice to extend it further; allowing for modifiable vertex buffers at runtime. Using CPU accessible vertex buffers would've allowed for geometry to be warped at runtime, rather than just moved around and duplicated. Doing this would have allowed the artists more control over the final result of the content coming out of the tool, and allowed objects to differ more from the input given through the JSON file.

Bibliography

ID3D11DeviceContext::ClearRenderTargetView method. URL: `https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-clearrendertargetview`.

ShowWindow Documentation. URL: `https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow`.

Walbourn, Chuck (2015). "Where is the DirectX SDK (2015 Edition)?" In: URL: `https://walbourn.github.io/where-is-the-directx-sdk-2015-edition/`.

What is a Swap Chain? URL: `https://docs.microsoft.com/en-us/windows/win32/direct3d9/what-is-a-swap-chain-`.