Jupiter Ascending: Constructing Large Scale Environments

James Bird Maxim Fleury
Double Negative*

Abstract

Managing the complexity of large-scale environments is now a regular challenge in feature film visual effects. We will show how decoupling the representation of our layout from our geometry allowed us to improve our environment pipeline, and ultimately deliver the largest environments that Double Negative has ever built.

1 Production Requirements

We were required to build multiple CG environments (including Chicago and a refinery on Jupiter) containing a mixture of static and moving rigid objects. The layout for Chicago was based on a real world location but the exact path through this environment was not decided up front. The refinery was to be built from concept art, which depicted a 25km long industrial complex. In both cases we needed to be able to iterate quickly on different layouts. Both environments were assembled from a variety of assets and therefore contained a large quantity of repeated geometry. This geometry was modelled at different levels of detail such that it could be used for both hero quality and far-distance shots. Production required that these layouts could be consistently realised in a wide number of applications (including Maya, Houdini, Nuke, Clarisse, and Katana).

2 Underlying Data Model

On previous projects, we would construct a hierarchy of assets containing a mixture of transforms, geometry and references to other assets. Coupling this data prevented these hierarchies from scaling well because a large quantity of geometry could be encountered at any location in the hierarchy. This coupling also prohibited us from realising these layouts in applications without support for custom geometry formats. For Jupiter Ascending, we addressed these problems by separating the geometry from the transforms and references of our layout, such that we could iterate quickly on the layout in isolation. For the layout, we designed a simple file referencing schema that defines a hierarchy of transform nodes, where each node can either be classed as a grouping node or a referencing node. The grouping node can accommodate other nodes as children, and the referencing node can accommodate a named collection of references to other files on disk. At the leaf levels of the hierarchy, we had references to our geometry assets stored as Alembic caches.

3 Layout of Large Scale Environments

The nodes that formed our layout were guaranteed to sit at the top of the hierarchy, and we could therefore traverse these nodes without having to load any geometry. This allowed us to cheaply realise these nodes in a wide number of applications. We developed plug-ins for these applications that represented our layout as hierarchies of custom transform nodes. The artist was able to switch the display of these nodes between ”bounding box”, ”full geometry”, and where possible, ”GPU accelerated preview”. This allowed the artists to load only the parts of the hierarchy that they needed to work on, thus keeping memory footprint low. This was particularly useful for the refinery layout, which contained over 580 million polygons.

4 Working with Multiple Levels of Detail

On leaf-level reference nodes, we defined a table that mapped integer LOD numbers to named references. Starting at the root of the hierarchy, we then propagated an initial LOD value down the hierarchy. By default, the nodes inherited this value from their parent node, but were also free to override it. This numeric LOD value was eventually resolved at the leaf levels of the hierarchy. This allowed an artist to affect the LOD of multiple nodes at once, by setting the value on their parent node.

5 Rendering

To allow the artist to render a layout without having to load all of its geometry, we implemented delay-load procedurals for PRMan and Mantra that could traverse the layout at render time and emit further procedurals for leaf-level geometry files. We again exploited the separation of geometry and layout, by cheaply traversing the layout up-front in order to identify many repeated instances of the same geometry file. We were then able to emit these as instanced geometry to the renderer. We also stored attributes in the layout that were emitted at render time and evaluated in the shader, such as UV offset for varying the look of each asset instance.

*e-mail:{jsb,mnf}@dneg.com

SIGGRAPH 2015 Talks, August 09 – 13, 2015, Los Angeles, CA.
ACM 978-1-4503-3636-9/15/08.
http://dx.doi.org/10.1145/2775280.2792579