Parker Britt | MSc Computer Animation and Visual Effects

Designing and Implementing Enzo: A Node-Based Parametric modelling Framework for Digital Content Creation

Lines of Code: 9508
Word Count: 3570

Contents

Introduction **3**
Problems 3
Scope 3
Previous work **4**
Technical Background **4**
Key Terms 4
Separation of Components **5**
UI Design **5**
Parameter Panel 5
Network Panel 6
Node Creation Panel 6
Viewport Panel 7
Geometry Spreadsheet 7
Engine Design **8**
Network Manger 8
Geometry 8
Attributes 9
Operators **9**
Dynamic Loading 9
Operator Table 9
Registering Plugins 10
Operator Definition 10
Operator Context 10
Parameter Template 10
Nodes **10**
Optimization **10**
Profiling 10
Multithreading 11
Devops **11**
Packaging 11
Cpack 11
RPM 11
TarGZ 11
Continuous Integration 12
Doxygen 12
Third Party Contributions **12**
Tests **12**
Conclusion **12**
Bibliography **13**

Introduction

This paper details the motivation, design, and implementation of Enzo, a node-based procedural 3D modelling framework for Visual Effects and 3D Animation. The project was driven by the need for an accessible alternative to existing industry software, which is often prohibitively expensive, closed source, and resistant to user driven innovation.

Enzo aims to demonstrate how procedural modelling tools can be designed with openness, extensibility, and usability in mind. Rather than attempting to replicate the scale of established applications like Houdini, the project focuses on creating a proof of concept, a lean but functional platform that lays the foundation for further development, experimentation, and community contribution.

Problems

There are several key problems with the state of current procedural 3D modelling software, and many minor ones. The first of these problems is licences and their prohibitive cost. Industry standard tools can range in price from £1,866 per year in the case of Autodesk Maya[1], £5,079 per year for Foundry Nuke[2], and $6,325 per year for Houdini[3]. When the yearly cost of a single commercial licence is one fourth the salary of a Junior VFX artist and far greater than a student can afford, it becomes completely unobtainable for individuals. These costs are recurring, meaning if you fall on tough times or are laid off, which has become unfortunately frequent in recent years, it becomes extremely expensive to produce show reel work or commercial work to recover. Higher costs also mean a higher barrier to entry, making creating art on computers less democratic and learning for students even harder.

Additionally, restrictive digital rights management to enforce these licences is built into the foundations of their software and files. It can be frustrating sinking months into building a tool for Houdini knowing that nobody will be able to use it commercial, severely limiting your user base.

Another key problem with existing software is most options are closed source. In its simplest form, closed source software means the source code written to compile the project is not available to the public, but this is an oversimplification. According to the Open Source Initiative[4] open source software has several tenets including free distribution, open source code, the ability to create derived works, and no discrimination against persons or groups. When software companies close the doors behind their technological innovations it stifles the industry growth that has long been built on open scientific communications. Since the early days of computer graphics scientists and engineers could build off each other's works, learning, collaborating, sharing papers and knowledge. This is a massive boon to the technological growth of computer graphics. One only has to look as far as the open source project, Blender, to see what kinds of growth can come from open source, communities openly collaborating, sharing knowledge, and building together. Closed source software dominating the industry threatens this.

Additionally open source software opens the doors for users to create the software they want to see. When using software it's all too common to find long standing gripes that are just not a priority for the company behind development. With open source software, users and developers have the ability to improve the software they love and build something better.

Scope

Houdini is a massive software, it's had the benefit of more than one hundred experienced developers working for the last 28 years. I'm a single junior developer with about 2 months. There's no way to build software to the scale of Houdini, and even then there's no end to potential features to be added. To keep the project in scope I aim to build a proof of concept to demonstrate a working basic

engine and GUI. Several nodes will be required to test the system and demonstrate a working result. For future work I would like to expand the system with more nodes, as well as more features, to demonstrate the potential projects that can be built using the software.

Previous work

The project's identity emerged from looking at existing node based DCCs, namely Blender geometry nodes, Tooll3, Euclide, TouchDesigner, and more. Ultimately, Houdini served as the main inspiration. Beyond being a favorite for procedural workflows, a testament to its design, Houdini provides an established success to reference. Working from reference rather than pure imagination helps avoid costly detours. Each decision into uncharted territory would potentially slow limited development time. This mirrors the difference between creating a model from imagination versus from a board of real life references.

The goal is not a one to one clone of Houdini, while it may currently miss many of the features Houdini has built up over the years, the idea is to make the changes I wish to see in Houdini. Fix Long standing gripes, add the features and quality of life that feel missing, and design the interface the way myself and others wished it functioned and looked. Crucially, accessibility for new users is a priority through simple design and naming changes. While not all of these things are possible in the duration of my thesis I would like to continue developing with these principles in mind.

Technical Background

When building a 3D asset there are many techniques that can be used to achieve the result, box modelling, sculpting, SDF, and procedural to name a few. Procedural modelling differs from traditional techniques in its declarative structure. Much like a programming language the model is built up from individual operations like extrude, copy, or convert to volume. These operations are not baked into the geometry, but evaluated live and per node. This creates a non destructive workflow where the user can at any point change any part of the process.

Key Terms

Nodes
Nodes represent individual operations, when chained together they create complex systems.

Attribute
Attributes are the basic building blocks of geometry that represent data like mesh information.

Parameter
Parameters are per node values that can be changed to modify the output of that node.

Owner
An attribute owner is the scope of the attribute and what part of the geometry the attribute is attached to. For example an attribute with a point owner, a point attribute, could represent the temperature of that point, a primitive attribute can tell the viewport whether that face is closed or open.

Cooking
Cooking is the process of computing a node's output.

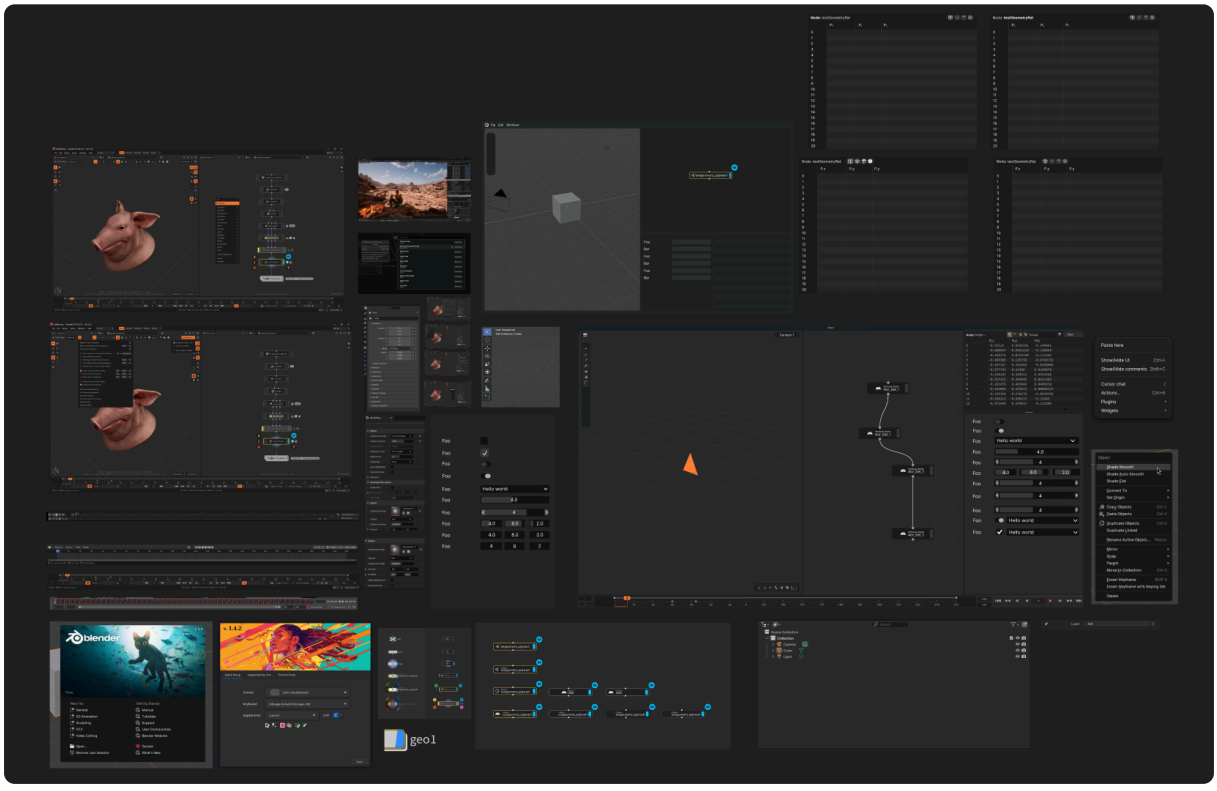
Dirty
A node or cache is dirty when it contains outdated information. This state is stored on the node and is used to lazily evaluate updates.

Separation of Components

I'd like this project to be one that can be developed beyond the Master's submission, so consideration had to be made on how the project was organized. The project is split into three separate parts, the front end, back end, and nodes. The back end is essential but the other two parts can be swapped out. The front end and backend mostly communicate through boost signals. The backend sends out signals to whatever process is listening, in the case of a headless design this may not be utilized. The UI then calls functions on the engine to manipulate it. This design prevents the engine from being coupled to the UI, allowing developers to build it into other tools, build different interfaces, or run it headlessly.

UI Design

Before building, I needed to figure out what the UI would look like. For this, Figma was used to prototype the design. Taking inspiration from modern flat interface design like Unreal Engine and Blender. I started laying out the look of essential systems. The interface is split into panels, each with an explicit purpose, separated by resizable splitters. The panels consist of interfaces for viewing and manipulating the network, viewport, parameters, and viewing underlying data with the geometry spreadsheet.



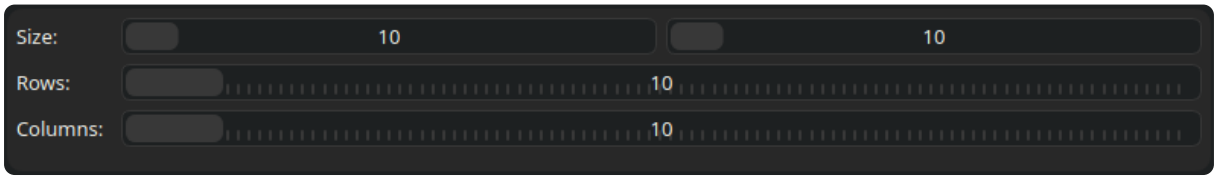
Parameter Panel

`src/Gui/ParametersPanel`

The parameter panel is built as a modular column of controls, each representing a property of the node cook process. Every control can optionally include a label that is padded to ensure all parameters line up neatly on the left edge for readability. The system supports a variety of parameter types such as integers, floats, booleans, strings, and vector inputs with multiple sliders arranged inline. Unlike Houdini which splits sliders and numeric fields, the design takes inspiration from Unreal Engine where sliders and numeric fields are combined into a single control. This saves space and makes vector inputs more practical by keeping the interface compact and efficient.

When a new node is selected, the panel clears any previous parameters and fetches the new set from the node definition. Interaction flows in both directions so changes made in the interface immediately update the backend while changes in the node state refresh the widgets. This ensures the panel always stays in sync and allows the program to modify the parameter state as well as the user.

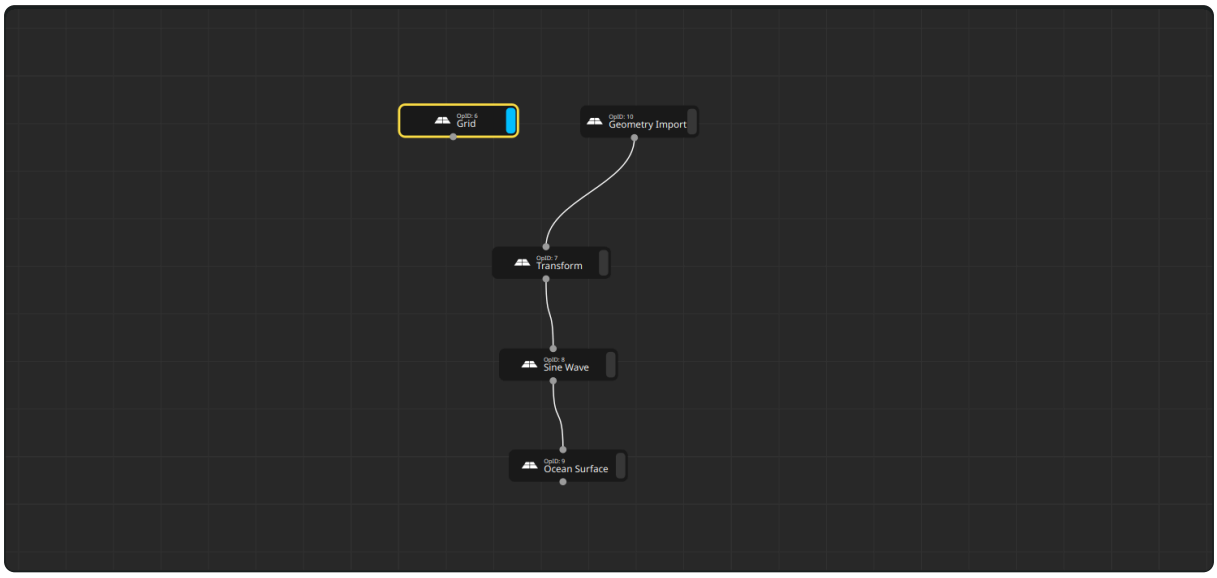
The parameter panel is made of parameter views, these are implemented as small self contained widgets that act as bridges between the data model and the user interface. Each parameter type has its own view class responsible for drawing the control, handling input, and sending signals when values change. These views are loosely coupled so new parameter types can be added without altering the overall panel system. When a parameter's state is modified, the panel notifies the network manager to dirty the corresponding node, triggering recooking when appropriate.



Network Panel

`src/Gui/Network`

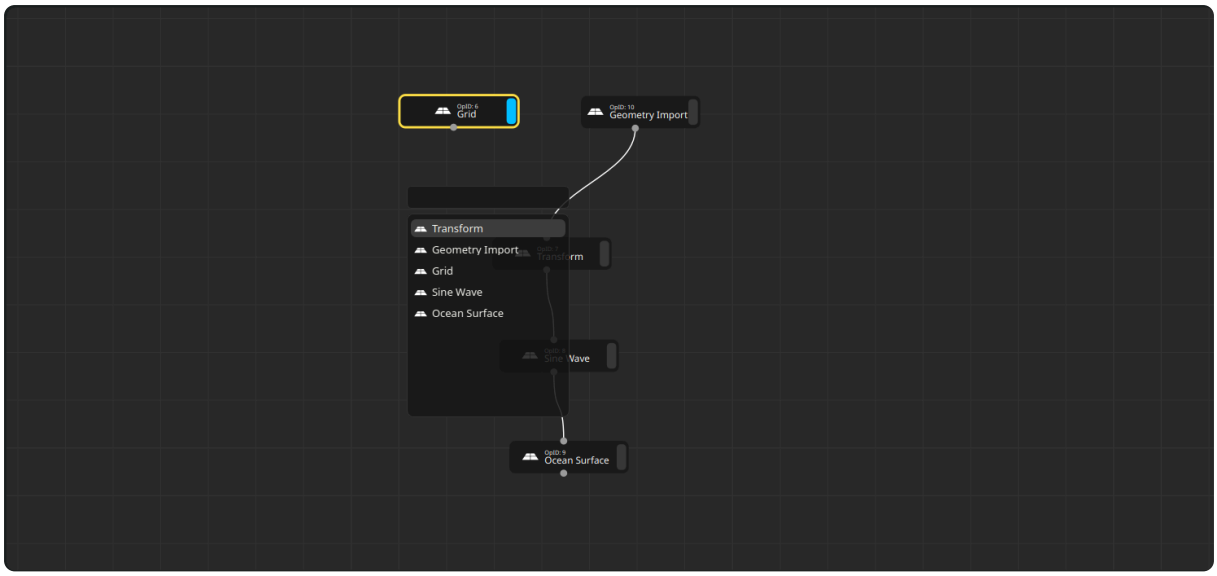
The network panel is the central control panel for modifying the network and nodes within. It provides a graph like view, visualizing operations as nodes and edges as the connections between them. The system provides tools for adding, deleting, and modifying nodes and connections. The panel's design was inspired from concepts designed by Dmitry Yershov on the Odforce Houdini forums[5]. The entire node system was written from scratch using Qt's QGraphics model view framework. Lots of time went into developing the visuals and user interaction of this system.



Node Creation Panel

`src/Gui/Network/TabMenu`

The "node creation panel" or "tab panel" is a special window that appears when the user presses the tab key. It displays all nodes registered as with the plugin system. For convenience it includes keyboard only navigation as well as a search bar to find specific nodes.



Viewport Panel

`src/Gui/Viewport`

The Viewport Panel provides a way for the user to visually view their model. It provides standard 3D controls for zoom, panning, and rotating. The viewport itself was written with OpenGL including classes for the camera, grid, and displayed mesh. The displayed mesh is automatically updated whenever a new change to the geometry is made and normals are computed automatically.

Geometry Spreadsheet

`src/Gui/GeometrySpreadsheetPanel`

The geometry spreadsheet provides a numerical way for the user to view their model and the underlying data stored within in a spreadsheet-like view. To be able to performantly handle anywhere from hundreds to millions of points, the interface is created using Qt's Model View

framework. When the view is changed, for example by the user's scrolling, it queries the custom implementation of QAbstractListModel which then gathers the corresponding data from the engine, updating only the section displayed on screen to the viewer. This proves to be extremely performant regardless of how large the mesh is. Icons at the top show the different owners available, point, vertex, primitive, and global. Clicking the icons changes which attributes are displayed.

Index	P.x	P.y	P.z
0	-0.166896	3.68572	2.01089
1	0.155688	3.69041	2.00962
2	-0.146725	3.38287	2.01678
3	-0.231545	3.41584	2.00906
4	-1.20753	4.62899	-0.461263
5	-1.01676	4.53041	-0.0247629
6	-1.28558	4.40031	-0.516576
7	-0.901531	4.2836	0.0391018
8	1.22361	4.56893	-0.275686
9	0.910953	4.53168	0.00821174
10	1.27917	4.43544	-0.526101
11	0.912614	4.27322	0.0355809
12	3.43344	-0.499164	-1.61646
13	3.69432	-0.819243	-1.8929
14	-0.891449	2.26023	-0.83787
15	-1.25946	4.68831	-2.07868
16	-1.05228	5.00296	-2.06942
17	-1.12744	4.98079	-2.15942
18	-0.904141	1.30119	-0.959869

Engine Design

Network Manger

`src/Engine/Network/NetworkManager`

The network manager is the central coordinator of the engine's node system. It manages the lifecycle of operators, including their creation, storage, and validation, while also tracking dependencies between them. Acting as a singleton, it ensures that all parts of the engine work with a single consistent view of the network, providing global access. Beyond just storing operators, it also controls cooking and traversing dependency graphs, ensuring that updates flow correctly through the network when nodes change.

The Network Manager also maintains interaction state, It tracks which node is currently displayed in the viewport, manages node selection, links each node to an operator ID, and broadcasts changes through signals. This event-driven design allows the UI, viewport, and other subsystems to react to network changes without being tightly coupled. It serves as the backbone of the engine.

Geometry

`src/Engine/Operator/Geometry`

The Geometry class represents the core data being passed between nodes in the engine. It is the operand of the network. Nodes read, modify, and output geometry, which is then used to populate the viewport and the geometry spreadsheet. Geometry is stored in an attribute based structure, similar to a spreadsheet, where each column is an attribute owned by points, vertices, primitives, or the global context. Most geometry information, including built in properties such as positions and connectivity, is stored through this same system. These built in attributes are intrinsic, but they share the same base as user-defined attributes to keep the data model consistent.

The class provides methods for constructing and querying geometry, including adding points and faces, computing normals, and building a CGAL half-edge mesh representation. It also exposes accessors for reading positions, primitive connectivity, and vertex data, while tracking special cases such as isolated points.

Attributes

`src/Engine/Operator/Attribute`

Attributes are the containers for geometry data in the engine. Each attribute is a single column in the spreadsheet that makes up your geometry. The column has a single name and type. Based on the attribute owner it will also have a value for each element. For example, if the owner is a point then your attribute might be 'Color', your type might be a vector, and the attribute will have a color value for each point in the mesh. Attributes own the underlying storage for their values, which is contiguous and strongly typed (e.g., ints, floats, vectors, bools). This allows a single attribute class to store data regardless of the type without templates. Attributes cannot be read or written from by themselves. For that, an attribute handle is required.

`src/Engine/Operator/AttributeHandle`

An Attribute Handle is a typed view into an attribute's storage. It binds at construction to a concrete type and exposes operations like reserving capacity, appending values, and reading/writing by index. Because the handle uses templating, most misuse is caught at compile time, and runtime guards raise errors if an attribute/handle type combination isn't accounted for. In the future implicit casting can be added for convenience. Handles don't own data, they just reference the attribute's storage.

There is also a read-only handle variant that provides the same typed accessors without mutation. This is useful when an operator needs to inspect data but must not modify it, when the engine exposes attributes to user code with limited permissions, or when implementing const member functions that require attribute access.

Together, attributes define the schema and storage, while handles provide the typed access that nodes and tools use to operate on data directly.

Operators

Dynamic Loading

`src/Engine/Operator/OperatorTable`

When designing the node creation system, it was essential to make it easily extensible without requiring users to recompile the entire software. This allows developers to build their own nodes for a precompiled version of the engine, while also significantly reducing compilation times compared to recompiling the entire engine with a node. If existing tools don't meet users' needs, they must have the flexibility to create their own. Which is especially important in community-driven open source projects. To achieve this, the system relies on dynamic runtime library loading via Boost DLL. Implementing this, however, posed challenges such as ABI mismatches, safely passing standard library types across different compilers and ABIs, and preventing per-library singletons.

These operators have their own CMakeLists, which compiles them into a .so or .dll library. At runtime, the engine searches for this file and loads it dynamically using boost::dll.

Operator Table

`src/Engine/Operator/OperatorTable`

The operator table is a central repository of static operator data. It's responsible for loading plugins, storing operator data, and providing an interface to access this data. When the user wishes to create a new node they open the tab menu which then accesses the operator table to find what operators

are loaded and available. When the user selects a node the tab menu requests the constructor from the operator table in order to create that node.

Registering Plugins

`src/Engine/Operator/OperatorTable`

New operator plugins can be added from the plugin using the newSopOperator function. This provides the function pointer to add an opInfo object for your new operator directly to the operator table. Here you provide it information about itself, such as its constructor, parameters, and inputs.

Operator Definition

`src/Engine/Operator/GeometryOpDef`

The operator definition is a base class from which new geometry operators are inherited from. It provides an abstracted interface, to read and write data about itself and the context it is being computed.

The class exposes utility functions for setting outputs and reading information about itself like the number of inputs.

The most important part of this node is the virtual cookOp member function. This must be overridden to implement the node's logic when being cooked. When a node is cooked it takes the optional input geometry from the context class and outputs new geometry based on the purpose of that operator.

Operator Context

`src/Engine/Operator/Context`

The context class is an interface for the cookOp function that provides important runtime context about the network. It allows querying parameters, reading input geometry, and in the future provides values like time.

Parameter Template

`src/Engine/Parameter/Template`

The parameter template is a compile time descriptor for a parameter's properties declared in the operator definition. It defines metadata like defaults, sizes, ranges, and type.

Nodes

`src/OpDefs`

Several basic nodes were created to demonstrate the platform and guide development such as Grid, Geometry Import, Transform, Ocean Surface, and Sine Wave. While I plan to add more nodes in the future to showcase the engine, and improve the tool, for the scope of this project this was all that time allowed.

Optimization

While release mode performance was acceptable, debug performance was not. In order to resolve this profiling an optimization was required.

Profiling

For performance profiling I used a combination of perf, hotspot, and flamegraph. This provided insight into what calls were taking the most time to execute. One of the main problem areas was the

naive querying of constant values in for loops. Each call added unnecessary overhead, which was multiplied by each point in the mesh.

Multithreading

After finding and fixing the obvious performance bottlenecks it was time to find other methods to speed up the program. Most nodes perform the same operation on each point, primitive, or vertex based on the already cooked input. This makes it a perfect candidate for parallel processing. Certain adaptations were required to avoid the race conditions brought with concurrency such as the locks for lazy evaluating certain geometry data.

Devops

Packaging

There are many packaging formats available, just to name a few there's tar.gz with runtime dependencies, RPM, DEB, Snap, Appimage, Flatpak, install script, and Nix Flake. Building packages for these usually requires an in depth exploration and understand of each format. Because of this Enzo uses CPack to build the packages, currently supporting RPM for Linux distributions in the RHEL family and tar.gz with runtime dependencies to target unknown systems. In the future it would be ideal to add support for other formats like Appimage, Flatpak, and DEB.

Cpack

Cpack was the ideal choice for Enzo as it's widely used, fits into the existing CMake build system, is relatively simple, and provides support for many generators. CPack generators are extensions of CPack that allow it to package for specific formats. For example the RPM generator builds .rpm files and the TGZ generator builds tar.gz files.

RPM

RPM is Red Hat's package management file format. It's used on Red Hat Enterprise Linux (RHEL) as well as other distributions in that family. This is a prime target for support as RHEL distributions have long been the standard for Animation and VFX. Packing an RPM file can be a time consuming process, but thankfully CPack's RPM generator makes it easy. When installing this way system dependencies can be automatically installed from the distributions repos.

TarGZ

Tar is an archiving format designed to combine multiple files into one, GZIP is designed to compress a single file. Together these serve to combine many files in to one compact, easily sharable file. This is a common format for distributing software usually consisting of a bin folder for binaries and lib folder for dynamically linked libraries. Packaging this way provides the advantage of portability and allows the user to use the software without installing it onto the system which would require certain privileges such as root access. Packaging this way, however, proved much more difficult. The first obstacle is collecting the runtime dependencies for a project, CMake provides tools which provide a baseline of dependencies. From there filters must be put in place to avoid installing unwanted system packages like glibc. Then additional rules must be created to install dependent files that CMake couldn't find like Qt's plugins. Finding all these requirements can be a very time consuming and unpleasant task. The next obstacle is glibc compatibility. The computer used to develop Enzo has a fairly new version of glibc, making the binary incompatible when used on computers with an older version. This required a docker container to be created for compilation. The image would install dependencies, build and package the project, and output it for use outside the container. For this Rocky 9 was chosen as it's stable, free, and RHEL based. Earlier versions of Rocky would provide compatibility with older platforms, however the dependency versions required weren't

available in the Rocky 8 repositories. Even then, building for a platform with older dependencies like Rocky 9 required several modifications to the source code for backwards compatability. Despite so many obstacles packing for TGZ is available in the CMakeLists file, however it is not recommended. Likely due to the older version of QT, at least one bug was introduced by this method of packaging.

Continuous Integration

To provide continuous integration for the project I run a self hosted Jenkins server. The server would regularly poll the SCM looking for changes in the source code. If a change is detected it will pull the repo, build, and run tests. The status of the last completed test will then automatically update on my custom written badge web API on the Github README. The badge API displays an SVG with the job's pass fail status. This can alert users or maintainers of failures in the main branch that need addressing.

Doxygen

In order to facilitate third party contributions a Doxygen webpage was created. Using a Github action and Github pages the webpage automatically updates when a change to the main branch is made. This keeps potential developers up to date with important code and documentation. The documentation is currently filtered to only include code for the Engine as this is what's required for contributing nodes.

Third Party Contributions

Efforts were made to involve friends in contributing to the project, as it would be exciting to get more experience developing collaboratively as well as lending itself to the open source nature of the project. Although most were unavailable due to the timing of the deadline, Owen was able to step in and help resolve a persistent bug. The issue involved certain normals flipping unexpectedly. Owen identified that the problem stemmed from normalizing too early his solution was to postpone normalization until after checking the cross product. This small adjustment proved to be the key fix.

Tests

Admittedly the tests leave more to be done. During early development keeping a solid test driven development cycle drastically slowed down development as interfaces and designs were constantly changing. Tests would have to be frequently rewritten to keep up with these changes. Because of this testing took a bit of a backseat until things became more stable.

Conclusion

In conclusion, Enzo represents the first steps toward a more open, accessible, and user-friendly approach to procedural 3D modelling. While its scope is necessarily limited compared to industry giants like Houdini, the project demonstrates a foundation, both in technical design and broader philosophy of openness and collaboration.

Though much work remains to be done, the progress made establishes a clear proof of concept, a node based system that is performant, flexible, and approachable. Enzo's value lies not only in what it can already achieve but also in the possibilities it opens for future development and community contribution. In this way, it offers a vision for what procedural modelling tools can be when designed with accessibility, transparency, and adaptability at their core.

Bibliography

[1] Autodesk, “Autodesk Maya | Get Prices & Buy Official Maya 2025 | Autodesk.” [Online]. Available: https://www.autodesk.com/uk/products/maya/overview

[2] Foundry, “Nuke Studio | Foundry.” Accessed: Aug. 18, 2025. [Online]. Available: https://www.foundry.com/products/nuke-studio#editions

[3] SideFX, “Houdini Store | SideFX.” [Online]. Available: https://www.sidefx.com/buy/

[4] O. S. Initiative, “The Open Source Definition.” [Online]. Available: https://opensource.org/osd

[5] D. Yershov, “Houdini UI needs an overhaul.” [Online]. Available: https://forums.odforce.net/topic/55385-houdini-ui-needs-an-overhaul

[6] O. Tutorial, “Particles / Instancing.” Accessed: Aug. 18, 2025. [Online]. Available: https://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

[7] M. Shah, “Generics in C++ 5 - Variadic arguments and Variadic Function Templates | Modern Cpp Series Ep. 75.” Accessed: Aug. 18, 2025. [Online]. Available: https://www.youtube.com/watch?v=irFkMavpL9A

[8] r., “Cast vector<int> to vector<unsigned int>.” [Online]. Available: https://stackoverflow.com/questions/76004849/cast-vectorint-to-vectorunsigned-int

[9] J. Rodríguez, “docs.gl.” [Online]. Available: https://docs.gl/

[10] C. Reference, “cppreference.com.” [Online]. Available: https://en.cppreference.com/

[11] Qt, “Qt Documentation | Home.” [Online]. Available: https://doc.qt.io/