A JIT Expression Language for Fast Manipulation of VDB Points and Volumes

Nick Avramoussis
DNEG
nna@dneg.com

Richard Jones
DNEG
rhj@dneg.com

Francisco Gochez
DNEG
fjg@dneg.com

Todd Keeler
DNEG
tdk@dneg.com

Matt Warner
DNEG
mw@dneg.com

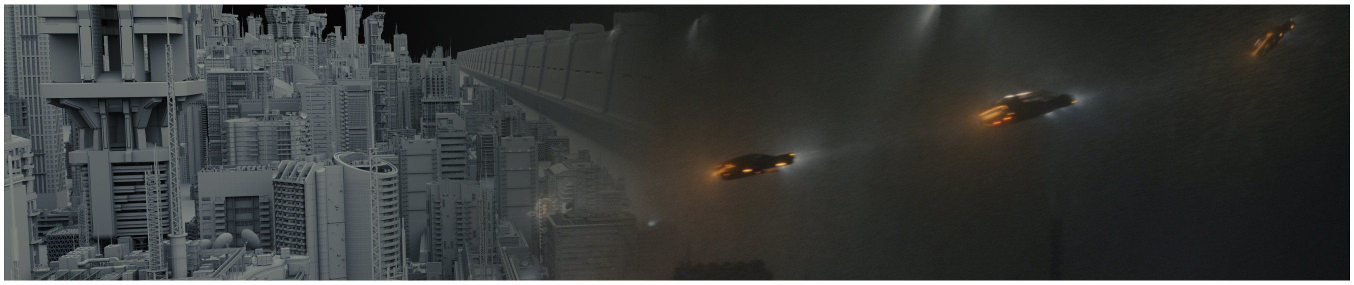


Figure 1: This expression language has been incorporated into our in-house scattering tools in Clarisse, built on VDB points. Together these create a flexible workflow for handling massive amounts of geometry, e.g. the city of LA in Blade Runner 2049[*].

ABSTRACT

Almost all modern digital content creation (DCC) applications used throughout visual effects (VFX) pipelines provide a scripting or programming interface. This useful feature gives users the freedom to create and manipulate assets in bespoke ways, providing a powerful and customizable tool for working within the software. It is particularly useful for working with geometry, a process heavily involved in modelling, effects and animation tasks. However, most widely available examples of these are either confined to their host application or ill-suited to computationally demanding operations. We have created an efficient programming interface built around the open-source geometry format, OpenVDB, to allow fast geometry manipulation whilst offering the required portability for use anywhere in the VFX pipeline.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; **Graphics file formats**; *Modeling and simulation*;

ACM ISBN 978-1-4503-5895-8/18/08…$15.00
https://doi.org/10.1145/3233085.3233087

KEYWORDS

expression language, just-in-time, geometry manipulation

ACM Reference format:
Nick Avramoussis, Richard Jones, Francisco Gochez, Todd Keeler, and Matt Warner. 2018. A JIT Expression Language for Fast Manipulation of VDB Points and Volumes. In *Proceedings of The Digital Production Symposium, Vancouver, BC, Canada, August 11, 2018 (DigiPro '18),* 4 pages.
https://doi.org/10.1145/3233085.3233087

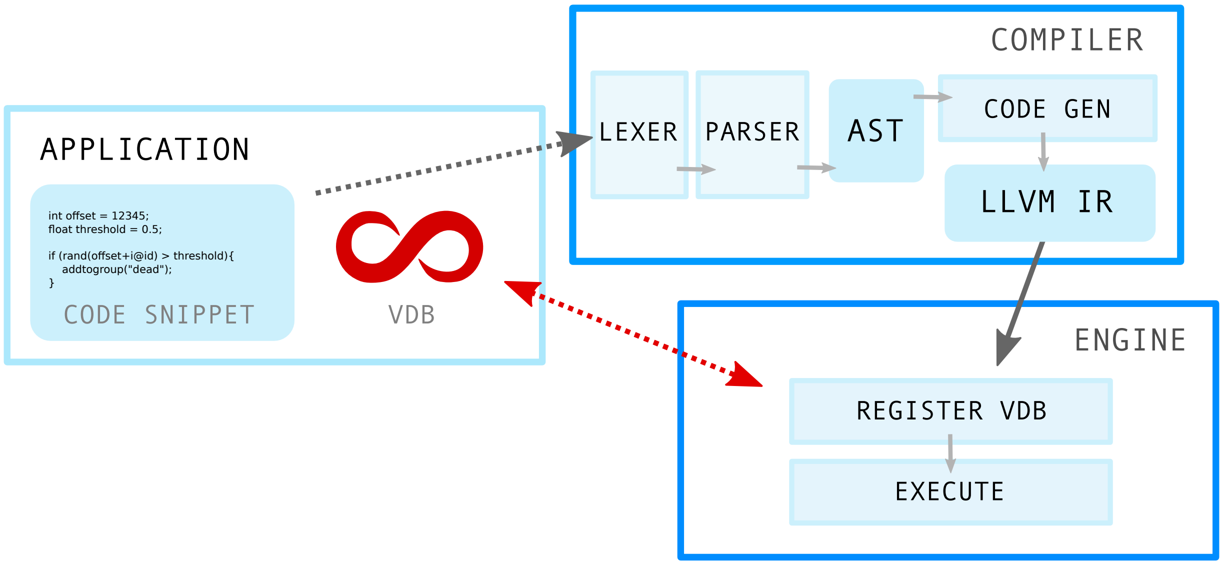
1 INTRODUCTION

The success of Alembic [Sony Pictures Imageworks 2018a], OpenVDB [Dreamworks Animation 2018], USD [Pixar Animation Studios 2018] and other open-source geometry and scene representation formats is largely due to the ability to use them in all parts of the VFX pipeline. Their open-source nature allows integration into almost any parent application and, as such, caters to the ever-changing needs of a modern VFX studio.

OpenVDB in particular offers an efficient, portable and flexible data structure for the storage of volumetric data - qualities that lead to adoption at DNEG and our development of an extension to incorporate point data. With its heavy integration into our pipeline, it quickly became apparent that exposure of a unifying interface would be the perfect tool to maximize our ability to take advantage of this portability. Whilst OpenVDB does include some Python bindings, we found that a more efficient and specialised approach was necessary to expose the finer-grained control we desired.

To tackle this, we have developed a new expression interface for the direct manipulation of point and voxel data in VDBs that is:

[*]©2017 ALCON ENTERTAINMENT, LLC., WARNER BROS. ENTERTAINMENT INC. AND COLUMBIA PICTURES INDUSTRIES, INC. ALL RIGHTS RESERVED.



```
int offset = 12345;
float threshold = 0.5;

if (rand(offset+i@id) > threshold){
    addtogroup("dead");
}
```

Figure 2: The structure of this tool and how it integrates with a chosen application, broken down into the two main components, the compiler and the execution engine.

- Fast - speed comparable to a compiled, multi-threaded plugin
- Portable - transferrable across DCCs and applications
- Easy-to-use - requires minimal programming experience

This has since been used to provide greater artistic control throughout the pipeline at DNEG with exposure in Houdini [Side Effects Software Inc. 2018], Clarisse [Isotropix 2018] and from the command line. The flexibility it provides has been particularly useful on recent shows such as Blade Runner 2049 (Fig. 1) and Pacific Rim: Uprising as part of our in-house scattering and point toolset.

2 PORTABLE LANGUAGE DEVELOPMENT

The integration of generic scripting languages such as Python, Lua etc. into DCCs offers users of different applications a familiar interface whilst exposing custom application and task-specific functionality. However, they are commonly unable to provide the performance requirements to operate on heavy data structures such as representations of geometry. A custom operation on billions of points, vertices or voxels requires lower level instruction mapping and a good multi-threaded framework to achieve practical results. Such tasks instead tend towards compiled methods such as C++ plugins, but these bring with them the issues of portability, complexity and extensibility - with limits on the amount of functionality that can be practically exposed to the user.

To overcome these limitations we looked to the use of just-in-time (JIT) compilation, an approach that has been particularly successful when coupled with a custom and task-specific frontend. Shader languages, such as OSL [Sony Pictures Imageworks 2018b] and Houdini's VEX language are good examples of this. These are often exposed much like scripting interfaces, but their compilation and targeted design results in far more efficient execution, making them suitable for geometry manipulation. Whilst extremely powerful, we encountered limitations with these existing solutions such as lacking flexibility for supporting new geometry types or ill-suited frameworks for more generic geometric operations (e.g. the creation and deletion of data). Therefore, following a similar design we chose to develop a new language, combining JIT compilation through LLVM [LLVM Developer Group 2018] with OpenVDB geometry. The result provides a transferrable interface for fast manipulation of production assets that can be easily integrated into many target applications.

2.1 Task-Specific and Parallel By Design

Shader languages offer a nice solution to the boilerplate code required with other less task-specific languages, in our case allowing an easy element-centric expression interface that can be efficiently parallelised. Our design focuses on the manipulation of the 'lowest' element in a VDB tree, i.e. a point or voxel, executing the user-supplied expression over each element independently. This iteration process is 'embarassingly parallel' and when coupled with our highly-optimised, JIT-compiled functions is extremely efficient and capable of handling very large datasets.

3 IMPLEMENTATION

The implementation can be broken down into two main components (demonstrated in Fig 2): the LLVM function generation that takes an input expression and JIT compiles it into a custom function to be run on each geometric element; and the OpenVDB component that registers access to the supplied VDBs and subsequently performs execution over them.

3.1 LLVM function generation

The use of LLVM allows us to generate custom compiled functions from our expression language, and comes with built-in support for a number of optimisation and validation passes. As shown in Fig. 2, we begin by parsing an input code snippet to create an abstract syntax tree (AST), a representation of the syntactic constructs that occur in the supplied code. This is traversed by our code generation framework which converts each node of the AST into LLVM's intermediate-representation (IR). Finally, the resulting IR is JIT-compiled and handed to our engine for later execution. The parsing, IR generation and JIT-compilation are very fast operations and due to the intensive optimisation passes offered by LLVM, the resulting functions are of comparable speed to ahead-of-time compiled C++ code (Table 1). We also make sure to decouple this process from any particular input VDB, therefore only requiring a single compilation pass for any number of inputs. Instead, each unique input simply undergoes a fast registration step before execution to expose the correct data to the compiled function.

3.2 OpenVDB integration

In our expression interface we allow read-and-write access to OpenVDB point attributes and voxel values. Our OpenVDB bindings give our compiled functions direct access to the values within any supplied VDB trees. To do so, on execution we bind handles to point attributes/volumes and store them on our execution engine. To ensure we only require a single code generation pass for any number of VDBs we do not compile these directly into IR. Instead we insert IR instructions to retrieve these handles from the engine at runtime[1].

For multi-threaded execution we leverage OpenVDB's node structure [Museth 2013]. In this way, we parallelise over each node in the supplied VDB trees, executing our function on each element (point/voxel) they contain. This execution pattern fits intuitively into OpenVDB's data structure, inherently load-balancing over its spatially-organised nodes.

3.3 Available operations

Attribute expressions are written in a simple C-like language with many of the usual syntactic constructs e.g. conditionals, function calls, binary operators etc. (Figs. 3 & 4). We incorporate element accesses through a specific identifier, @ (inspired by Houdini's VEX language), to easily differentiate from local variable usage. The function calls we provide can cover a large amount of functionality, from basic mathematical operations, noise and random number generation to element-specific (and geometry-specific) behaviour such as collecting points into groups.

Whilst currently each element only has access to its own attributes/values, it should be possible to extend to allow access to other elements in future. This could allow even more complex operations like smoothing surfaces or accessing nearby points for neighbourhood operations.

4 PRODUCTION IMPACT

The motivation for this project arose during the development of OpenVDB Points at DNEG [Museth et al. 2015]. Although originally designed as a particle framework for simulation, it was primarily used for data interchange due to a lack of frontend tooling. The flexibility offered by alternative point formats (e.g. in third-party DCCs) with more comprehensive toolsets raised further concerns over adoption on a wider scale. To succeed as a toolkit that users could directly interact with, a fast way to expose a lot of custom and controllable functionality became critical. Following previous successes with LLVM [Bailey et al. 2011] and recognising the flexibility arising from geometry shader languages such as VEX, this tool was conceived as a solution to this problem. Once this custom expression interface was exposed for OpenVDB Points, we saw a huge increase in interest in the format as a whole. Then, due to the nature of the integration into OpenVDB, and the similarities between its volume and point storage, it was relatively straightforward to extend this framework to provide manipulation of volumetric data as well. VDBs are now the standard for storage of volumetric and point data at DNEG, with the ability to manipulate them directly and deterministically throughout the pipeline proving to be a significant asset.

[1]A similar method is used for other arbitrary external data e.g. time or frame number, allowing use of variable data without requiring re-compilation.

5 EXAMPLE USE

In the following, we discuss a couple of different use-cases for this tool and demonstrate the simplicity of the code required to perform such tasks. First we consider an example for FX simulation and then, further downstream in the pipeline, manipulation of FX data in Lighting.

```
// get timestep
float dt = 1.0f / (4.0f * 24.0f);
// gravity
vector gravity = {0.0f, -9.81f, 0.0f};
// drag
vector dV= {2.0f, 0.0f, 0.0f} - v@v;
float lengthV = length(dV);

float Re = lengthV * @rad / 1.225f;
float C = 0.0f;
if (Re > 1000.0f) C = 24.0f / Re;
else C = 0.424f;
// calculate drag force
vector drag = 0.5f * 1.2f *
        C * lengthV * deltaV * 4.0f * 3.14f *
        pow(@rad, 2.0f);
// update velocity
v@v += (gravity -
    drag / ((4.0f / 3.0f)
    * 3.14f * pow(@rad, 3.0f)) * dt;
// update position
v@P += v@v * dt;
```

Figure 3: A particle simulation step using gravity and drag against a constant wind force.

5.1 Example: Simulation

The flexibility given by this expression interface can be used to very quickly create operators such as those required in simple particle or volumetric simulations. For example, consider a particle system acting with respect to a collection of independent motions or driven by external influences - we are able to very easily create this kind of operator using our expression interface as in Fig. 3. As these operators are often executed many times in sequence, performance is extremely important. We show in Table. 1, the performance of our JIT-compiled expression matches very closely to that of an equivalent operator written in C++.

```
int offset = 12345;
float threshold = 0.5;
// remove points
if (rand(offset+i@id) > threshold) {
    addtogroup("dead");
}
```

Figure 4: Decimating points in a set using a percentage threshold.

5.2 Example: FX to Lighting

Having the ability to modify geometry in this way is important in areas beyond FX simulation. However, making modifications or tweaks to geometry is not always easy outside of packages designed explicitly for it, for example in rendering applications such as Clarisse. Integrating our expression interface into such applications facilitates on-the-fly modifications without requiring costly back-and-forth between FX artists and lighters.

This level of control has seen great use in this context at DNEG, with lighters able to perform many tasks without having to send data back up the pipeline. Some examples we have seen being used include:

- Randomizing scattering ids for instancing.
- Modifying colours, orientation, scales and velocities of scattered point data.
- Decimating excessively large point sets.

Sample code for this latter example can be found in Fig. 4.

Table 1: Performance of C++ implementations vs our JIT-compiled expression examples running on 32 core Intel Xeon 3.10Ghz CPU with 64GB RAM.

Code Example	# elements	C++	JIT	Performance
Fig. 3	50m points	0.43s	0.58s	0.74x
Fig. 4	50m points	2.69s	2.70s	0.99x

6 CONCLUSION

This tool has shown to have production impact beyond our original expectations. We have found that by exposing this interface in multiple applications (e.g. Houdini, Clarisse, command-line) we have been able to speed up both user and developer workflows, allowing on-the-fly edits to assets and exposing access to the data in ways that can also be used in the creation of more advanced tools. Whilst the language and the design itself follow other tools before it, the portability arising from pairing to an open-source technology such as OpenVDB has been invaluable to its success. This further demonstrates the importance of transferrable technologies in a VFX pipeline built upon so many different applications. Interestingly however, the decoupled nature of our compiler and execution engine suggest that such a tool could also in the future be extended to support other types of geometry beyond VDBs.

ACKNOWLEDGMENTS

The authors would like to thank Dan Bailey, Harry Biddle and James Bird for their important work on the original development of this tool and their great contributions to the OpenVDB Points framework. We would also like to thank Khang Ngo for his work on the Clarisse scattering toolkit at DNEG, and finally, Ken Museth and those at Dreamworks for open-sourcing OpenVDB.

REFERENCES

Dan Bailey, Ian Masters, and Matt Warner. 2011. GPU Fluids in Production: A Compiler Approach to Parallelism. In *ACM SIGGRAPH 2011 Talks (SIGGRAPH '11)*.
Dreamworks Animation. 2018. OpenVDB. (2018). http://openvdb.org/
Isotropix. 2018. Clarisse. (2018). https://isotropix.com/
LLVM Developer Group. 2018. LLVM. (2018). https://llvm.org/
Ken Museth. 2013. VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.* 32, 3, Article 27 (July 2013).
Ken Museth, Dan Bailey, Jeff Budsberg, John Lynch, and Andrew Pearce. 2015. OpenVDB. In *ACM SIGGRAPH 2015 Courses (SIGGRAPH '15)*.
Pixar Animation Studios. 2018. Universal Scene Description. (2018). https://graphics.pixar.com/
Side Effects Software Inc. 2018. Houdini. (2018). https://sidefx.com/
Sony Pictures Imageworks. 2018a. Alembic. (2018). http://opensource.imageworks.com/
Sony Pictures Imageworks. 2018b. Open Shading Language. (2018). http://opensource.imageworks.com/