An Interface for Real-Time Networked Streaming of 3D Scenes

Daniel Green

K0099719

School of Computing
Teesside University
Middlesbrough TS1 3BA

Submitted in partial requirements for the degree of

MA Computer Animation and Visual Effects

Supervisors:
Dr. Paul Noble
Dr. Matthew Holton

August 2016

Dedicated to the curious and inquisitive, passionately working hard today to make for a better tomorrow. You are the future.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

To be explicitly clear, the works submitted to be marked are listed as follows: Pre-development and post-development survey, including all content created for them; Thesis and Implementation document; Source code and binaries at the time of submission for the developed *Atom* plugin, and a proof-of-concept client implementation; Custom documentation for the *Atom* plugin; Poster display.

Daniel Green
August 2016

Acknowledgements

I would like to express gratitude towards all those who have aided me throughout my journey, not only during, but before my studies. An extended thank you to the following persons for their contributions.

Michael and Irene Shepherd, my grandparents, for supporting me during my studies.

Dr. Paul Noble, a supervisor and friend, for being genuinely interested in what I do and pushing me to overcome ambitious tasks.

Ken Yoshimura, for all of the compelling exchanges about space, and for inspiring me to pursue technical art. Don't even stop making awesome things!

Additionally, a warm thank you to those who completed one or both of the surveys from this project, driving development and providing valuable feedback. Of those willing to be named, in alphabetical order: *Luke Bourner*, *Zachary Conlyn*, *Adam Delaney*, *Tom Elliot*, *Jef Fleurkens*, *Ross Furmidge*, *Alex Gifford*, *Arnaud Halbardier*, *Piotr Hurny*, *Nick Jackson*, *Chiwoon Leow*, *James Lowrey*, *Malcolm McKneely*, *Masahiko Murakami*, *Ruthie Nielsen*, *Craig Richardson*, *Ste Seress-Smith*, and last but not least, *Mengna Zhu*.

Abstract

The video games industry is actively tackling the problem of pipeline efficiency. One outstanding issue is mitigating the steps required for artists to preview 3D models with in-game visuals during development, to increase workflow and save time. The developed project that this thesis details - *Atom* - is an attempt to target this problem through networked streaming of 3D scenes, with the aim of unobtrusively providing remote previews to artists. For instance, previewing a Maya scene in real-time in a client running on a tablet device.

Chapters will sequentially introduce the topic and provide analysis of existing solutions, making a case for the need of the project. A high-level description of the developed project will be explored to provide an overview of its structure. A highly detailed discussion of its intricacies will follow. Finally, reflection on the project's success and third-party opinions on the topic will be discussed and analyzed.

Overall, *Atom* was largely successful, tackling an ongoing issue within the video games industry. Although not without flaws, it does provide a valuable solid foundation to build further research upon.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As time progresses, the use of 3D workflows is ever increasing, becoming an integral part of any pipeline. Special effects are bested by simulations. 2D animation jumps up a dimension to join the 3D realm. Video games, film, and television, among others, all bring artistic vision to life increasingly with the use of 3D.

3D software packages have advanced rapidly side-by-side the amelioration of both CPUs and GPUs. Not only have they become more powerful, but also more accessible to meet ever increasing demand. In the past, even simple renders were too expensive, leaving artists largely guessing on a visual front. Most software packages today support advanced rendering in the viewport while artists create. This increases productivity and quality by orders of magnitude, as at any point, artists can preview how a scene may appear in its final quality. Select software packages include support for external interactive preview rendering, which can display previews iteratively increasing in quality separate from the software package's core editing.

Offline rendering pipelines, such as those found in film and television, are often tightly integrated with the 3D software packages themselves. For instance, a scene created in a software package is normally rendered in the same package. If an external renderer is instead used, first-party integration usually eases this process. In these cases, render time, whilst important, is not a major concerning factor. There are very few steps required for an artist to preview their work in its final quality, even during development.

In video games, however, this process is not as streamlined. The number of steps an artist must take to preview their work with final in-game visuals is staggering. Supporting multiple platforms complicates matters further, with each platform requiring unique steps. The visual output of the 3D scene is not from the software package itself, but instead from an independent game engine running in real-time. This rules out traditional in-software

rendering for previews, as the visual quality is different, and time is of the essence. Solutions have been developed to mirror offline previewing techniques, but each have their drawbacks.

This raises the question of how these 3D scenes can be previewed external from the 3D software package in question, such that artists can visualize their scenes with in-game quality in real-time, and be able to make edits to them, without having to go through tedious exporting pipelines each time a change is made.

This project aims to tackle this question with the creation of an open interface into Autodesk Maya, targeted at video games pipelines, allowing for developers to selectively stream in data and modifications from Maya in real-time over networked connections. For instance, a game engine could receive the streaming data as changes are made, and display them for artists to see their modifications live, bypassing obnoxious exporting steps every single time a preview is desired.

Chapter 2

Background Research

Existing Works

As games continue to improve, artistic workflows must adapt to become more streamlined and less intrusive. A typical asset pipeline, as in Figure 2.1, has a recursive nature. Assets start their life in a Digital Content Creation tool, or DCC, such as *Autodesk Maya* (Autodesk, 2016a). Assets are then configured - being placed by level designers, for instance - and individual assets are tweaked. Assembled levels are then lit. Complete levels are then imported into the game. Configuration, lighting, and importing require significant rework and artists will find themselves traversing between these stages often. The time taken on these repetitive steps could be better spent elsewhere. The student project by Labschütz, Matthias et al. (2011) demonstrates this pipeline in practice, including the recurrent steps discussed.

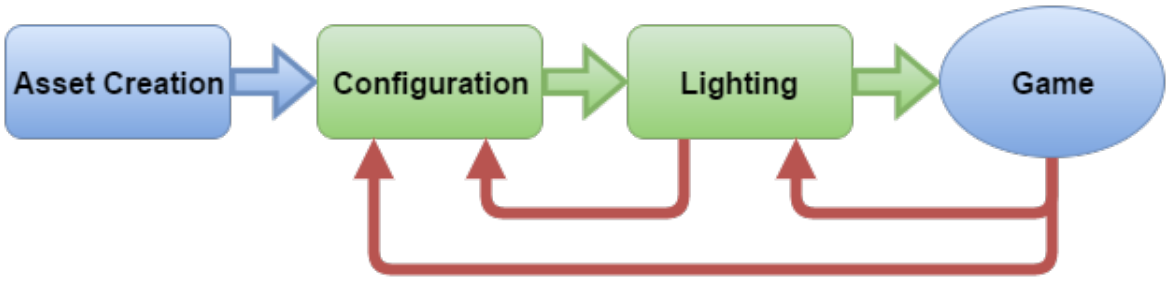


Fig. 2.1 Generic, high-level asset pipeline.

Although pipelines differ, it is widely accepted that the stages required for previewing artwork in-game during development are a hindrance to workflow efficiency. Rémi, Arnaud (2010) analyzes in detail the asset pipeline, expressing the need for robustness and flexibility, and reveals various approaches to speed up previewing, such as hot-loading exported assets

in-game when changes are saved out from a DCC. The suggested approaches help, but do not provide real-time feedback to artists.

Guerrilla Games (Giordana, Francesco, 2011) implemented a multitude of solutions to this. Jan-Bart van Beek et al. (2011) and Sander van der Steen (2014) further detailed the creation and effect of these solutions on the development process. The first being a custom hardware shader node for Maya that overrides shading graphs, building custom CG shader code for real-time previews. This was limited to eight light sources, and did not correctly reflect that of the game's visual standard. Second, the core deferred renderer of their game engine was directly implemented into Maya as a viewport renderer, giving real-time previews true to the game, with strong integration with Maya's native rendering pipeline. This removed the light limitation and worked very well. Finally, a live connection between Maya and a local instance of the game was created, for real-time updating and previewing in the actual game itself.

A recent alternative method of increasing artistic productivity comes from the tight integration of *Houdini Engine* (SideFX, 2016) into other DCC tools, including game engines, such as *Unreal Engine 4* (Epic Games, Inc., 2016) and *Unity* (Unity Technologies, 2016). While this does not allow for previsualization of assets a la previously discussed works, it does successfully reduce pipeline stages for iterative development cycles artists commonly struggle against.

In addition, DCC tools have recently been including the ability to export directly into game engines, such as Unity and Unreal Engine 4. Autodesk Maya, for instance, supports both of these. Although functional, they are essentially glorified FBX exporters, still requiring exporting and manual interaction each time a change is made. All known implementations lack real-time or remote support, too.

All of the above, while productive, share a common flaw: visualizations are localized. If these approaches were instead remote, many new possibilities open. It would be very convenient if an art director could have a level displayed on a tablet device while artists modify the scene from a workstation elsewhere, for example. This raises the question of how we can bypass, or at least mitigate, slow artistic pipelines for efficient remote real-time visualization, even with manipulation, of 3D scenes from a DCC.

Proposed Solution

The proposed solution to this problem, of which the rest of this thesis will discuss at length, is the creation of an interface into Autodesk Maya to allow for real-time streaming of scene data over networked connections. The core product will be a plugin for Maya that acts as a

server, streaming data to networked clients. Clients will be able to interact with the plugin through a networked interface, with two-way communications akin to a chat service. A custom game engine, for instance, could act as a client, communicating with the proposed API over a networked connection to stream in scene data to the engine for real-time previews, even while editing. The plugin would be developed such that it is completely standalone, without knowledge of any pre-existing clients, so that it is not restricted to specific third-party client applications.

Pre-Development Survey

A short survey was distributed before development started to gain third-party feedback on planned features and goals. The purpose of the survey was to discover the most common use-case scenario in real-world application, find out what components of the proposed product were considered most important, and to gather opinions on how to shape the public-facing portion of the product. A total of 20 participants completed the survey. Graphs of the results can be found in Appendix A. The results were gathered from a multitude of disciplines. Programmers within the video games industry were the majority (Figures A.7 and A.8). The end-product is targeted at exactly this demographic, which is ideal.

An overwhelming 95% of participants declared the project practical (Figure A.1).

Speed of data transfer, regardless of the participant's field, was considered a significant factor to success by a large margin, and will therefore be taken into consideration when designing data interchange formats (Figure A.2).

Ease-of-use of the proposed API was tied between a somewhat and very important factor (Figure A.3). As a result, usability will be considered, but not at the expense of performance.

The transferring of data is the core of the project. Geometric and material data were rated as most important, with lighting and rigging following (Figure A.4). Cameras and curves were likewise popular, albeit lower. A suggestion from a participant was to include both vertex color and animation data, too. These results will be considered during design of data structures.

Given the option of generic data with a simplified API, versus customizable data at the expense of API complexity, 65%, a majority of programmers, think that customizable data is better (Figure A.5). Since the API is targeted towards this demographic, customizability of data will be acknowledged when designing the API interface.

Presented with three use-case scenarios, the most popular would be to use the tool to make large-scale edits, ranging from geometric manipulations to adding and removing objects

(Figure A.6). The project will need to handle significantly more cases than if it were a static preview tool, and therefore the majority use-case will be kept in mind during development.

Additional Works

At the very end of the development of this project and thesis, Autodesk released an updated version of their *Stingray* (Autodesk, 2016b) game engine with support for real-time streaming from Maya. While this is in a similar vein to the proposed product of this thesis, it is not in the fashion of an API and is limited to a specific application, and therefore the need for such a project still exists.

Chapter 3

High-Level Design

The final product is a plugin, named *Atom*, for Autodesk Maya that acts as an interface into scene data via a networked methodology similar the observer pattern. A client, both for development and documentation purposes, was too created. As displayed in Figure 3.1, the plugin - Atom - receives information directly from Maya. The information is then forwarded from within the plugin over networked connections to connected clients. An important observation from this diagram is the disconnect between the plugin and the clients. Both the plugin and potential clients have no knowledge of each other. The intermediate stage - networked transfer of scene data - is responsible for allowing both to communicate in an abstracted manner.

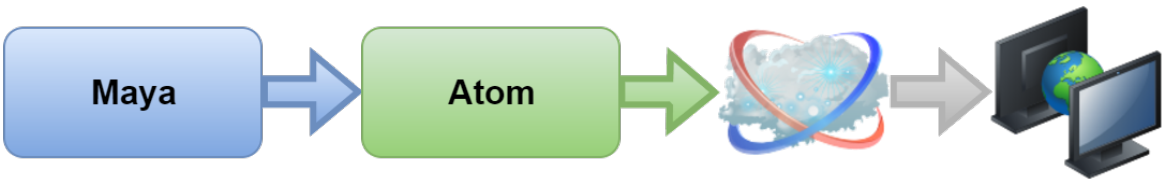


Fig. 3.1 High level structure of Atom's operation.

To demonstrate flow, an example use-case scenario will be briefly analyzed. In Figure 3.2, a prototype client (left) has a live networked connection to an instance of Maya (right). The scene loaded in Maya is a fully textured 3D model. Once Atom is loaded within Maya, the Directed Acyclic Graph (DAG) is mirrored and monitored for modifications using a system that has become to be named *AtomDag*. This will be discussed at length later in this thesis. A multi-client server, which will likewise be investigated later, is also started, which provides the two-way communication layer for the public-facing networked interface into Maya. Clients can relay information back to Atom's server using a command-based interface, i.e. `command <args>;`. When clients connect, they are sent a simplified scene description

with only object names and types. Clients are then free to choose which objects they would like to monitor by sending a `watch` command to the server with the object in question's name. Upon Atom receiving a watch request, a live link between the respective client and the object are set up, such that scene manipulations including the object are forwarded over the network in subsequent messages. At any time, a client can disregard watched objects using the `unwatch` command. Upon a client requesting to watch an object, the initial state of the object is sent. Any further modifications are internally handled by the AtomDag system, which eventually forwards any changes made within a set interval.

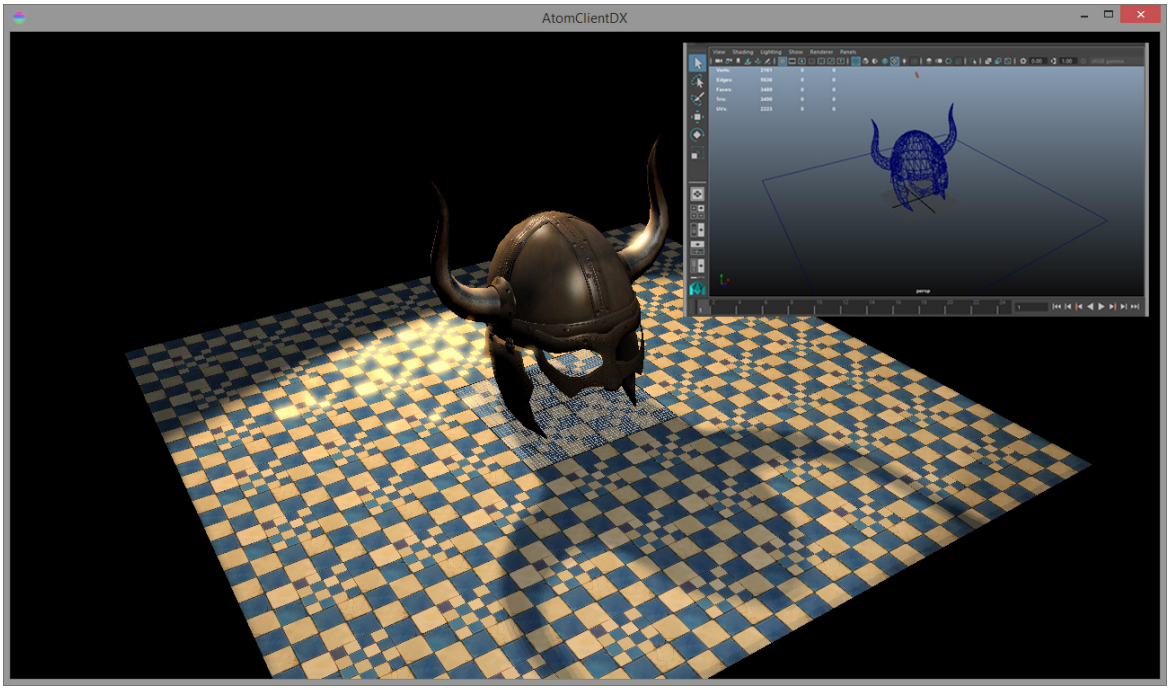


Fig. 3.2 A simple textured game-ready model displayed in a networked client using Atom.

The server component of the plugin is responsible for the interaction between connected clients and the Maya scene. It is a thread-safe, multi-client configuration. Even when constantly overwhelmed with requests from multiple threads, it is safe and stable. Its implementation is based upon *Asio* (Boost, 2016) due to its superior asynchronous capabilities and widespread usage. An observer system is used for external processing of received data. A simplified diagram of client and server interaction can be seen in Figure 3.3. When the server is started, it continuously listens for new connection requests. Upon accepting a new request, a link - `Connection` - is made, and an asynchronous read loop is started. When data is sent from a remote client, this read loop will compile all of the incoming data, and upon completion, notify observing systems about the message. Connections can process, from multiple threads at the same time, the request to send data to the remote client.

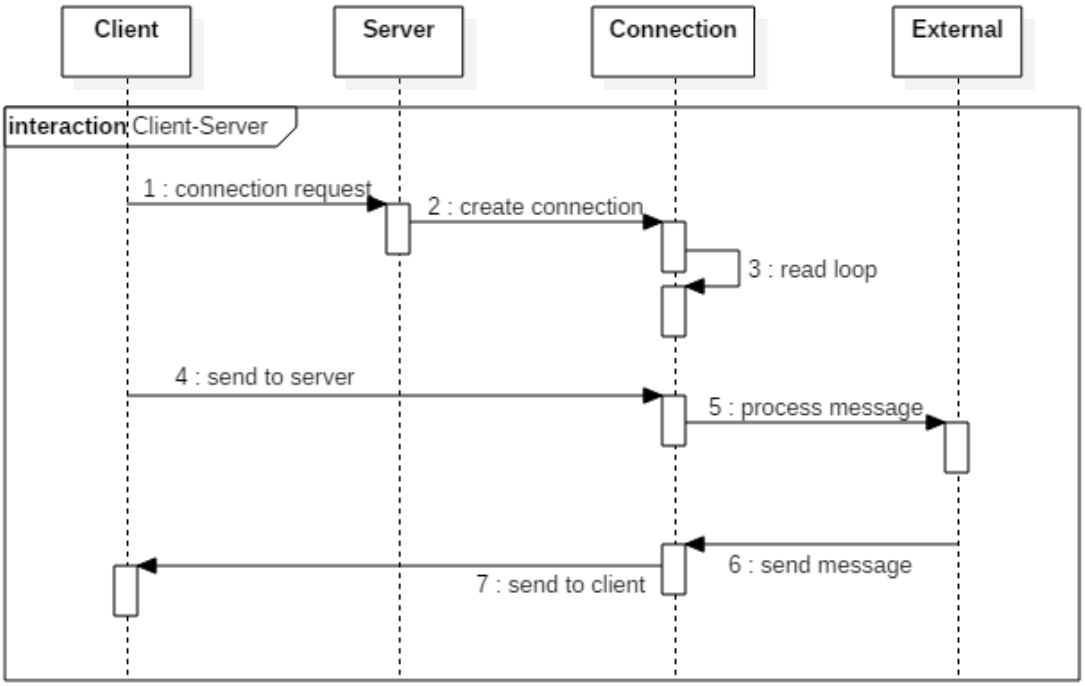


Fig. 3.3 Simplified client/server interaction.

It would be obnoxious and inconvenient for programmers implementing clients to have to deal with networked messages that were raw without structure. Parsing a multitude of formats from an arbitrary byte array is not a simple task. Instead, it would be desirable to have some kind of common interchange format that is easy to use, without sacrificing performance, irrespective of the programming language used. This project takes advantage of Google's *Protocol Buffers* (Google Inc., 2016) library. Protocol Buffers allow for language-neutral and platform-neutral serialization of structured data. They can be compressed to and rebuilt from raw byte arrays, which makes them ideal for transferring large amounts of complex structured data over networks. Not only that, but the source Proto files double as documentation. Messages sent by Atom are all serialized using Protocol Buffers. An advantage of using Protocol Buffers for every message is that there need not be any distinction between message types when rebuilding from networked data. Instead, a message from Atom consists of but a 32-bit signed integer denoting the size of the data, followed by the raw data itself. This data can then be used by Protocol Buffers to rebuild a single base message type - `AtomMessage`. Protocol Buffers contains a feature, namely `oneof`, which allows for a message to contain many possible data entries, but only for a single one to be actively included; all others are disregarded. This both saves memory and makes way for an

easy-to-use API with but a single major message type. The messages Atom provides will be investigated later in this thesis. Protocol Buffers is also hyper optimized, which is ideal for real-time applications, especially those that are networked.

Figure 3.4 shows a simplified model of how messages are sent and received. For sending, a Protocol Buffer class is created and the `Connection` is asked to queue it for sending. An outgoing `Message` is created and populated with the Protocol Buffer data, and is then sent over the network to the remote client. Similarly for receiving, an incoming `Message` is created and populated with networked data, which is then processed external to the `Connection` by observers.

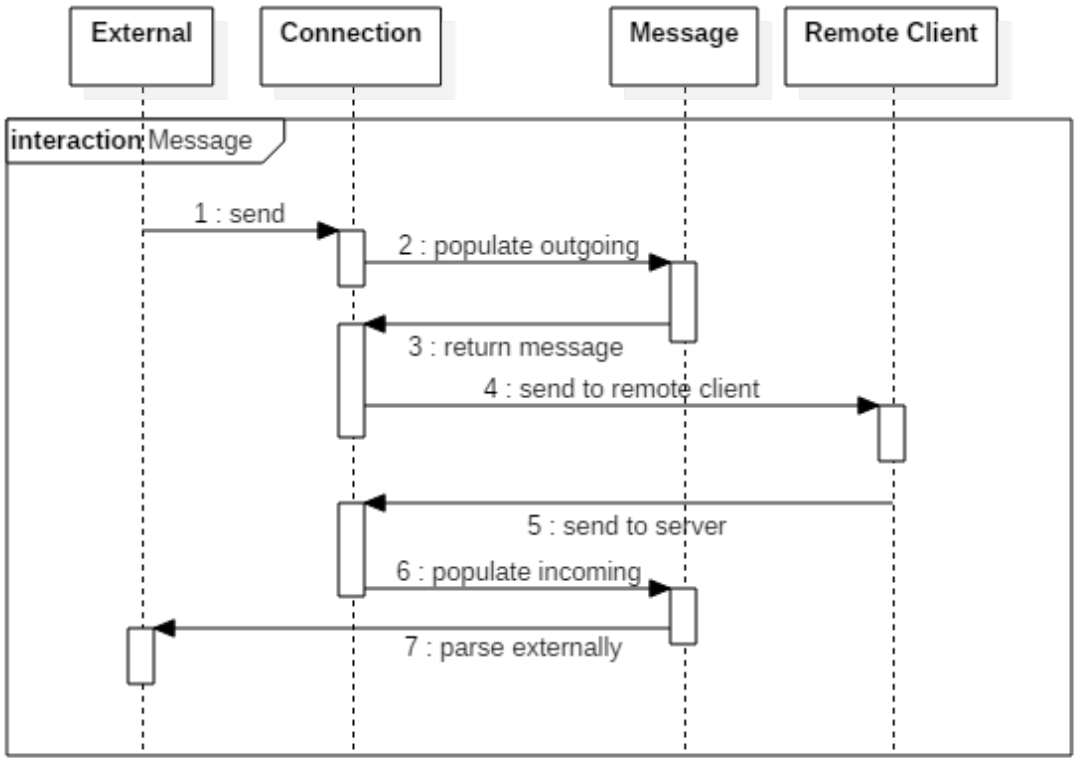


Fig. 3.4 Simplified sending and receiving of messages.

The aforementioned AtomDag system is a core component of the Atom plugin. AtomDag shadows Maya's internal DAG to monitor, manage, and propagate changes in the hierarchy and the nodes within. Internally, AtomDag parses Maya's scene, and then responds to events such as parenting, nodes being added or removed, and so on, using Maya API's callback system, e.g. `MDGMessage::addNodeAddedCallback`. Similar to the server, AtomDag is

populated with numerous callbacks so that external systems can respond to messages caused by Maya events.

Nodes themselves are handled and processed based on their internal Maya type. A listing of all AtomDag node types is displayed in Figure 3.5. The base type - `Node` - contains all essential data regardless of type, such as the name, underlying `MObject` and `MDagPath`, parent, and so on. The base type also contains another watching system, which will be discussed shortly.

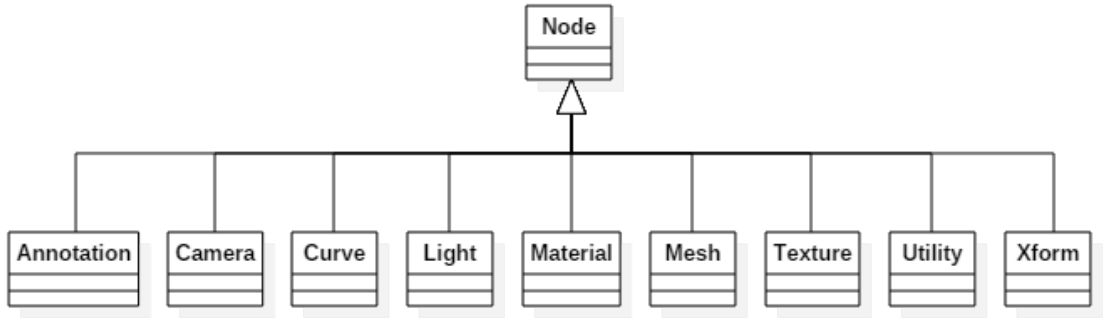


Fig. 3.5 AtomDag node type hierarchy.

Each derived type represents a different Maya node and has specific functionality associated with that node. These specialized node types provide functionality for reading data from the underlying Maya nodes. For the most part, they directly mirror a single node type, such as `Mesh` being a representation of the Maya `MFnMesh` node. The `Light` node, however, combines all supported light source types into a single node for convenience. The `Material` node is an exception, due to how Maya's internal material assignment works. The `Material` node represents a shading group (`shadingEngine` node), which monitors the, if any, connected `surfaceShader` - the material node that users generally interact with - providing methods for reading a large assortment of properties from this connected shader node. `Utility` nodes are also an exception, and will be mentioned shortly. `Xform` mirrors `MFnTransform`, which has zero or more child `Xform` nodes, and a list of components (such as meshes, cameras, and so on). The `Xform` node can be thought of as a node in a scene graph with a list of components nodes attached. In contrast, the remaining nodes, save `Utility`, can be regarded as pure data to be used by an `Xform`. For convenience, the API is configured such that one can ignore `Xform` nodes entirely and rely solely upon the raw data nodes, but this comes at the disadvantage of losing features like instancing.

As previously hinted at, all nodes have yet another watcher system built in, this time with the purpose of propagating changes of the internal Maya node out from the AtomDag system.

Nodes listen for a change in specific attributes, and add a notification of change of that attribute accordingly. These changes are then accumulated until later purged once handled. The AtomDag, at any time, can force all of the accumulated changes to be processed, and any listening observers are notified of the respective change. For example, if an `MFnMesh` has its points manipulated, this is detected inside the corresponding `Mesh` node, and a change is queued that points have been modified. At a later time, these are then processed. This procedure can be seen simplified in Figure 3.6.

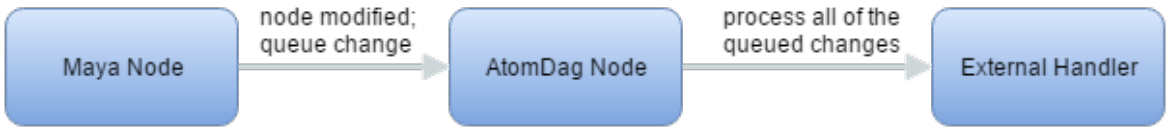


Fig. 3.6 Simplified model of a Maya node change being propagated out of AtomDag.

The aforementioned `Utility` node acts as a method for propagating changes within Maya's DAG that would otherwise go unnoticed. For instance, when performing an extrude in Maya, a `polyExtrudeFace` node is created. When this nodes is modified, i.e. the user performs the extrude, the connected mesh is not notified of its change, but instead, the newly created node is. The `Utility` node handles these new nodes and passes on the respective changes based on the node type it represents to the node it modifies, such as the mesh in the above example. These nodes are transparent to the public-facing API, and act instead as a background element to assist the notification of nodes' attributes or data being modified.

Chapter 4

Implementation

This chapter will analyze in detail the implementation of all major components that make up the Atom plugin. The focus will be to highlight technologies and techniques used throughout development, and provide a road map for how all elements of the system work, and where applicable, interact with other sub-systems.

4.1 Utilities

Before discussing the core components of Atom, a few smaller topics must be covered to ensure a proper understanding of the inner workings of the subsystems.

Configuration File Reader

Atom reads all configuration data (server port, send frequency, etc.) through an INI file located in the same directory as the plugin. Fallbacks exist for invalid or otherwise missing values. The custom-written configuration reader is fairly robust, which makes user formatting errors not too much of a concern.

ManualResetEvent

The `ManualResetEvent`, based upon the .NET implementation of the same name (Microsoft, 2016), provides a way of manually notifying threads that some event has occurred. Its primary purpose is to hold up a thread until a flag is set. Internally, it is implemented with a mutex for locking, an atomic boolean representing a set flag, and a condition variable for the wait-notify system.

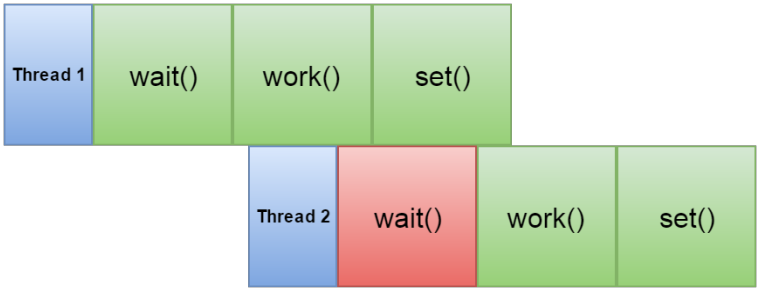


Fig. 4.1 Example use of `ManualResetEvent`.

Figure 4.1 demonstrates a simple scenario. In this example, Thread 1 and Thread 2 have some work they would like to do, and are surrounded by `wait` and `set` operations of a shared `ManualResetEvent` instance. The assumption is made that the `ManualResetEvent` is already set, e.g. upon initialization. When Thread 1 begins, it doesn't have to wait, as the event is already set, so it is free work. While Thread 1 is still working, Thread 2 tries to start. When Thread 2 calls `wait`, the event's flag is not currently set, and therefore Thread 2 will halt its execution and automatically continue upon the flag being set from the other thread. When Thread 1 finishes working, it calls `set`, which in turn allows Thread 2 to continue its work, which then likewise sets upon completion.

The primary reason that this class is core to Atom is that Maya's callbacks, which will be discussed in an upcoming chapter, can cause issues if called while processing other data, as in the example above.

Maya's DAG

To understand the AtomDag system, it helps to firstly have a high-level understanding of Maya's own DAG system and its relationships between node types.

In Maya's DAG, there exists primarily two kinds of nodes: `Transform` nodes and `Shape` nodes. `Transform` nodes contain information regarding the object's transformation information, such as position, rotation, scale, sheer, and so on. These nodes also hold parent-child relationship information. In this sense, `Transform` nodes can be thought of as the primary nodes in a tree system. They also contain zero or more `Shape` nodes, which can be considered leaf nodes in the tree. Practically, every `Shape` nodes must have one direct parent `Transform` node. An example of a `Transform` node that has no `Shape` nodes would be a group, which acts purely as a structural node, e.g. often used in rigging to offset elements.

These objects can be uniquely identified by the path to them from the root node. For instance, in Figure 4.2, the `Shape` node `pCubeShape1` has a path of `|pCube1|pCubeShape1`, as it is a child of the `Transform` node `pCube1`. This diagram also illustrates how multiple

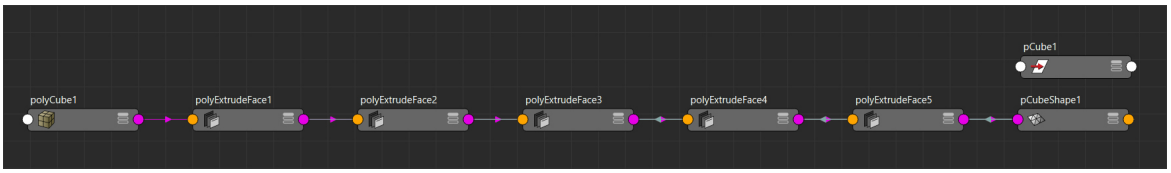


Fig. 4.2 Tree of Maya DAG nodes.

nodes work together to build a `Shape` node. At the far left exists the `polyCube1` node, which is responsible for building the initial cube geometry based on user-provided parameters. Following are a series of `polyExtrudeFace` nodes, of which each performs a different extrude operation, modifying the mesh after each node. At the very end lies the final geometric `Shape` node, `pCubeShape1`. This `Shape` is then directly parented under the `Transform` node `pCube1`. If any of these nodes, prior to the final `Shape` node, are modified, then the modification is passed up the hierarchy until it reaches the final `Shape` node, resulting in modified geometry. It is crucial to keep workings such as this in mind when designing a similar system such as AtomDag.

In addition to DAG nodes, there is also Dependency nodes. They are of a similar nature, but where DAG nodes are acyclic, these can be cyclic, able to pass data in both directions. They are technically a different subsystem, but interact with the DAG. The interaction they provide with the DAG must also be understood to correctly mirror such a system.

4.2 AtomDag

In order to correctly organize and propagate hierarchical changes within Maya's DAG, AtomDag was devised. AtomDag shadows Maya's own DAG, primarily with the purpose of collectively monitoring changes to nodes, and providing a prebuilt hierarchical tree, for easy searching and event propagation. AtomDag is one of the major components of Atom, providing the layer of interaction between the Maya scene and Atom.

AtomDag is built with a wide range of interesting features, most of which will be discussed forthcoming. Put simply, AtomDag is a tree of nodes, some representing transformations, others representing shapes, which are structured based on the Maya scene in question, and respond to changes made within the Maya scene, sending notifications to other systems when certain events occur.

4.2.1 Maya Callbacks

Maya provides numerous callbacks in their API so that plugins can respond to defined events accordingly when they happen. AtomDag uses these callbacks extensively to communicate with Maya so that the shadow copy of the DAG stays true to the original's current state. The notification of addition and removal of nodes, for instance, can be done through Maya API callbacks.

Connections

The `MDGMessage::addConnectionCallback` callback can be used to track connections made or broken between two nodes in Maya. In the case of AtomDag, this callback is used to detect when a material (formally of type `kShadingEngine` - a shading group node) is applied to a mesh. Neither meshes nor materials are informed of material assignment, hence why connections such as this must be tracked to do so. In response to a mesh and shading group being detected in this callback, the according AtomDag node is found, and it is notified that its material has changed. This function can be used for not only mesh/material assignments, but anything that requires connections to be made.

Parenting

It is not enough to simply mirror an initial state of the DAG. Hierarchical changes will quickly desynchronize Maya's DAG and AtomDag. To ensure both contain the same state, the `MDagMessage::addChildAddedCallback` callback is used to monitor parenting operations. In this callback, both the child and parent nodes are provided. Unfortunately, the paths to these objects are often not fully formed at this stage, and since AtomDag internally relies upon a full path name for node identification, their AtomDag counterpart must instead be located by their `MObject` handle. Providing both the corresponding AtomDag child and parent nodes can be found, they are linked accordingly, to ensure synchronization of both DAG systems. In addition, when nodes are moved around the scene hierarchy in Maya, they are not notified of their path changing, so that is enforced here on the found AtomDag nodes.

One major issue encountered during development of Atom is that sometimes the order of nodes being added and parented occur in the wrong order. For instance, if an existing node is placed into a new group, two operations take place: the group is added, and then the existing node is parented under the new group. Due to a bug in Maya, the parenting operation is first called before the new node is even added, which naturally makes for impossible parenting operations. To rectify this, parenting operations containing nodes that cannot yet be found, such as in the above example, are processed instead when the new node is added.

Adding Nodes

The `MDGMessage::addNodeAddedCallback` callback is used to receive notifications of newly created nodes being added to the scene. Upon this event firing, an `MObject` of the new node is provided. An important distinction to make at this stage is filtering out of node types or names that should not be processed, as well as intermediate nodes. Intermediate nodes must be ignored as, for instance, if Maya decides it requires an intermediate mesh to be created for some operation (as is often the case when dealing with triangulated meshes), then both the final and the intermediate meshes would be processed, resulting in not only two copies of the same object, but one in an earlier state than the other. Unfortunately, at the point this callback is delivered, objects are not fully configured, and therefore the distinction between intermediate and final objects is not yet made. This is a confirmed bug with the Maya API (Green, Daniel, 2016a).

The implemented workaround to this issue, although by no means elegant, is to not process nodes upon being added, but instead append them to a temporary list of nodes that need to be processed. After adding to the list, prepare, if not already done so, a callback for when Maya is next considered in an idle state, by using the `MEventMessage::addEventCallback` with *idle* as the event type. This will then queue any nodes that need to be added and not process them until Maya is next idle, when the objects are guaranteed to be fully initialized, and therefore intermediate objects can be successfully filtered out.

Upon the idle event firing, the list of accumulated nodes can then be processed as normal. Due to the nature of Maya's callbacks being freely called at any time, and node processing having to iterate through the accumulation list, the list must be made thread-safe. When each element is processed, they are first filtered based on being an intermediate object, and then on their Maya API type and name. This filtering is to avoid over-saturation of superfluous nodes within the AtomDag to keep it as lightweight as possible. Manipulator and selection set nodes, for instance, are of no use to AtomDag, and therefore are filtered out. If the node in question passes the filtering stage, a corresponding AtomDag node is created and linked to the Maya node using its `MObject` and `MDagPath`. As briefly mentioned in the section discussing parenting operations, if the node type is a Maya transform, then its children are searched for in the AtomDag system, and if found, parented accordingly, along with the corresponding path change notification as before. Following, if the node that is added is a Maya transform, its parent node is found within the AtomDag system, and it is added as a child node of the parent. If that fails, it is added to the root as a failsafe. Alternatively, if the node is not a transform - i.e. it is a shape - then it is added as a component of the parent AtomDag node. Although few in number, there also exists some dependency nodes that are not in the Maya DAG, and those are instead added to specialized lists rather than to the

AtomDag hierarchy directly. Finally, for each added node, a custom callback is triggered to notify external systems of the newly added node.

Removing Nodes

Removal of existing nodes is handled by the `MDGMessage::addNodeRemovedCallback` callback. Nodes cannot be removed by their path name at this stage, as Maya removes nodes in a top-down hierarchical fashion, meaning the paths change in reverse order and no longer represent the expected path. Instead, they are removed based on their `MObject` handle. When a transform is removed, they are removed as a child from any parent transform. Similarly, nodes representing shapes are removed from their parent transforms. A custom callback is triggered at the end notifying external systems that a node was removed.

4.2.2 Node System

The node system is the backbone of the AtomDag architecture. Nodes form the tree system within AtomDag. Primarily, they are referred to by their path name within Maya. This is ideal because the path name is unique and can serve as a useful and human-readable identifier, which is vital to ease working with networked objects. As previously mentioned, nodes can be found by either their name or Maya objects.

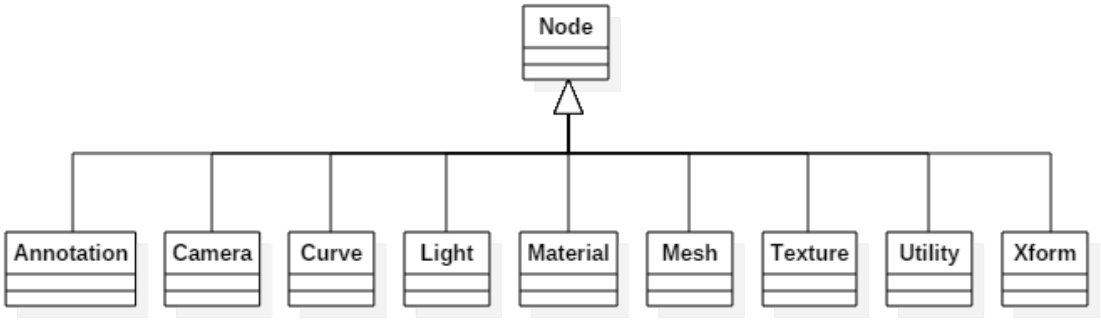


Fig. 4.3 AtomDag node type hierarchy.

There are a variety of node types within AtomDag, but all of them derive from a base type `Node`. The `Node` class contains most of the complexity, and derivations are chiefly responsible for pulling type-specific data from the corresponding Maya node.

The link to Maya from these nodes is in the form of an `MObject` and `MDagPath`. The former represents a single scene object's instance, and the latter a path through Maya's DAG

to reach the node. It is these nodes that, for the most part, derived nodes will use to retrieve data when required.

The `Node` class furthers the use of Maya callbacks. Changes in name are detected with the `MNodeMessage::addNameChangedCallback` callback, and manipulation of node attributes are handled with the `MNodeMessage::addAttributeChangedCallback` and `MNodeMessage::addNodeDirtyPlugCallback` callbacks. As these callbacks are targeted towards a single object, and can be invoked at any time, instances of `ManualResetEvent` are used to prevent interfering callbacks from modifying the same data concurrently. All nodes check for the `visibility` attribute being modified, as to support hiding of objects.

Also stored at the `Node` level is a link to a parent. A constraint is defined that all shape and transform nodes, with the exception of the root transform node, must have a valid parent. This is stored in the form of the derived node type `Xform`. In addition, when `Xform` nodes are translated, rotated, or scaled, their children and components likewise are notified respectively. This will be discussed further when the `Xform` class is analyzed.

In addition to using Maya callbacks, the `Node` system has its own callback system, driven by change. The purpose of this system is to notify external systems of specific manipulations made to the node in question. For instance, if an object were moved, external systems would be notified that it were moved. The change system works by queuing predefined events representing a change in state of the node. At any point, a node can processes the accumulated list of change events, notifying external systems of each change made. Changes are mostly added internal to the derived node in question based on what modification was made. External changes are less common, but occur in, for instance, nodes that exist specifically to modify other nodes, such as extrude nodes. These changes will become clearer as each node type is discussed. They are synonymous to, and sometimes named, notifications.

Annotation

The `Annotation` node represents Maya nodes of type `MFn::kAnnotation`. Annotations in Maya are nodes that hold text strings that are displayed in the 3D scene, often used to label objects. `Annotation` nodes are trivial, with their only focus being on retrieving the text element of the node. In the callback that is triggered when the Maya node's attributes are modified, the attribute in question is determined and responded to accordingly. In the case of the `text` attribute, the `Annotation` node is notified that its text has changed. A function is implemented to retrieve the text element from the Maya object's `text` attribute plug upon demand.

Camera

The `Camera` node represents Maya nodes of type `MFn::kCamera`. Camera nodes in Maya are what defines views into the 3D scene for both creating and rendering. Cameras in AtomDag at one point simply computed and sent the view and projection matrices when components of these matrices changed. This approach, while simple, was limiting for clients. The issue with sending only matrices is that all data that Maya encodes in its cameras are often not used in games, and the client would have to extract single parameters, such as position, if desired. Similarly, values baked into matrices, such as the aspect ratio, can cause issues if the client resolution does not match that of Maya, for instance. Instead, the components that make up the matrices are sent so that clients can either create their own matrices or just use the data. The camera data is largely retrieved from the `MFnCamera` class. When `Camera` nodes are translated or rotated, they are notified that they have changed accordingly.

Curve

The `Curve` node represents Maya nodes of type `MFn::kNurbsCurve`. This includes CV curves, EP curves, Bézier curves, and so on. The CVs of the curve are retrieved in object space directly from the `MFnNurbsCurve` class.

Light

The `Light` node represents all supported light node types within Maya. The reason for a consolidated light group verses specialized light classes is that it makes working with and sending networked data a lot simpler. The light type (e.g. point light, spot light, directional light) is available to differentiate what the grouped parameters represent. Light data is retrieved from Maya using specialized classes dependent upon the situation. For instance, when retrieving the inner cone angle from Maya, the `MFnSpotLight` class is used, and if the light is not a spot light, then a fallback value is provided. This pattern is repeated for other type-specific data. In a similar manner to cameras, lights also respond to translation and rotation notifications. They also filter out specific attribute modifications such that only a single attribute change is notified rather than having to rebuild and send the entire light state irrespective of change to the clients.

Maya's light sources use physically-based falloff models. Video games, on the other hand, often use a single radius or length. One option is to compute or approximate the equivalent value, but that is not very intuitive for the artist to use. A creative approach to this problem is to use the light source's `centerOfIllumination` value. This value is defined as the distance between the light source's origin and it's directional manipulator. In Figure 4.4, the

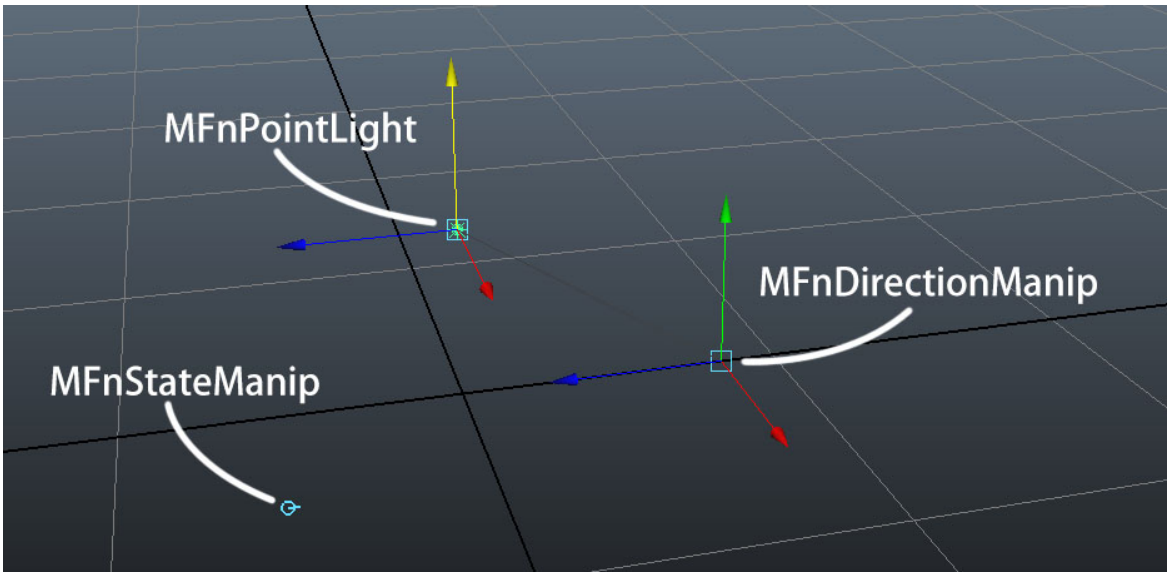


Fig. 4.4 Point light, manipulator, and state.

`MFnPointLight` is the light source's origin, and the `MFnDirectionManip` is the directional manipulator. Measuring the distance between these nodes - the `centerOfIllumination` value - provides artists with a visual guide to the extents of the light source. This also applies to other light sources, such as the spot light. Another advantage of this methodology is that the directional manipulator can be toggled off using the `MFnStateManip` or simply switching to another tool, such as select or move.

Material

The `Material` node represents shading groups and their connected surface shader. In Maya, shading groups control the application of shaders to components and volumes, e.g. mapping a shader onto specific polygons. In video games, however, the tradition is to have a single material per object, with multiple passes being used for layered effects. To simplify both the implementation and data sent to the client, only a single shading group can be applied to an object. It is not a strenuous task to add support for multiple materials per object, but the average use-case scenario puts it at a disadvantage over a single material. While shading groups control the allocation of materials, the surface properties are defined by the shader connected to the `surfaceShader` plug. This is sometimes an empty connection, which must be considered when parsing accordingly. Since the shading group itself can change its allocated shader, the modification of the `surfaceShader` plug is checked for, and notifications are added accordingly. This means that the user can connect a different shader to the `surfaceShader` plug of a shading group in Maya and see it reflected in the client. The

advantage of this system is that the use of shading groups, as in Maya, is largely transparent to the user, with the connected shader itself being what triggers property updates.

The `Material` class traces the connected shader, if any, through the `surfaceShader` plug, and adds additional Maya API callbacks on the found shader object. In a similar fashion to the `Light` class, certain attributes, when modified, are filtered out to ensure notification of only specifically changed values, as to avoid over-saturation of data that did not actually change. This is especially important for materials, as they contain a lot of data. Each material parameter is of the type color (RGBA), single (float), or texture (`Texture` node name). The majority of parameters can have either a color or single, combined with an optional (if set) texture. For instance, the `color` channel can be either color or texture, depending on if a texture is connected to the slot. Although there are numerous concrete material types within Maya, all values are received directly from plugs, irrespective of type. The type is used only to define which data to send to the client.

Mesh

The `Mesh` node represents Maya nodes of type `MFnMesh`. Internally, Maya stores its geometric data scrambled and in a very confusing way. There is reason for this, as it makes working with it from an algorithmic point of view easier. For games, however, where pure triangles are the norm, it is not suitable. When `Mesh` nodes parse Maya's geometric data, it is all automatically processed as a triangulated mesh, regardless of the Maya mesh itself being triangulated or not. Both multiple UV/tangent/binormal sets and color sets are supported. Unfortunately, there are a number of bugs in the Maya API specifically related to getting UV information. If `MItMeshFaceVertex` is used, the UVs retrieved using the `getUV` command do not update. The alternate is to use the `MFnMesh`'s `getUV` command. This method does update the UVs when they are changed, but is also susceptible to issue. When UV points are rotated or scaled, the `uvsp` (UV Set Points) attribute is modified. When a notification of this attribute is invoked, and a triangle mesh is used, the function for retrieving UV data will crash internally within Maya, which makes the approach not suitable. As an unfortunate side effect of this Maya API bug, AtomDag must instead rely upon translation of UVs, which triggers a different event.

Attributes are filtered to ensure the entire mesh isn't sent when only a subset of the geometry is modified. When points are modified, for instance, the attribute `pt[x]` is generated for each point, where `x` is the index of the vertex that changed. As an optimization, the points that move are accumulated until just before sending of changes occurs, adding a last minute notification that the accumulated list of points changed. Similarly, the translation of the UV pivot - a translation of UV points in the UV editor - is captured. Unfortunately, indices

are not provided as with movement. Instead, the active selection list is parsed and likewise accumulated until just before sending of changes. To avoid rebuilding geometry all of the time, a dirty flag is used to specify when to rebuild geometry accordingly. To allow clients to ignore `Xform` nodes and instead directly use `Mesh` nodes, they respond to translation, rotation, and scaling operations, and have a world matrix in addition to their local-space data.

Texture

The `Texture` node represents nodes of type `MFn::kFileTexture`, which includes both standard textures and derivations, such as PSD nodes. In Maya, the placement parameters of a texture - repeat, wrap, rotation, and so on - are stored in the texture node itself. They are, however, often controlled externally, such as by nodes of type `kPlace2dTexture` or `kPlace3dTexture`. The `Texture` node has access to all of the parameters expected for texture placement, and they are included when sent to clients. More importantly, the full path of the file the Maya texture node represents is included, useful for scenarios where textures are accessible regardless of the location, e.g. a shared machine. For networked clients that do not have access to the full path provided, the raw bytes are also included, so that clients can load the texture from memory. The extension of the file is also included to make loading the texture from raw data easier. Similar to other nodes, attributes are filtered out as to not send the entire texture data when single attributes change.

Utility

`Utility` nodes consolidate many types of intermediate nodes. Included are nodes such as polygon extrudes, vertex merges, polygon generators such as poly cubes, and so on. These nodes act as building blocks to generate the final mesh, each node making an iterative modification. When these nodes are modified, the end product, such as a mesh, is not notified of the changes, and therefore the purpose of this class is to ensure the changes are correctly distributed to respective nodes accordingly. The nodes are completely transparent to clients and act purely as a notification propagation medium.

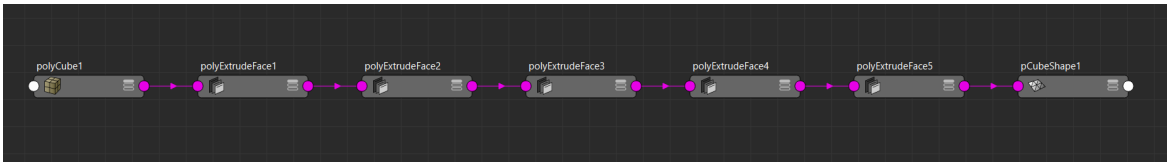


Fig. 4.5 Multiple utility nodes building a mesh.

A practical example can be seen in Figure 4.5. From the left, the first six nodes are, in AtomDag, `Utility` nodes, and the final a `Mesh` node. The first node, of type `MFn::kPolyCube`, generates a polygon cube with tweakable parameters. Each time one of these parameters is changed, the `Utility` node notifies the resulting mesh - `pCubeShape1` - that its geometry has entirely changed. The five nodes that follow represent a single extrude operation, and are all of type `MFn::kPolyExtrudeFacet`. Similarly, these also notify the resulting mesh that its geometry has changed. Other operations this node represents modify the geometry only partially, such as applying softened edges. In these cases, the component indices are read from the corresponding plug and accumulated similar to how the `Mesh` node accumulates moved points. As can also be seen from the above diagram, not all nodes have the resulting node as a direct successor. To rectify this, the resulting node is found by recursively tracing the `output` plug until the desired node is found.

Xform

The `Xform` node represents Maya nodes of type `MFnTransform`. These nodes are quite simple, but serve an important purpose. Each `Xform` node contains a list of children and components. Children are further `Xform` nodes directly below the current node in the tree, which can in turn contain more children. Components are other data-based nodes, such as meshes, lights, cameras, and so forth. Components are stored as direct links in memory, but to clients, are sent by name. An `Xform` node has no limit on the number of children or components, although it is most common to have zero or one components. An `Xform` with zero components can be likened to a group node in Maya. In addition to the child and component system, `Xform` nodes are responsible for propagating translation, rotation, and scale events to its children to in turn be passed down the tree to all leaves. Similarly, name changes are likewise propagated, due to Maya not triggering rename events when parent objects are renamed or moved in the hierarchy. Naturally, they also contain a transformation matrix.

As previously mentioned, clients have the option of either using or ignoring `Xform` nodes. If they are to ignore them, features such as instancing will not work, as `Mesh` nodes would have but a single world matrix. Instead, using the `Xform` nodes as transformation information for raw data nodes allows for instancing, among other features, to work correctly. This is because the `Xform` node simply references the `Mesh` node components, and provides a unique placement for that node, enabling sharing and reuse of nodes, as does Maya.

4.2.3 Rebuilding

AtomDag can respond to changes in nodes and the graph hierarchy. That alone is not enough, as the graph must first be created, in order to process existing nodes. This is done through rebuilding operations. There are multiple reasons a rebuild operation may be triggered. For instance, upon initialization, in response to a Maya event, such as opening or closing a scene, or even manually.

The rebuild operation itself consists of a number of steps. Firstly, the old contents are discarded to ensure a clean slate. Next, the root node of Maya's DAG hierarchy is obtained, and stored in AtomDag as an `Xform` node. The root node is retrieved from Maya using the `MItDag` class. Following, a recursive node processing function is started at the root. This function iterates over the node's children, creating the according AtomDag nodes where applicable, and recursively processing for each new node. After creation of the node, the parenting is set up accordingly for `Xform` nodes, and all other raw data nodes are assigned as components to the parent in question.

Unfortunately, not all nodes are captured with this method, as dependency nodes are not officially part of the DAG hierarchy. This includes nodes considered a `Utility` in AtomDag, as well as `Material` and `Texture` nodes. This is resolved by iterating with an instance of `MItDependencyNodes`, which linearly traverses every dependency node of the given type. Each node found is created accordingly.

After rebuilding of the graph is complete, a custom notification is sent out to external listeners along with the reason of instigation for the rebuild operation. This is used to let clients know that the scene has been newed or opened, for instance.

4.3 Messages

The transmission of data to clients is done using Protocol Buffer messages. This is for both for performance and ease-of-use in terms of reconstructing different message types from raw networked streams. There are a large number of Protocol Buffer messages within Atom. Only the single core message, `AtomMessage`, is sent directly, which means clients can simply rebuild this message type from the network data stream with confidence that it will be correct. All other message structures are embedded within this generic message. `AtomMessage` uses a feature of Protocol Buffers, namely `oneof`, which allows for a definition of many potential parameters to be listed, but only a single to be encoded and sent, saving memory by not including those not picked from the parameter list. This simple method means all potential messages can be listed with only the message in question being physically present. This is optimal for performance, size, and ease-of-use.

Since Protocol Buffers rely entirely upon either data internal to themselves, or upon other existing Protocol Buffer messages, all basic data types have to be redefined. While this is inconvenient, the advantages of using this system greatly outweigh the downsides. Atom defines common messages such as vectors, colors, matrices, and vertices.

Each AtomDag node type, with the exception of `Utility` classes, has their own corresponding Protocol Buffer message in order to send data from the respective class. Each message always contains the name of the AtomDag node, as a string. This makes declaring links and knowing which objects to update easier. A simple ID, such as the hash of the node's name, could have been used, but this is not very user friendly, and makes issues difficult to debug. Due to the efficiency of Protocol Buffers, the sending of a string is not too large.

There are too many classes to cover in detail, but there are a couple that should be highlighted. The `Scene` message is how clients are notified of destructive modifications to the tree's hierarchy. This message is used to notify clients of scene contents upon connection, a new scene in Maya being created, an existing scene in Maya being opened, nodes being added, and nodes being removed. In the message is multiple lists of another message type, `SceneObject`, which in itself simply contains a name. These lists define the types of objects in the scene without the need for an enumeration, e.g. all scene meshes reside in the list representing meshes. When nodes are added or removed, the lists will be populated accordingly with the objects that need to be acted upon.

In a similar fashion to the `Xform` node, the `Transform` message contains a world matrix, and a list of components. Child hierarchies are not transferred, but instead are internal to AtomDag. This is something that could be added with relatively little work, though. The components are defined as an object name alongside an enumeration declaring the type of the component (mesh, light, camera, and so on).

Since all objects in AtomDag support renaming, a message `NameChanged` exists for this purpose. Inside data is trivial, simply containing an old name and a new name. It is then up to the client to find and replace the identifier that they use. A similar situation is with the `MatrixChanged` and `VisibilityChanged` messages, which send a newly updated world matrix and visibility settings respectively. The visibility of an object is controlled by three factors. The first is the object's own local visibility flag. The second is the draw override visibility, taken into account when the draw overrides are active; this is so that features such as display layers work as expected, as they control the draw override parameters. Finally is the parent hierarchy's visibility; in other words, if any parent object is invisible, the node in question is also considered so.

Lights are true to their AtomDag class, sending only but a single compound class representing all supported light sources. This is because a lot of the data is shared, and the

class is so small that splitting it up would be almost redundant. Instead, a simple enumeration defines the light's type. In addition, when parameters of the light change, a `LightChanged` message is sent, which exploits Protocol Buffers' `oneof` property once more to ensure only the modified property is sent versus the entire light. This is for optimization and to ease the use for clients.

Meshes work similar to lights, with a base message sending all of the data, with other messages sending only updated data when required. When a mesh's points are modified, the `MeshPointsChanged` message is used, which contains only the modified data, and not the entire mesh. For models with a large vertex count, this is essential. When a mesh's material changes, `MeshMaterialChanged` acts with a similar purpose. For those operations that are destructive, however, where the geometric data has been altered in size (i.e. a vertex or index added or removed), the entire mesh is sent again. While this does mean that operations such as extruding require the entire mesh to be sent once more, potentially wasting data for a couple of added polygons, it is better than the alternative, which is to ask the client to implement an extrude function themselves, among other geometric manipulations.

Materials work in a similar method to the base message, once again using `oneof` to declare which concrete material class the message truly represents. Opposing the light's methodology, each material type has its own standalone Protocol Buffer message. This is because, unlike the light, there is a lot more data associated with materials, and although they all share the same base properties as Lambertion shaders, their extensions differ greatly. Each concrete material class has all of its parameters embedded within the corresponding message. As with in AtomDag, parameters can either be colors or singles, or a texture. The majority of material parameters use the `TexturedParameter` message, which contain either a texture, or other type of data, again using a `oneof` so that clients know which data to retrieve for the parameter in question. As with previous messages, single parameter changes are propagated to clients through the `MaterialChanged` message, using `oneof` to ensure only relevant data is sent, which is especially important with messages with as much data as materials.

The downside of using Protocol Buffers for messages is that clients are now limited to use a programming language that is supported by Protocol Buffers. This list, however, does contain all of the major traditional languages, and support is growing all the time. Another advantage of using Protocol Buffers is that the source code - `proto` files - double not only as source files for language-specific compilation, but also as documentation for the messages received.

4.4 Server

Atom's server is responsible for the bidirectional communication between clients and the Atom plugin instance. It is based upon the ASIO networking library. ASIO was chosen due to its performance, stability, and robust asynchronous workflow. A number of considerations had to be taken into account when designing the server infrastructure. Firstly, it was known that clients could be interacted with from multiple threads in parallel, to request a networked write, for instance. In addition, receiving information would likewise be parallel from multiple clients concurrently. All of these methods need to be thread safe and reliable. Notifications of connections made, connections lost, messages received, and so on, must be sent out to external systems within Atom, so custom callbacks have to be created, too. Although UDP would be much faster, it is less reliable than TCP, and when reconstructing data from bytes, everything must be in order and accurate. Therefore, TCP is the obvious choice of protocol.

The resulting server contains three classes that work in tandem to bring the system together. The `Message` class is used to transfer data, both to and from, networked clients. The `Connection` class works as a local representation of the networked client, providing the interaction for both reading and writing standalone from the core server. Finally, the `Server` class uses the aforementioned classes along with its own logic to provide a server socket and useful client-based functions, such as sending messages to all clients, and containing callbacks for set events.

Message

The `Message` class is responsible for holding both incoming and outgoing data. It is built to be automatically configured for either operation. The class structure can be seen in Figure 4.6.

For clients to read the correct amount of data, they must be aware of the size of the data. When sending an outgoing message to clients, `data_` contains an encoded header followed by the raw data. The encoded header is the size of the raw data, of type signed 32-bit integer (`int32_t` in C++), bit shifted and packed into a four byte array. Clients can then unpack the bits and reconstruct the value, providing them with the size they need to read. In the case of incoming messages, clients are expected to provide the same style header in return. Atom only uses the decoded header to allocate the `data_` block, which streams in only the raw data, excluding the header bytes.

The `Message` class itself cannot be directly created, but instead, one of the two static creation functions must be used: `createOutgoing` and `createIncoming`. The former sets

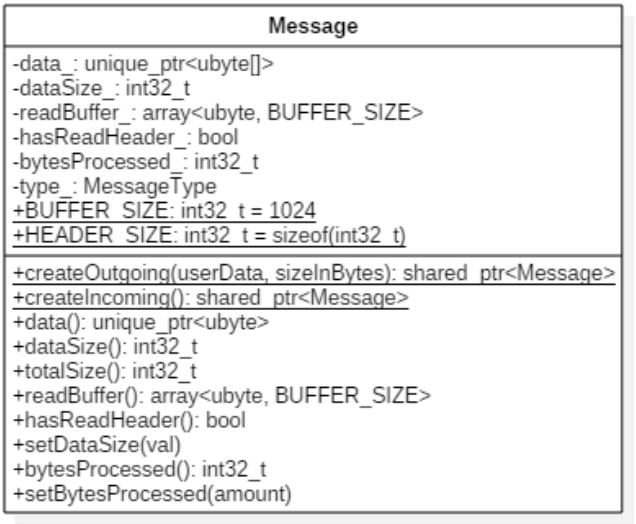


Fig. 4.6 `Message` class structure.

the `type_` to be outgoing, sets the `dataSize_` so that the write loops know how much data to send in total, and allocates and fills the `data_` block with the encoded header followed by the raw data. The latter instead simply creates an empty message with only the `type_` being set to incoming; all allocation is done after the header has been read from the client.

When writing, the data already exists, and so the data is recursively sent until the number of bytes processed matches the total size of the allocated data block. With small messages, all of the data can be sent at one time, but due to the size of the data that Atom deals with - textures, geometry, and so on - not everything can be sent in one go and must instead be streamed until complete.

When reading, the first four bytes of the received data are taken, decoded, and used as a size to allocate the incoming data block. The data is then recursively streamed until all has been read according to the given size. The reason for streaming the data is the same as before; not everything is guaranteed to fit in a single read buffer.

Starting the Server

In Figure 4.7, the process for starting a `Server` instance is detailed. Initially, a server socket is opened for listening. This is done on the specified port on the local machine. Following a successful opening of the server socket, an asynchronous, looping thread for listening is started, indicated by the second lifeline in the diagram. To avoid accepting multiple clients at the same time, a `ManualResetEvent` is firstly reset at the top of the loop. An asynchronous

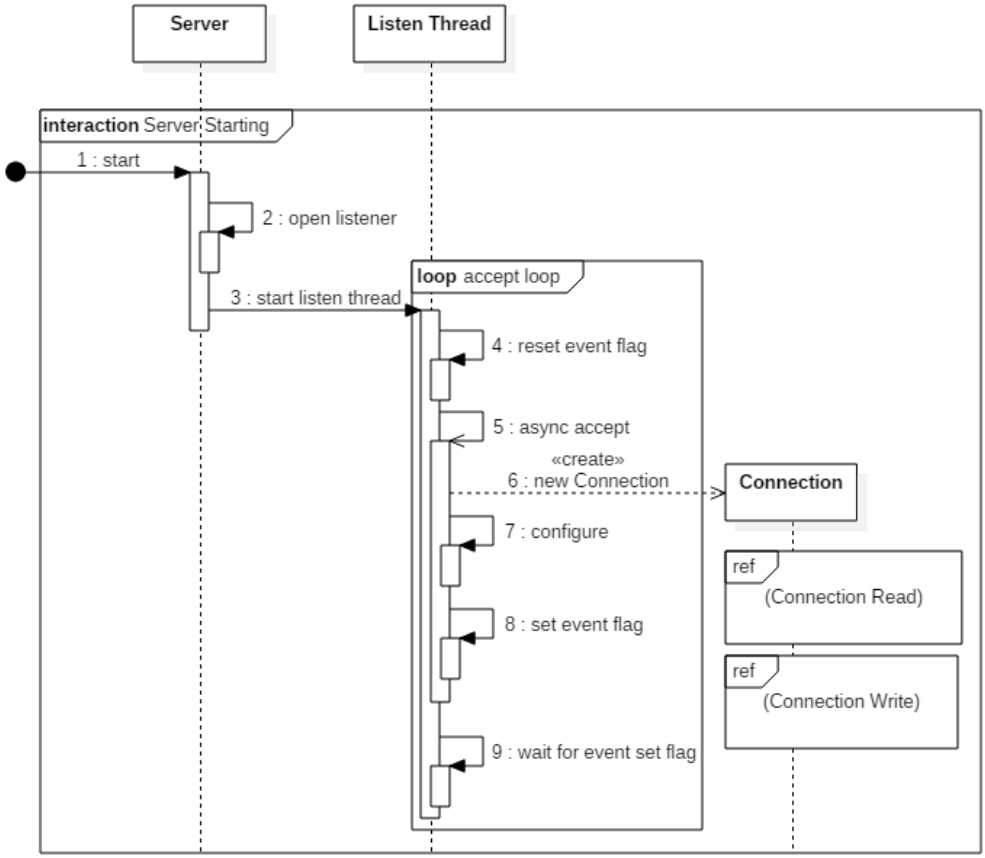


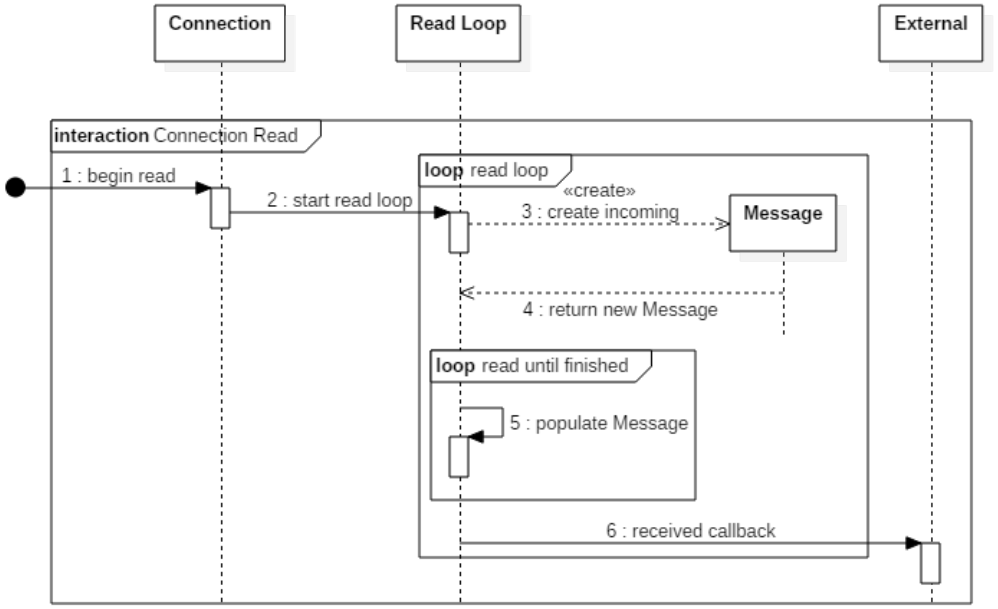
Fig. 4.7 Starting the `Server`.

acceptance function is then called, which due to its nature, returns control without blocking. The thread then waits on the event, which is not set until a single client has been accepted and processed. Failsafes exist to force the asynchronous call to fail, e.g. when the server is closed, causing the accept loop to terminate. Upon a client connecting, the asynchronous function's callback is triggered, and a new `Connection` is created. The details of the `Connection` will be covered shortly. After storing the newly created connection accordingly, the event flag is set, allowing for the loop to then continue waiting for the next client.

Connection Reading

Connections read information in their read loop, as shown in Figure 4.8. The read loop, just like the `Server` accept loop, is asynchronous. Firstly, a new `Message` is created and configured for incoming data. After, an asynchronous read is started to populate the new `Message` with data. Firstly the header is read, as to know the size of the remaining incoming

data. Following, the actual data itself is read until completion, at which point the external callbacks are notified that a message has been received from a networked client, and the read loop is restarted in preparation for the next message.

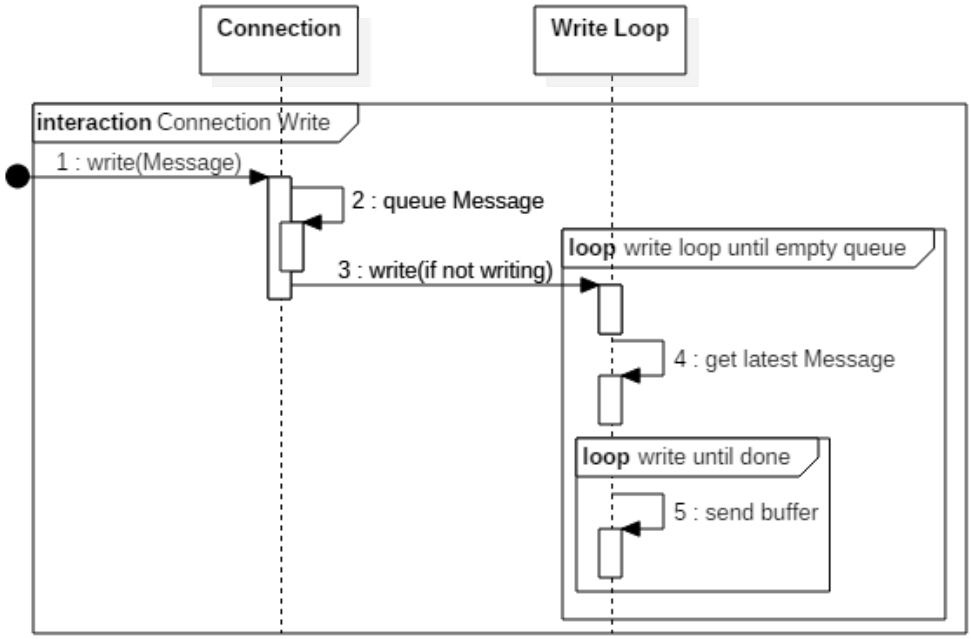


0.9

Fig. 4.8 Reading with a `Connection`

Connection Writing

Connections queue write requests, which are processed in an asynchronous write loop, as can be seen in Figure 4.9. The `Message` requested to be written is one configured for outgoing communications. When a write is requested, the `Message` to be written is added to a queue of pending messages. If this queue was previously empty, the asynchronous write loop is started. If it is otherwise already running, appending the queue is enough to have the message be processed. When processing queued messages, the front `Message` is peeked and is then written asynchronously. Similar to the read loop, it is written until completion. Upon the write completing, the `Message` is removed from the queue, and any remaining (including those added during the asynchronous write call) are then processed. If there are no more left to process, the asynchronous function returns and terminates until a subsequent write is requested.



0.9

Fig. 4.9 Writing with a `Connection`

4.5 Client/Server Interaction

Since the public-facing API of Atom is the core of the product, it helps to have an understanding of the interaction between Atom and clients. To demonstrate this, the accompanying client implementation will be analyzed, chiefly focusing on the communication with Atom. For simplicity, the client implementation project will be referred to as the *client*.

The client is built in C#, using *MonoGame* (MonoGame Team, 2016) as a base. Everything else was developed specifically for this project, from the ground up. To show how practical this client can be for previewing assets, it contains a custom High Dynamic Range rendering pipeline with numerous shading models, varying tonemapping effects, automatic eye adaptation simulation, brightness blooming, and many configurable options. It also contains a networking layer that is in similar design to that of Atom's own. There is also a custom scene implementation to receive and store data from Atom, as well as handlers for each of Atom's incoming messages.

Scene

Before discussing how the client interacts with and parses data received from Atom, it would make sense to firstly gain an understanding of the scene structure the client uses to store such data. An unfortunate downside of Atom, as a direct consequence of how Maya handles notification order internally, is that sometimes objects can be sent to clients in an order that doesn't quite make sense. For instance, a transform could be sent with a list of components it has, before the components are sent. While this is not too difficult to work around, it is a little tedious and client implementation shouldn't have to do such a thing. This is an acknowledged issue and an area for improvement in future iterations of Atom. With that in mind, the client's scene structure was designed to take this potentially missing data into account.

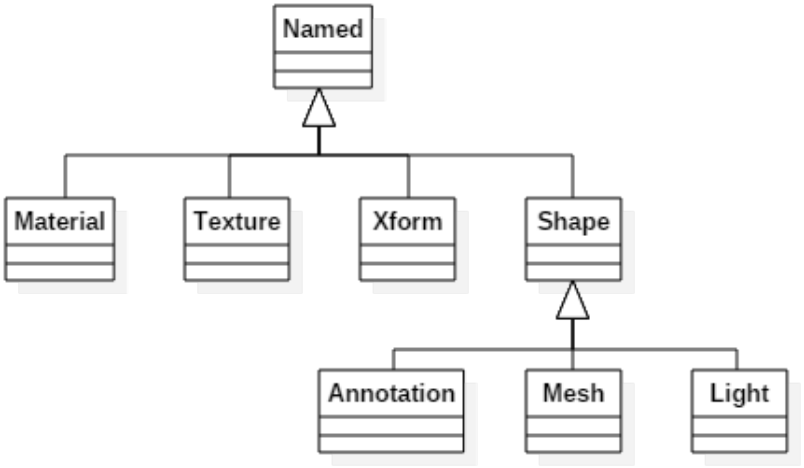


Fig. 4.10 Client's core scene classes.

The core classes of the scene are displayed in Figure 4.10. `Named` and `Shape` are the base classes from which concrete objects inherit. The difference between the two is that the former is simply named, whereas the latter includes visibility and transformation information. `Texture` objects are simple, housing the ability to load textures dynamically from either a local file or from memory. Atom sends either a texture filename or raw bytes, and this class can handle both. `Material` objects store all material information that is used by the shading models supported by the client's rendering pipeline, as well as cached links and names of related `Texture` objects. As previously mentioned, the order of objects is not guaranteed, and therefore a `Texture` may be referenced before it is sent. To solve this, the names of the objects are stored, and if they do not exist, are linked upon addition of the `Texture` itself. `Mesh` objects contain all of their geometric data, with methods of updating that data, as well

as cached links, akin to how `Materials` are linked to `Textures`, for assigned `Materials`. `Annotations` are simple in nature, providing only a textual string. `Lights`, on the other hand, are more fulfilling, supporting numerous types of lights in one consolidated class, just as Atom does. Finally, the `Xform` objects hold cached links to their respective `Shape`, as well as visibility flags and a transform. As mentioned previously, Atom is designed so that transforms are optional, at the expense of features such as instancing. This example client implementation chose to use the transform objects, but it is entirely possible to create a client that does not.

All of these objects are tied together by the `SceneManager` class. The purpose of this class is to centralize lists of transforms, shapes, textures, and materials, and provide methods for managing them. Upon addition of objects, links including the object in question are processed. For instance, when adding a new `Texture`, all `Material` objects that have requested a texture of the same name are linked to the new object. This process is repeated similarly for all links. Upon object removal, links are removed where appropriate in a similar fashion. The manager also supports renaming of objects, setting of matrices, and setting of visibility. When receiving data from Atom, the manager is often interfaced with.

Handling Atom

The client utilizes a collection of handlers, each with a specific purpose, to respond to the incoming data from Atom. All of these handlers are unified in a `ClientHandler` class, which upon receiving a complete message from Atom, converts the message into an `AtomMessage`, and then iterates through the registered handlers until a single handler reports it successfully parsed the message.

Inside each handler, the set message within the `AtomMessage` is checked, and if it matches the expected type, the data is processed. For instance, if the handler for annotations detects the `Annotation` message was set, then it will respond accordingly. If it were not set, the next handler in the chain attempts to handle the message.

Messages relating to the creation or modification of shapes, such as `Mesh`, `Transform`, `MaterialChanged`, and so on, all follow a similar pattern. For the messages referring to instantiation, the handlers request an object by name from the scene manager, and if the object is not found, they then create, populate, and add it accordingly. When messages that update objects are received, they instead check for the existence of an object, and update the respectively modified parameter instead.

Messages that have no specific underlying type, such as setting matrices, changing of name, or changing of visibility, rely entirely upon the scene manager to do the work for

them. The handlers for the `NameChanged` message, for instance, uses the scene manager's renaming functionality with its `OldName` and `NewName` parameters.

The exception to all of these formats is the `Scene` message. The client responds to this message differently dependent upon the `TransferReason` enumeration. When the scene is initially sent, a new scene in Maya is opened, or a new object in Maya is added, the lists of objects accompanying the message are traversed, and each object that the client cares about is requested to be watched. Watching is done by sending a string back to Atom in the format `watch object_name;` where `object_name` is the name of the object to watch. When Atom processes this, it sets up a link between the client and the object so that any future messages (instantiation, modification) relating the object will be passed to the client, where one of the previously mentioned handlers will respond accordingly. Currently watched objects can also be unsubscribed from at any point by sending Atom the `unwatch object_name;`.

Chapter 5

Conclusion

Atom aimed to provide an abstracted, open API for programmers to interact with from custom clients in order to stream Maya scenes for interactive previewing during development, mitigating tedious exporting steps in traditional artistic pipelines within video games. Overall, the project has been a large success. The API has proven favorable, and the example client is a fantastic proof-of-concept to show what is possible with relatively little work required. The project has been rather technically challenging, but the results are worthwhile. While at the time of writing, Atom is not yet production ready, it provides fantastic foundations for future works to build upon, both in terms of concept and code base.

The project is, though, not without flaws. A number of improvements could be made relating to Maya API workarounds. The ordering of messages as of writing, for instance, is not guaranteed to be intuitively sequential (i.e. an `Xform` may come before the `Mesh` it references). This is one of Atom's biggest areas targeted for improvement, as it makes implementing clients that much more difficult. While the problem has roots in Maya's API, the clients should not suffer and Atom should handle this, and similar cases, to function as would be expected.

Performance

To test performance of Atom in a best-case scenario, both the Atom plugin and example client were run on the same local machine. While still technically networked (i.e. communicating via sockets), the delay is minimal. Physically distant connections will naturally be slower. Three scenes of differing complexity in a variety of ways were initially opened. One had a large number of textures, another representing an average video game mesh without textures, and one extreme case with multiple thousands of objects and meshes. Specifics can be seen in Table 5.1.

Table 5.1 Local performance test data.

Scene	Data					
	Xforms	Lights	Meshes	Annotations	Textures	Materials
helmet.ma	7	3	3	0	6	4
mako.ma	51	3	39	4	0	2
natural_history.ma	3082	0	3081	0	0	14

Each scene was individually loaded into Maya, and upon the scene being fully opened, the Atom plugin was initialized. Following initialization of Atom, the client was launched and connected to Atom, downloading the initial scene. Once the scene had been completely transferred to the client, statistics were polled. These statistics can be found in Table 5.2.

Table 5.2 Local performance test sizes.

Scene	Sizes		
	Messages	Total KB	Total ms
helmet.ma	26	2649	27
mako.ma	116	2021	29
natural_history.ma	6180	159390	1724

Naturally, more data takes a longer time to transfer. An interesting correlation between the number of messages sent and the total transfer time can be seen by comparing the *helmet.ma* and *mako.ma* scenes. The former has a moderate 26 messages, but totals 2649KB, whereas the latter has a higher message count of 116, but a lower total size of 2021KB. Surprisingly, the latter, with less data, took longer to transfer. This is most likely due to the fact that the message count was higher, and therefore the number of unique transfers was greater. If instead all data was somehow transferred in one large chunk, it is without doubt that the smaller would transfer quicker. For future works and improvements on Atom, the actual message count should be considered with this knowledge at hand.

Another notable area for potential improvement is the gathering of data within Atom. Operations such as mesh geometry rebuilding can be costly, causing potential slowdowns in Maya, which is undesirable. Performance in AtomDag should be considered in future iterations.

Third-Party Feedback

To gain post-development third-party feedback, an additional survey was distributed. The survey consisted of a video (Green, Daniel, 2016b) showing the core principles behind Atom in action with the developed client as an example. For programmers, API documentation and the source code to the example client were also provided.

The survey allowed participants to answer as either an artist or programmer, with questions tailored to the respective discipline. Artists were questioned about the usefulness and practicality of Atom as an end-product, whereas programmers were questioned about the clarity of the API documentation and whether they felt comfortable using it to create a client. A section was provided for both to give general comments on the project, and programmers had the option of specifying how they would like to see the API improved. All results can be found in Appendix B.

The number of artists versus programmers taking the test were fairly even. Out of the artists who took the survey, all but one considered the project useful if a client were to be implemented in their favorite game engine or rendering software. Since further clients can be developed for these platforms, the consensus of this result shows that Atom is practical when applied. One of Atom's initial goals was to reduce the number of steps required for artists to preview work with in-game visuals during development, and when asked about this, almost every participant agreed that Atom would successfully reduce the steps required when used. This overwhelming response confirms Atom has met one of its primary goals. A common trend between artists' comments was that they find the tool useful and that it does indeed reduce the steps required for previewing. One concern was that programmers must assist with configuration. While this has truth to it, once the custom client has been created, no more programming knowledge is required and everything will work automatically. A number of people said they find Atom more suitable to single-object previews. Although Atom is not limited to single objects, this is a common use-case scenario. Animations were highly sought after. At the time of writing, Atom does not support animations directly, as Maya does not trigger attribute changes on animation. This is something that could be added in the future, though. Credible claims were also made towards working with custom nodes in Maya that are therefore not supported by Atom. This is an area for improvement, possibly reworking how nodes are handled and having Atom be configurable by type. Since Maya works on name-based attributes, this is a possibility, even for custom node types. While artists' concerns are valid, one has to recall that Atom is an API; the example client was just a proof-of-concept for using the API and does not show the full potential in comparison to using a fully-fledged game engine.

Programmers, on the other hand, when queried about the clarity of the API documentation, collectively met with an overwhelmingly positive response. As programmers are responsible for implementing a client and are therefore the target audience of the API, this result is welcomed. Additionally, all respondents agree that they are confident to implement their own client after looking at the API documentation and example client. Being the target audience of the core of Atom, their feedback on the project structure must be considered. Recommendations include incorporating code samples from the example client directly into the documentation, which would make it easier to read and apply. Another suggestion was to categorize the documentation into clearer headings, such as *Setup* and *References*. One of the most useful suggestions was to modify Atom to send error codes where applicable so that clients could know what went wrong if something did so. This is an invaluable recommendation and will definitely be considered as a focus during future iterations of Atom and its API design. Although most thought that the documentation was clear, it could be improved. A modified version of documentation would ideally explain at a level suitable for beginners, with more clarity and lack of ambiguity between parts, where applicable. Multiple participants mentioned that they work in AAA game companies with solid pipelines, but as a side effect, artists cannot visualize quickly during development, and share the opinion that Atom would be indispensable for their project. One participant even suggested that it would be suitable for their own personal engine.

Feedback has been overwhelmingly positive and constructive, with many valid points being raised that lay foundations for future work on the topic. Atom has been a successful project both in terms of artistic and technical practicality and technical achievement.

References

Autodesk (2016a) *Maya* (2016 sp6) [Computer program]. Available at: http://www.autodesk.co.uk/products/maya/overview.

Autodesk (2016b) *Stingray* (1.4) [Computer program]. Available at: http://www.autodesk.com/products/stingray/overview.

Boost (2016). *Asio*. Available at: http://think-async.com/.

Epic Games, Inc. (2016) *Unreal Engine 4* (4.12) [Computer program]. Available at: https://www.unrealengine.com/.

Giordana, Francesco (2011) 'High Quality Previewing of Shading and Lighting for Killzone3', *ACM SIGGRAPH 2011 Talks*. Vancouver, British Columbia, Canada, New York, NY, USA: pp. 50:1–50:1.

Google Inc. (2016). *Protocol Buffers*. Available at: https://developers.google.com/protocol-buffers/.

Green, Daniel. (2016a) 'kMesh intermediateObject bug', *Autodesk Developer Forums*, June 21. Available at: http://forums.autodesk.com/t5/maya-programming/kmesh-intermediateobject-bug/m-p/6396421.

Green, Daniel (2016b). *Atom Project Feedback Video* . Available at: https://www.youtube.com/watch?v=RJbK1wPqgwU. Accessed: 05 August 2016.

Jan-Bart van Beek, Valient, Michal, Giesbertz, Marijn, and Bannink, Paulus. (2011) *The Creation of Killzone 3* [Presentation]. SIGGRAPH 2011.

Labschütz, Matthias, Krösl, Katharina, Aquino, Mariebeth, Grashäftl, Florian, and Kohl, Stephanie (2011). Content creation for a 3D game with Maya and Unity 3D. *Institute of Computer Graphics and Algorithms, Vienna University of Technology*.

Microsoft (2016). *.NET ManualResetEvent Class*. Available at: https://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(v=vs.110).aspx.

MonoGame Team (2016). *MonoGame*. Available at: http://www.monogame.net/.

Rémi, Arnaud (2010). The Game Asset Pipeline In Eric Lengyel (ed.) *Game Engine Gems* (pp. 11–39). Jones and Bartlett, Sudbury, Massachusetts.

Sander van der Steen. (2014) *Killzone Shadow Fall: Creating Art Tools for a New Generation of Games* [Presentation]. Game Developers Conference.

SideFX (2016) *Houdini Engine* (15.5.480) [Computer program]. Available at: https://www.sidefx.com/products/houdini-engine/.

Unity Technologies (2016) *Unity* (5.3.4) [Computer program]. Available at: https://unity3d.com/.

Appendix A

Survey Results

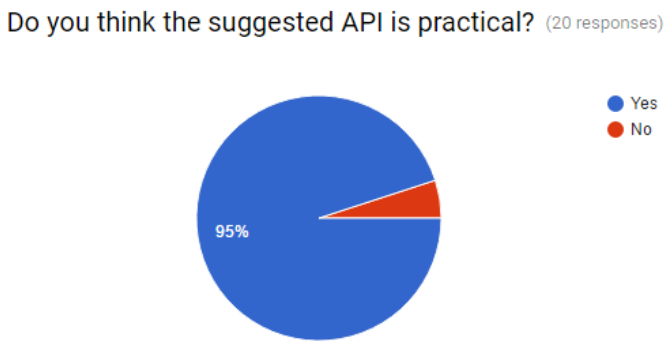


Fig. A.1 Question 1 graph.

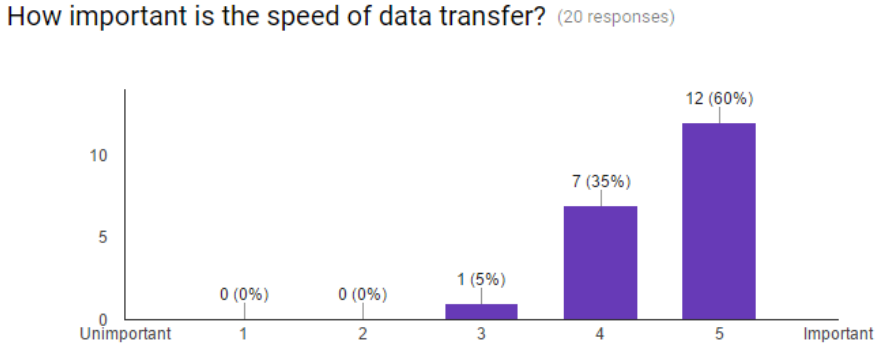


Fig. A.2 Question 2 graph.

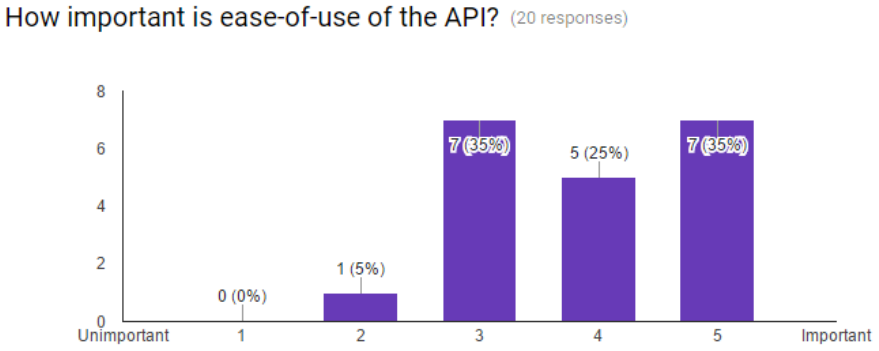


Fig. A.3 Question 3 graph.

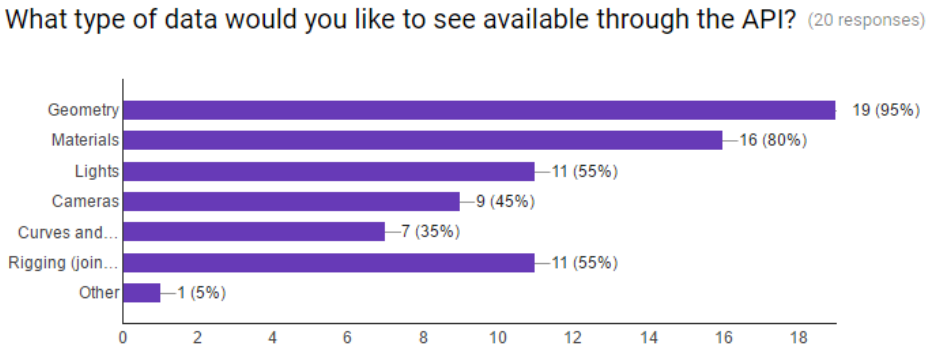


Fig. A.4 Question 4 graph.

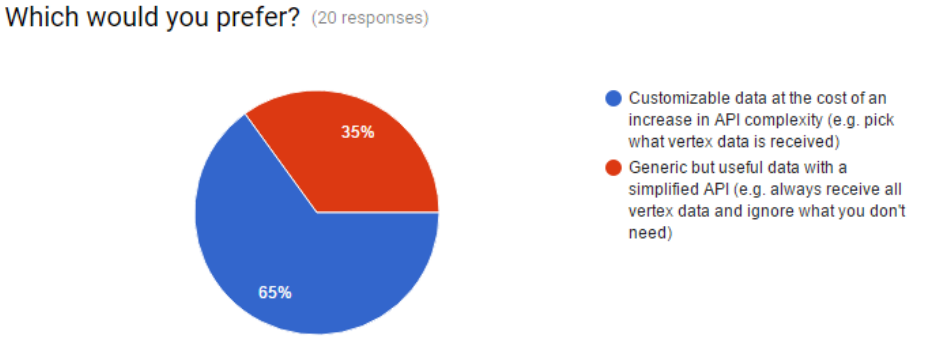


Fig. A.5 Question 5 graph.

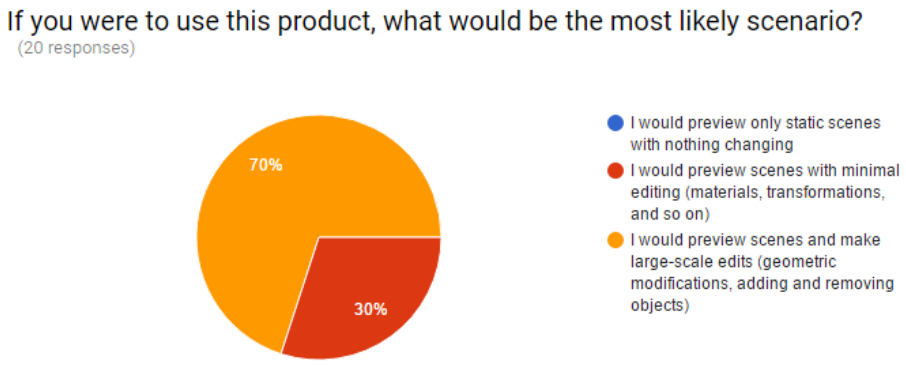


Fig. A.6 Question 6 graph.

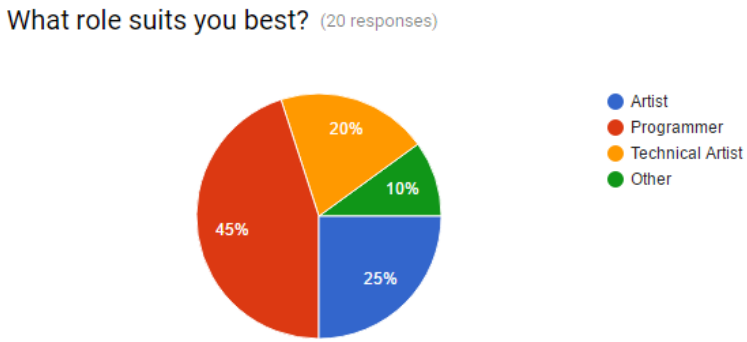


Fig. A.7 Question 7 graph.

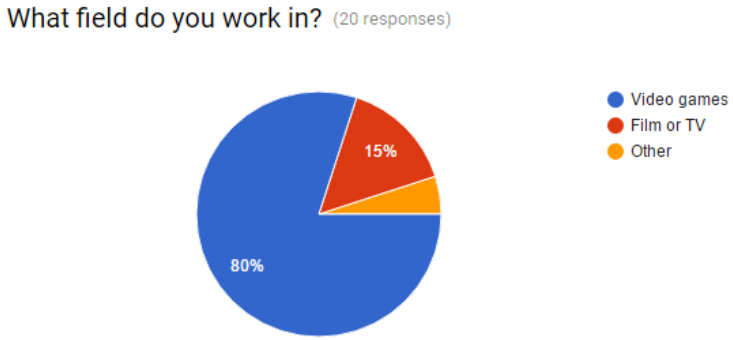


Fig. A.8 Question 8 graph.

Appendix B

Feedback Results

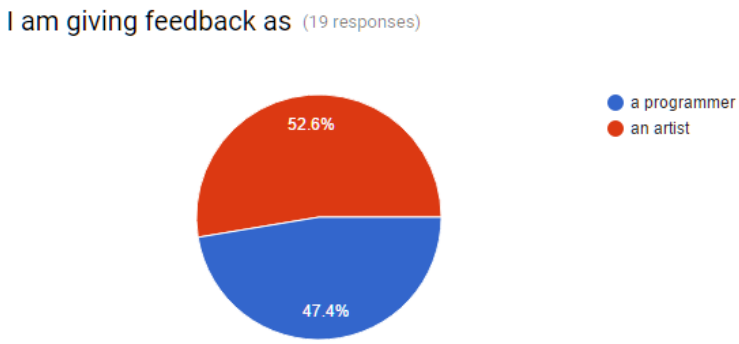


Fig. B.1 Feedback question 1 graph.

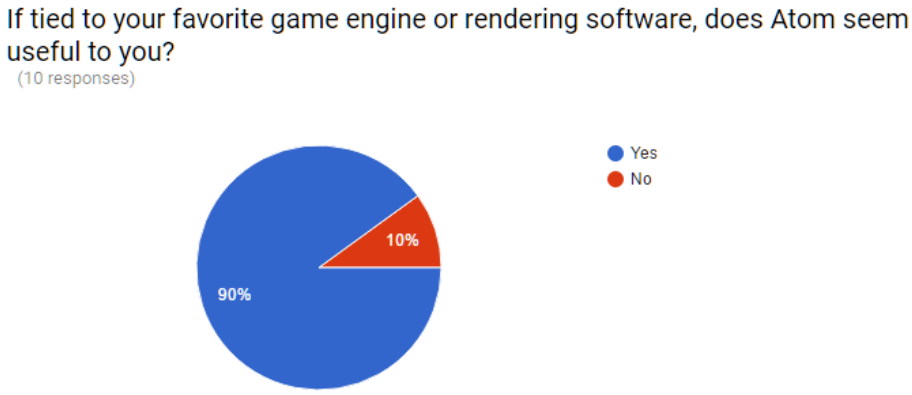


Fig. B.2 Feedback question 2 graph.

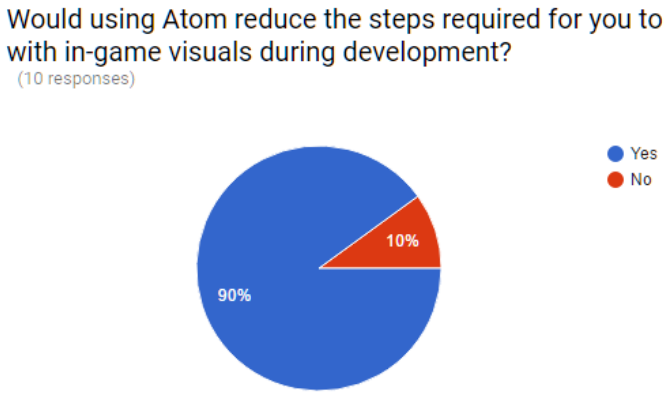


Fig. B.3 Feedback question 3 graph.

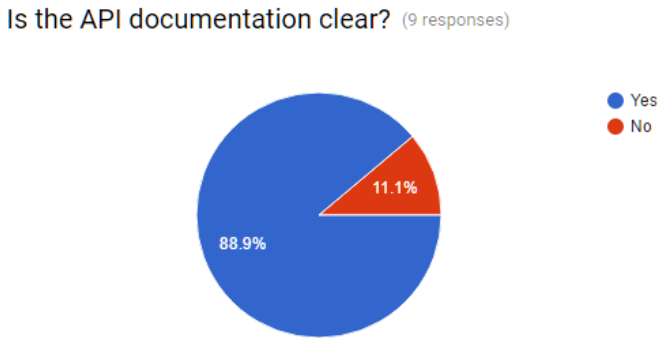


Fig. B.4 Feedback question 4 graph.

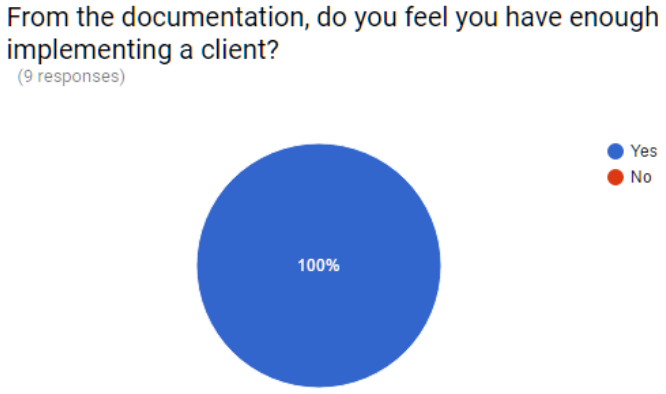


Fig. B.5 Feedback question 5 graph.

Artists' Comments

I think this would be very valuable in the gaming industry as you won't have to use the exporting step anymore. Workflow wise this is amazing. The only concern I have as an artist is that you will need a programmer to link up Maya and the game engine for you. Also, to make sure all the light and material connections (with all settings) work properly. It might even be necessary to create custom Maya materials, as the standard Blinn/Phong/Lambert might not have the same functionality as current day PBR materials in game engines. With this is in mind, Atom is not that easily accessible as an artist. It might be an idea to make an all enveloping Maya app/plugin that links up software, shaders and lighting for you. (If you're purely making this for companies, the previous point won't matter). Overall very cool work Daniel! But in the end, VFX > Games :D. Jef

---

Ideally, material parity would need to exist between the engine and maya. Substance designer/painter does this very well with engines like ue4/unity. Atom would mostly likely be more suited to a third party engine

---

What I find interesting is to see the lighting in an engine like Unity and Unreal. As an VFX artist its not useful in Maya as there are renders that update real time. The renders quality and amount of elements is important. If your render can't be used in an high poly environment with many lights and shaders, its not good. Obviously for separate object previews its perfect but only if its an unbiased render. I also think you need to show what else can be synched like particles, smoke, materials (specular, opacity, hair and ect.). Sorry I haven't seen the whole video, it wasn't that interesting and I think you got the point in the first minute. Amazing work man. Really good idea.

Looks great !

Watching your demo I can see how useful this application can be in all areas which is why I've checked both questions yes. I don't think I can say personally I'd use it for manipulating as much as you can for example the topology like in the demo. Or things like UVs as basic 3ds viewport does this fine for me. However with the ability to manipulate topology and UVs I'm assuming you could view and edit animations which would be extremely useful! I think it would be extremely useful for texture and lighting manipulation and as I am not really into animation that is most likely what I would use Atom for. As I said last time, incredible piece of software Dan!

Seems really useful

Seems like it would save a lot of time; it's an absolute nightmare when you have to keep updating something in one program and importing it into another program over and over.

I make these comments from the perspective of an animator, so grain of salt. I do not know how this tool in its current incarnation applies to me as I am generally not handling lighting or mesh work. Lighting is handled very differently from one engine to another. For Unity in particular (arguably one of the more used engines at the moment) this setup might be a struggle and may not capture a number of engine-specific workflow requirements. For example, you have to bake out lighting in a level for efficiency, or are often working with custom shaders that Maya doesn't support - what happens in these cases? Be very careful what you target with this tool - lighting in particular is something I would almost certainly want to control in an engine, as it effects all things and would be better to view and edit in context. What engines do you see this being used in? What's the ideal case where the setup and maintenance time would make it worth the quicker workflow? As a longer term consideration, how would you maintain compatibility between multiple game engines and Maya versions? Would this even be a priority? If working with a large scene file (say an environment) how nicely does this tool play with referenced objects? Is this intended to be an alternative to what Autodesk already supplies users with Stingray? http://www.autodesk.com/products/stingray/overview General Video Feedback: -If the functionality of the tool is where you want it to be (in terms of lighting etc) I would emphasize the lighting and materials changes on the *model* that is the focus of the scene, not the floor (it was strange to realize that was what was being adjusted in the helmet scene).

Programmers' API Comments

The documentation was useful in understanding the API, but I found the example source really helped bring all of the ideas together. So, possibly include more example code snippets in the documentation?

In the T̈ypec̈olumn, Lists don't mention what they consist of so it's not particularly obvious at first glance. My suggestion would be to present them as List<Type>, this makes the list type immediately obvious.

Nothing, but to be fair I know nothing about MAYA or gfx programming

I'd split it into two pages, Intro Setup and Message References.

It looks great but since developers can send anything to the Atom blackbox, it should be sending error codes where applicable. I would like to see these error codes in the documentation or at least what happens when you send faulty data.

Please change the each message back top arrow, it is a little confused.

I've never seen anything else like it, so it would be hard to say without using it.

Programmers' Comments

Praise Atom! On a more serious note, this concept is very interesting, especially from the perspective of projects such as my own. I am part of a large team working on a Triple-A title, and we have all sorts of resource pipelines in place to effectively manage our assets, and get them from tools such as Maya, into the game. However, this comes with the drawbacks of having to go through the process of exporting the asset from the tool, building it into the game's resource format, then building the game just to see how it actually looks, which obviously takes time. Being able to link the source tool such as Maya directly into the game, even if only to serve as a preview, would certainly help to smooth out asset development, and help to avoid costly mistakes in textures or materials. I could also see applications for the networked development of assets, possibly allowing artists to prototype and share ideas in real time, without having to stand over each other's desks.

I can see getting this to work with 3DS Max as well as offering plugin support for Unity and Unreal Engine will give this project major traction in being used in a professional environment as speeding up artist workflow is always one of the main driving forces behind software development in the games industry.

The API appears to be incredibly useful especially to a 3D artist. Having spent some time 3D modelling, exporting, importing, noticing errors, amending, re-exporting and then re-importing, over and over again I can see the potential Atom has to increase productivity and quality by cutting out that process. Atom is something that I would seriously consider integrating into my own game engine project.

The documentation has come a long way in a short time! Looks great now!

The API reads fine, I feel like I understood how the system works and the chronological ordering of the information makes me confident I could get Atom up and running. However, once I'm past that stage I feel getting quick reference for messages would be better served on a separate web page formatted for the task, perhaps with toggle tabs to clean things up. Once the client is setup, as a developer I wouldn't need access to the introduction every time I reference the docs. Even so, the api docs are readable and formatted well enough, I might just be nit picking. The project is bloody ace and as a programmer who works with a wall between us and our 3D artists I can totally see the use case for this!

Disclaimer: since I have no background in Maya or Maya's API I may be missing some knowledge that the ideal programmer might be expected to have. The documentation speeds through the concept of nodes. Evidently there are different types of nodes with different properties - I believe that a clear definition of this is needed. The second sentence of the introduction is ambiguous as it can be taken that cameras and lights are shapes but this conflicts with the third paragraph of the intro. Also I'm not sure whether it is possible to chain multiple messages in the same buffer and if so what syntax is needed. To make the documentation clearer I would recommend giving a concrete example of handling/receiving messages using As for the example project - if this is meant to be sample code for developers to download then I would like to see more comments for each *file and function* (you're giving people too much credit!).

Good job, proud of you did.

As I commented in the earlier stages this looks like a fantastic project, and should prove to be hugely useful during production.

Good use of threading. I appreciate the originality of the project.