Dependency Graph Design Overview

Joshua Leung

aligorith@gmail.com

Version 1.0
July 2013

Summary

This document aims to provide a user-level overview of the new depsgraph design. We look at the key features and components of the design, and how these relate to the the common problems that frequently encountered with the current system (as well as possible extension points for future developments).

Key Problem Areas

Based on several years worth of bug hunting/tracking experience, various todo-lists of depsgraph related problems, and analysis of the types of problems we have, the problems we face in the dependency graph and evaluation systems all fall into just a few categories (in terms of technical problem areas):

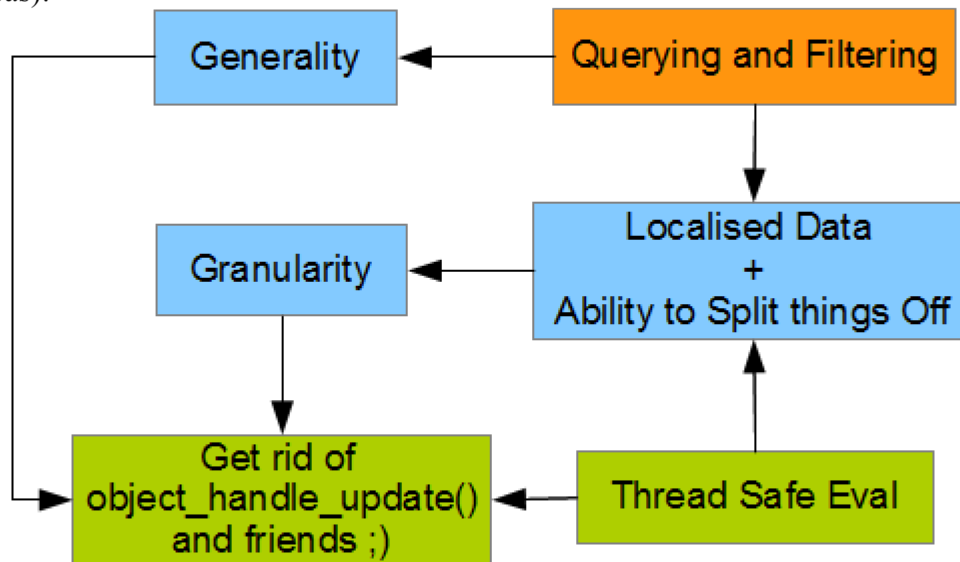

***Figure 1**: Diagram showing the set of key technical targets, and how these depend on each other (i.e. A → B = "A depends on B")*

The colour coding used in the diagram is used to indicate the component/subsystem that each of these areas belongs in, relative to the

- **Orange** = ***Depsgraph External API***. This is used by various tools (e.g. any operators used to edit stuff = including transforms, add/remove data or dependencies, etc.), RNA callbacks (e.g. bone renaming), code for baking sims/motion paths or for drawing ghosts, and possibly exposed to scripters too.
- **Blue** = ***Dependency Graph***. These mainly concern the structure of the dependency graph (i.e. what nodes are in it and where data is stored durign evaluation), but also concerns what low-level operations can be performed on depsgraph instances.
- **Green** = ***Evaluation Engine***. These are part of the Task Scheduler (i.e. performs evaluation)

To avoid confusion due to terminology differences, let's explore what each of these mean, in terms of functionality or capabilities that have an impact on users.

(**Note**: You may want to skip "The Long Version" bits in your first read through if pressed for time)

Generality

In Short:

1. All ID Datablocks can be represented in Depsgraph (if there are dependencies needing this)
2. Non-ID Datablock things can also be represented in Despgraph (e.g. bones, rigidbody sim, sequencer eval, etc.) as needed to ensure correct evaluation orders

The Long Version:
The current depsgraph we have is effectively limited to representing two types of data: "Objects" (i.e. things in *bpy.data.objects*) and "Object Data" (i.e. *ob.data*). It knows absolutely nothing about other types of datablocks since it has a very object-centric view of the world (i.e. all the "needs update" tags are stored on objects only!). But, we have many other types of datablocks in Blender, such as Materials, Textures, Scene/World data, etc, which can all have drivers (and a few other relationships too in some cases) set up between them or objects but are not updating correctly nowadays (or only "seem" to work as we've hacked them to get run all the time).

In order to be able to truly achieve the "everthing animateable" (or more precisely, "everything able to be animated can also be driven or used to drive something"), the types of things included in the depsgraph cannot be limited to objects and data directly attached to them. Otherwise, even if users are able to create drivers for these things, when it comes to evaluation time, none of these items can be tagged for updating or evaluated in the correct order. ***At the very least, we need to ensure that all ID Datablocks can be added to the Depsgraph so that all relationships between these can be represented and evaluated correctly.***

A secondary goal (though this overlaps slightly with the "Generality" issue) here is that we should have ***the ability to schedule up non-ID datablock data or evaluation steps that need to be evaluated between other datablocks being evaluated***.

- One of the prime examples here is the Rigidbody Simulation step/evaluation (i.e. this needs to be run after objects which participate in the sim have been evaluated, but before other objects which depend on the ones in the sim are evaluated; otherwise, additional relationships cannot be set up to occur as a result of the simulated behaviour).
- A second example here is for handling/hosting animation evaluation for "character + props", where it'd be nice to be able to include the props in the same action as the character animation (Note: the exact way we'd go about this are still unclear, since in this case it could end up being that we actually end up requiring users to use/create a proper datablock dedicated for this purpose). Regardless of how this is done, at some point, the problem of where to put the animation evaluation for this (i.e. since there needs to a "root" entity of some sort for the paths to be evaluted relative to) will end up meaning that something has to happen on despgraph level for allowing this to get evaluated.

Granularity

Note: This aspect has already caused a bit of controversy (as was to be expected). Understandably, if taken too far, the implications are quite far reaching, ranging from the changes which would have to be done to many parts of the codebase, to potentially creating graphs that had too many nodes in them to work well in practice. Having said that, as proposed later, there is a middle ground where we can get the best of both worlds here.

In Short:

1. Evaluation of bones, drivers, and objects can be interleaved to avoid pseudo-cycles
2. The mechanisms needed to support this should be general enough to extend to any data, so that we are not stuck again in the situation where we one day need greater granularity for some block which has now acquired a whole host of subdata with complex relationships between them, but then find that this will be extremely hard because all the existing cases were special hacks.
3. Having the ability to have smaller chunks can be helpful for some of the other goals we have (i.e. it is essential for optimal multithreading for example)
4. A little bit of complexity in the backend to keep things simple for users is still a good tradeoff

The Long Version:
It is necessary to be ***able*** to have sufficiently fine-grained nodes (representing data and/or evaluation steps) to prevent pseudo-cyclic relationships from occurring (i.e. this doesn't mean that all nodes would have to be fine grained). By “pseudo-cyclic” relationships, we mean that even though some relationships are obviously non-cyclic from our perspective, to a computer they would look cyclic if it didn't have sufficient “resolution” to tell that the dependencies were in fact pointing at different things.

Apart from the lack of generality (i.e. non-ID datablocks not updating at all or as required), *the problem of pseudo-cyclic dependencies is perhaps the most restricting and frustrating problem from a rigger perspective*. (Sure, the problem of not having localised data is bad, but there are low-tech workarounds that riggers can employ in those cases like simply manually copying out that data. But with the Generality + Granularity problems, these are fundamental architectural “archilles heels” that simply cannot be worked around, as the way these are handled outside the control of users internally by the software).

In our current depsgraph system, we're hobbled by two nasty facets:

1. **We have only two types of very crude/coarse-grained blocks** to describe the types of data present, and for performing/scheduling evaluation operations on those. This means, it's an all or nothing on each of these blocks when it comes to knowing if we want to evaluate them, and also which parts of them need evaluating. Also, practically everything about the system is hardwired to assume that these are the lowest-level primitives in the depsgraph, when in fact, relationships can exist between entities much much smaller than this contained within these coarse containers.
2. **All relationships between these blocks are reduced to “aggregate” relationships** (i.e. if A depends on B, and B is touched in any way – even if only 1% of B is touched – then *all* of A must therefore be dirty too.) There is no subtletly here for describing the fact that: only certain parts of A and B may depend on each other; or that in fact C – which also depends on and influences A – can be safely interleaved into the evaluation process for A.

By making the lowest-level executable primitive/atom in the depsgraph a datablock, we're precluding properly handling all the sub-datablock datatypes that users can create complex networks of dependencies between. Experience so far says that everytime you go, “Bah! Users won't need that '1%' case for flexibility and crazy setups”, the day you release your system, it is exactly that '1%' case that turns out to be either the blocking-link or the thing they really need or end up trying to use (often as the first thought/priority!).

To put this into context better, let's firstly look at the kinds of relationships we have now which

could benefit from being true “lowest-level” nodes:

1. **Driver** – Performs operation/computation to link value of *Property-1* with value of *Property-2*. Whether we exactly need to represent the properties themselves really depends on the situation – in many cases we don't, but in some cases this is needed to properly diffentiate between related/unrelated properties (e.g. location vs rotation). In any case, if knowing the exact property links isn't needed, this extra-fine info can be erased...
2. **RNA Update callback** – On some properties changing, it is necessary to execute a particular flushing operation on the block that hosts the property (e.g. Mapping node needs to flush the user-visible transform properties to an internal 4x4 matrix). Thus, we need to properly represent such update callbacks in the graph. (Mere tagging other nodes for updating though should be done a separate way, to avoid those infinite loops we got...)

One-level up from this, we have slightly larger chunks such as **bones, constraints, modifiers, particle systems (?), and other sub-object data**.

- With constraints, modifiers, and particle systems, it's true that perhaps for now we can just treat those stacks as “black box” units from an evaluation perspective (since those currently only form linear dependence chains anyway that you can't readily multithread on a structural level), and simply have those as the node representing those cases.
- However, with bones, the situation is actually quite complex, and there are many technical considerations (which I'll skip here for the sake of space, apart from saying that it's not as simple as saying “bones can be evaluated at scene level alongside objects” which implies certain implementation approaches which are frankly impossible/incompatible with some things there).
- I will also note briefly here that in the event that we go about implementing **“node-based constraints”** and/or **“node-based modifiers”**, the evaluation processes for these as they stand now have structural similarities to how the bone evaluation process goes (*init* / *subgraph or self* / *cleanup*), which, if we took advantage of this fact when setting things up now, we'd manage to fit in nicely into our architecture later.

Finally, it's fair to say that for **some datatypes (e.g. perhaps lamp and camera)**, there really isn't a hell of a lot of special sub-data going on within those datablocks. So, those could probably say big and blocky for now (at least initially).

Other considerations here:

- **Optimal multhreading requires that you have heaps of small tasks that can be scheduled up.** Small tasks can potentially be distributed across many processors (even when the coder may not have realised that potential) Large tasks though almost certainly block up one core while others sit idle, and dump full burden of optimising multicore performance on coders.
- **Having a more nuanced picture of the relationships can mean that when “splitting off parts of the graph” (see later), you're able to only pick and choose the parts you need to duplicate (and/or evaluate at each step)**. Whether this is for selecting just the nodes which need to be updated when transforming a bone in the viewport. or when baking sims (only parts of objects which collaborate end up needing to be evaluated at each step).

Localised Data

In Short:

- Multiple copies of data != Evil. Focus for now is maximal flexibility/correctness of evaluation, and not on premature optimisation.
- For multi-user data (e.g. several objects using the same mesh data or material, duplicators,

proxies), evaluating each instance shouldn't result in changes the base data (that's shared between instances). Instead, each instance should work on its own copy of the data.
    - An optimisation we can look into later is making it so that the “shared” parts are shared between instances and only computed once, but only if it's safe to do so.
- Baking/background operations which need to evaluate shared data in a different state/configuration should perform their evaluation on separate copies of the data (i.e. not the same set being drawn/rendered/edited by users now)

The Long Version:
One of the big problems with the evaluation system currently is that settings/state/results are all contained in datablocks – evaluating the state of a datablock involves reading the settings from that datablock and then writing the new state/results into separate compartments there. Although this is quite a nice and simple conceptual model, it falls down as soon as you have more than one user of a block of data – especially when those two users need their instance to be in a different state (e.g. ob1 tries to make the mesh red, while ob2 tries instead to make the mesh blue)!

The simplest and perhaps most foolproof solution is to simply make copies of all data you're trying to evaluate. As far as thread safety is concerned, it is **the only** practical solution if you're after both flexibility and simplicity. That's because, with thread safe code, you need to ensure that each piece of data is only ever being used by a single thread at once: regardless of if you're only reading the values from one thread but able to read/write in another, only one of them should ever have access to that piece of data in memory at a time, or else you're in for any manner of non-deterministic deadlocks and other weirdness that'll cause you hours upon hours of pain when you least need it.

Short of having duplicated data, there's no other way around it, especially without wasting heaps of developer time debugging increasingly weird and rare “threading bugs”. The basic consensus that's emerging nowadays regarding concurrency is this: sharing-based threads are problematic; fully separated solutions (e.g. processes) which only communicate using very well defined protocols instead of sharing data directly are the way forward. That, and not relying on developers to have to hand lock and unlock stuff.

There are several different cases here that we need to deal with. They are not the same, though the types of things we need to do in both are similar: that evaluation must not happen on the actual copies of the datablocks users interact with.

1. **Multi-User Data** – Datablocks referenced by multiple users (i.e. materials or mesh data linked to many different objects), or duplicators (i.e. dupliverts/dupliframes, dupligroups, particles).
   Key Point: *Multiple instances of data may be present, with/without overrided properties or different time evaluation (e.g. from time offsetted animated) on some of these.*
2. **Proxies** – Datablocks linked in from other files (i.e. library blocks), or perhaps instances of data referenced elsewhere, except we allow users to apply edits on top of a local copy while applying changes on top.
   Key Point: *Overrides performed on top of existing data.*
3. **Background Evaluation Operation** – Baking operations for simulations motion paths, or evaluation/drawing of onion skin ghosts, where it is necessary to recalculate a given object/bone's data at different points in time. To do this though, we also need to recalculate anything it directly depends upon.
   Key Point: *Must not disturb the data user is currently editing (otherwise, we get unposable rigs/objects for example)*

Ability to Split Off/Duplicate Subsets of Graph

In Short:

1. When performing Background Evaluation Operations, it would be helpful/useful to be able to grab just the *minimal set of nodes* you need to evaluate to ensure you've correctly updated the stuff you're interested in.
2. For dupligroups or characters imported (which don't really interact much with anything else), you want to keep these separate from the main graph (or at least evaluate them in isolation), as then you can have multiple instances of these without underlying shared data problems.
3. For thread safety during main eval, you want to be able to isolate out groups of unconnected nodes that can be evaluated separately from the rest (though this could also be done via task scheduler). Anyways, this is just an additional option for how to handle this problem.
4. When using **Depsgraph External API**, sometimes it's helpful to get a copy of the graph with just the things you're interested in (and all other parts removed). In other words, it's useful to be able to filter a graph and get a subgraph from that for other tools to use.

The Long Version:
There's probably not much more to be said about this than what I've said already. In addition to having local copies of data, we may want to split up/duplicate some of the nodes themselves for easier/faster processing in some cases.

Fine granularity of nodes/data steps comes in handy here, as we can just grab the parts we want when splitting up subsets of the graph, thus reducing the amount of execution time that would need to go into evaluating tasks which don't contribute to affecting any results we're interested in getting from the graph subset, and/or also reducing the amount of data we'd need to be copying across too...

Querying and Filtering

In Short:

1. Tools should be able to ask the depsgraph questions like: “What objects/datablocks depend on this thing I'm editing?”, or “If I'm updating this piece of data, what other objects/drivers/constraints do I need to update to correctly evaluate this?”
2. Alternatively, it should be possible to get the depsgraph to apply/clear some tags from nodes which fit certain criteria.

The Long Version:
A lot of information about relationships between various pieces of data in Blender are stored in the depsgraph (currently not enough, but we still know a lot). This data though is by and large unable to be accessed by other parts of Blender in any way, as the datastructures to grovel around under the hood are kindof private/internal to the dependency graph implementation.

Now, we don't exactly want to provide direct access to this (although at least one tool does do this – motion path calculations to be precise – as the performance penalty of not fiddling with the graph structure were severe in production rigs) as that's just bad practice (i.e. read up on the principles behind Encapuslation and Abstract Data Types), but there are clearly situations where tools could benefit from being able to work with this information (instead of rolling their own partial solutions, as some Python scripts used in production reportedly do).

Hence, it would be great to be able to have this kind of capabilities available for developers. In comparison to many of the other goals, this one is slightly lower priority if we're pressed for time in the initial stages/releases of this stuff. But, this is an integral/important capability that we must

account for when coming up with "the overall vision" for where the dependency graph is going in the long term.

Evaluation Scheduling

In Short:

1. Multiple nodes in the depsgraph should be able to evaluated at once (provided that doing so doesn't violate any dependency relationships). That is, multi-threaded evaluation of the graph should be possible.
2. Evaluation of individual nodes needs to be threadsafe
3. Where necessary, processor-intensive nodes (e.g. geometry modifiers such as subsurf, etc.) can be evaluated in a multithreaded, in addition to any graph-topology-based multithreading
4. No datablock-level functions for delegating/dictating order that sub-data gets evaluated in. All tasks that need evaluating are separate nodes in the depsgraph, with relevant relationships between them to allow scheduler to determine correct order at runtime

The Long Version:
With the current evaluation engine, performing updates is a matter of entering the evaluation process via one of the *BKE_scene_update()* functions (these are one of the functions implicated in "*object_handle_update()* and friends"), which act as monolithic fixed-order pipelines for the process of evaluating data in the scene. At various stages, different parts of the scene's data is looped over (the order that items are encountered is determined by an earlier topological sort of the items in the list by the depsgraph), and only the items which have been tagged are updated (by recursively calling the appropriate stuff).

The problem with this approach is that it is relatively hard to integrate any new steps into this pipeline (e.g. rigidbody sim) that get performed interleaved with the data that it actually needs to affect – developers must try and figure out a nice arrangement themselves, or hack around a bit until something works "well enough" for common cases. Similarly, it restricts the types of dependencies situations which can actually take place (e.g. object ↔ scene relationships can be problematic). All of these problems though overlap with similar ones from other parts. Finally, we'd still be stuck with what is essentially still a sequential pipeline that bunches up groups of tasks which can be run in parallel, but which probably wouldn't actually be able to move on to the next bunch of tasks that readily, leading to some downtime between bunches.

Basically, all problems regarding figuring out exactly how to execute the updates while preserving relationship constraints and with optimal resource allocation is perhaps something best left for "the scheduler". If we adopted such an approach (i.e. graph of "evaluation tasks" mindset), the dependency graph + evaluation engine we'd have would effectively be a little interpreted language/VM/callgraph (just that some of the opcodes there may be a bit "heavy" to deal with), which leaves the door wide open for "graph optimiser" techniques from the world of JIT-compilers / interpreted languages / etc. to be brought over by some enthusiastic dev in future who has experience with these things. Failing that, at the very least, we still have the option of doing what Dreamworks did, and replace our own scheduler at some point with the TBB + CnC libraries, which have been proven to work in such scenarios as long as our own architecture has enough similarities to be ported over to that approach.

Core System Architecture

This section describes the overall architecture of the new Dependency Graph and Evaluation System. From this, it should be possible to get a general feel for the components which make up the system, and how it all fits together/works. Some specifics of the implementation details (i.e. steps which may prove to be necessary to perform) may not be shown here, as many of these won't actually become clear until later on when actually implementing the system (and/or porting over existing code).

Summary of Key Components

As mentioned briefly, there are basically three components to this system: **Depsgraph API**, the **Dependency Graph**, and the **Task Scheduler** / **Evaluation Engine**.

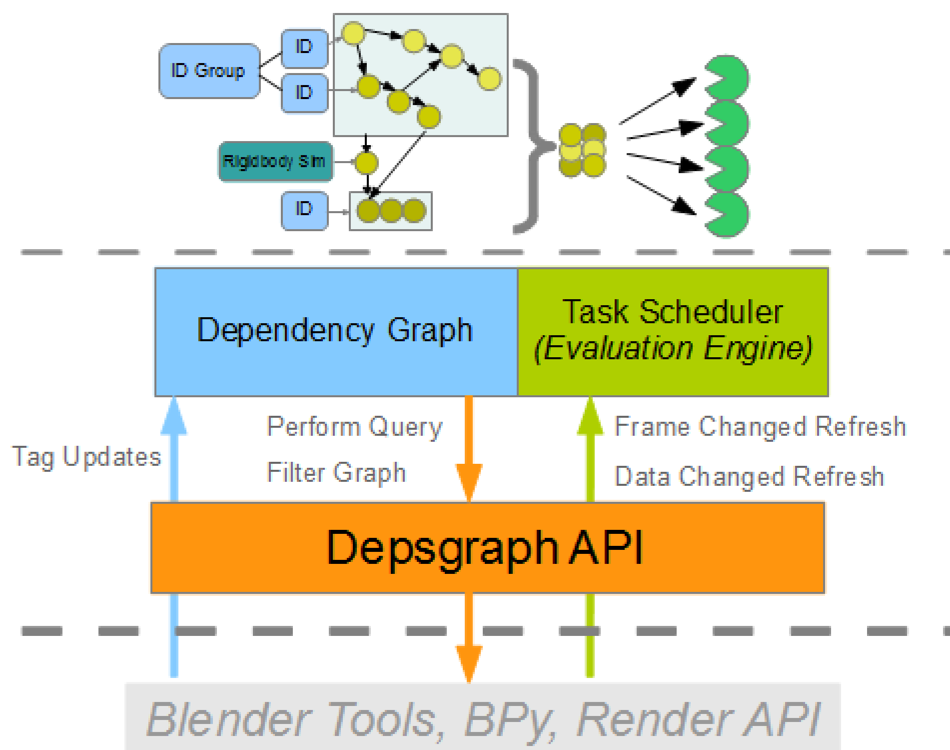

Figure 2: *Overview showing components of Depsgraph and Evaluation System*

As briefly mentioned at the start, the three components of the Depsgraph and Evaluation System are as follows:

1. **Dependency Graph** – This is the *datastructure* where relationships between entities are represented. It consists of *data/outer* nodes which mostly correspond to datablocks or other entities, and *atomic-eval/inner* nodes which are the atomic evaluation steps/operations which need to take place in order to evaluate the data.
2. **Task Scheduler / Evaluation Engine** – This is the *machinery* which takes the *atomic-eval* nodes from the depsgraph, queues up those which have been tagged for evaluation (taking into account the various relationships they must obey), and feeds these to cores/threads as they become available. The exact way in which the queueing is done is one of the key things which ends up controlling ultimate performance of the system (though the way the atomic operations work also matters a bit), and is where multithreading on graph level comes in.
3. **Depsgraph API** – This is the API through which the rest of Blender interacts with this system. Apart from the existing tagging and refresh-triggering entrypoints, the key new

addition is the ability to perform queries on the depsgraph (e.g. all ancestors/dependencies and decendents/dependents of some entity), and get subgraphs or lists of matching nodes out from the system. With the added querying capabilities, it should be possible for tools to be able to do more with the info contained in the depsgraph (e.g. performing necessary updates in a more efficient manner for instance, or having a way to safely obtain subsets of the graph for background processing).

Nodes in Depsgraph

Perhaps one of the key questions concerns what exactly the nodes in the Depsgraph are.

This issue has been covered in depth in some other articles I've written on this matter so far, so only a brief summary of the key concepts will be presented here (see Figure 3).

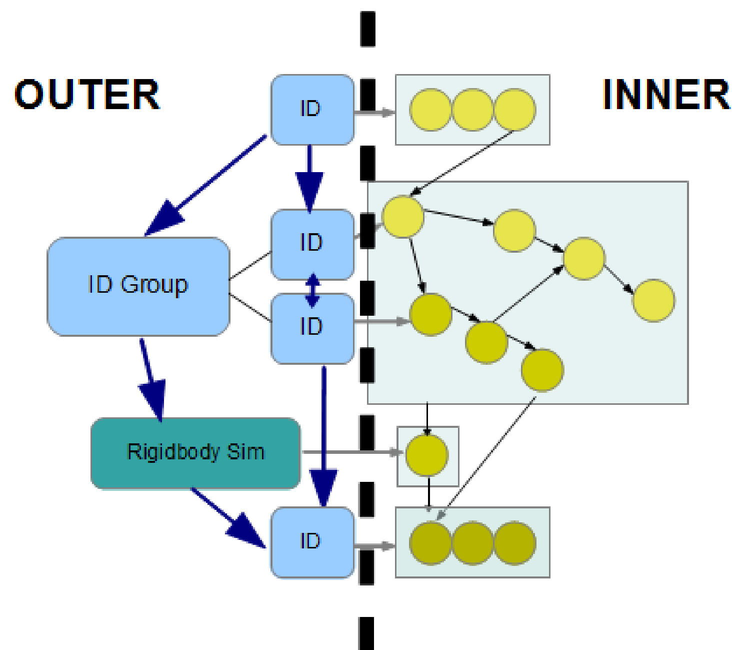

Figure 3*: Diagram showing some depsgraph nodes from a hypothetical setup. This example was constructed to try and illustrate a few of the key things about how this would work.*

The Dependency Graph part of this system is essentially divided into two sets of nodes: the "outer" nodes and the "inner" nodes.

- **Outer nodes** are used to represent and *keep track of dependencies on a broad scale* (namely, in terms of datablocks and other significantly large entities that users will consider "a thing" that would be interesting to see the dependencies between). For the querying API and many other purposes, the outer nodes are what you want to be dealing with (and all that you really need to deal with or see), which means that things will not seem any more complicated than what they are already.
- **Inner nodes** are used to actually *keep track of relationships on a fine-enough scale* that most pseudo-cyclic situations won't show up as such. To be precise, the set of inner nodes (i.e. all the nodes on the right hand side of the diagram) defines the full set of evaluation steps that can be performed/executed in the scene in response to tagged changes. Each individual atomic node here is an evaluation step that doesn't really contain any others. This representation is really optimised for computers to go through and algorithmically figure out ways of executing/scheduling up the correct evaluation of this setup quickly. Most

humans don't need to know/care too much about the details of this, though we may eventually consider making it possible for advanced TD's to hand-tweak this graph to get out extra performance (Disclaimer: this is highly unlikely to happen at this stage, but I'm just putting the idea out there in case we need it one day!)

As can be seen, there can be many different types of outer nodes. There are several reasons for the ones this, and the examples shown in the diagram illustrate some of the main classes of these things:

- **ID Nodes** – These are simply ID-datablocks. Nothing new. The only thing to note is that these are definitely nodes representing ID blocks and not Objects only. Also note, that we don't have the Object + Object.Data type nested tags anymore – each ID block is its own entity here, and any "obdata depends on object" relationships must be specified as genuine relationships.
- **ID Groups** – These are clusters of ID Nodes which, from a coarse-granularity point of view, are mutally dependent on each other (i.e. the form a "strongly connected component", or "there is a (pseudo) cyclic dependency lurking between them", etc.).
    - When building the graph, we start by just adding ID Nodes into the graph at top-level, but if during this process we find that the dependencies between two nodes are quite tightly intertwined, then we isolate the offending nodes and put them in a new group (or add the newly offending node to an existing group).
    - ID Groups have a "headliner" section which keeps track of the ID Nodes that live within it. That's useful for filtering purposes, where this can be used to help identify which group is affected.
- **Transient/Non-ID Blocks** – These are evaluation steps which aren't datablocks, but which need to be evaluated in between datablocks/groups for whatever reason. Examples include the Rigidbody Simulation step, Sequencer evaluation, and possibly the character-group animation system (whatever that is going to be like!)

Now, attached to each outer node is a subgraph of inner nodes. This subgraph is a fully self-contained set of evaluation steps which can be assigned a processing core (or processing cores) arbirtarily. During the graph-building stage, there is a "*build_subgraph()*" or similar call on the outer nodes which tells them to populate their little subgraph of atomic operations. As a basis for this, each ID type (or similar) will have some statically-defined tables of callback references which they'll use to instantiate their little graphs (which will be automatically hooked up based on their ordering in the tables, potentially with ability to override this if need be).

As far as representing dependencies go, at this stage, it makes sense that we may want to keep track of the dependencies both on outer-node level in addition to at inner-node level (where it's a must). Although these relationships could in theory be computed as needed (probably only when queries are run), doing so could be slightly costly. For now, let's assume we've enough RAM to hang onto all of this! Other important things to note from the diagram is that we can still have relationships between outer nodes of various types.

Scheduling Nodes for Evaluation

I'm only going to briefly touch on this topic. Basically, we just feed the evaluation graph of inner nodes to a task scheduler, by taking only the nodes which need to be updated, and/or also doing some pre-run cleanups to prune any dead branches not producing or contributing to the end results.

In order to speed things up, it is very likely that we can make use of "static rules" for basically gathering up the set of nodes which will need updating in response to certain properties being manipulated by users (i.e. eyebrow control being moved). To save time, we maybe only build and

store these rules when they get invoked the first time, instead of exhaustively building every one of these first.

Another interesting thing to explore is keeping track of how long each atomic node took to execute, and potentially feeding this info into the scheduler during later run throughs to indicate that tasks shouldn't be backed up being a certain slow one (for example). Then again, it might be even better to show this particular info to users somehow, so that they can optimise their setups...

One important thing to watch out for is that we don't end up feeding ancestors and descendents onto the cores at the same time (i.e. thus violating the expectation of each node that all its ancestors have been correctly evaluated already). There are probably good ways of doing this already, but if not, an initial proposal I have here is a "tiered queue": we basically partition the graph starting with sources (i.e. no inlinks), and build shells of nodes of depth *d* from the sources, until we reach the sinks (i.e. no outlinks); once we've done this, we know that it's safe to load anything from the same shell onto the cores in parallel, and only move onto the next shell once there are no nodes left in a particular shell. Sure, this will mean some cores are idle at times, but working around this then becomes a rigger problem (provided we eventually start providing some performance-monitoring facilities).

Data Instances Problem – State vs Results

This is truly a very sticky issue to solve, complicated by a few different overlapping problems.

For what to do with settings/RNA type property stuff, the following heuristic is probably a good starting point:

> **if** Data/Setting is Visible/Editable by User:
>
> > Can store state of properties datablock
>
> **else**:
>
> > Results should be stored on copy of datablock or separate datastore (as needed)

Other parts of this problem we can tackle in other creative/analytical ways. For instance, mixed into our datablocks and datastructs is often a mixture of settings/state, and results.

- **Settings/State** – Properties which users can edit, animate, drive. These tell Blender how you'd like to manipulate whatever you're dealing with.
- **Results** – Where computed results of operations go. Some of these are nearly cleanly separate now (though practically live on the datablocks they've been computed for) – e.g. *DerivedMesh* (geometry with all modifiers applied) – or partly separate – e.g. *DispLists* for Curves (?) – or are somehow mixed in beside the settings they affect – e.g. *obmat*, *pose_mat*, *chan_mat*, *arm_mat*.

While the things above can easily be abstracted out into "InstanceData" holders (i.e. we dump geometry result data or "transform matrix" result data into such containers), not everything is quite a straightforward.

Part of the dilemma here unfortunately is the anim system, which by necessity needs to write to the datablocks to set the correct current state for users to work from, but at the same time, it's dumping its results on the datablocks directly to do this (via RNA). Hence the heuristic above. However, when this is being applied to instances (multi-user stuff, duplicators, proxies), then we'll probably need full copies of those datablocks to let RNA have full natural read-write access on the data (Note: I'm not sure at this point how well RNA will hold up thread-safety-wise even when dealing with these separate copies; Hopefully there will be no major problems).

For “true” proxies (probably something well out of scope for this/depsgraph project, and a significant project on its own), we'd probably need an RNA-based system (or so) for binding up “Frakenstein-Datablocks” that can inherit data from a specified base (or chain of bases), keep track of overrides that it applies to that data – this would form that proxy instance's “instance data” component that gets saved into files – and then correctly handle these aspects. A few years back I had a little concept sketch regarding this very type of system, though I can't seem to find it anymore (though at the time, it didn't have any specific implementation ideas attached to it).

Preliminary Breakdown of Work Required

- Design and implement core depsgraph node datastructures
  - Depsgraph “graph” container
  - Base node type
  - Common nodes:
    - *outer* – base, group, data-blob, …
    - *inner* – atomic step, bunch of steps (unexpanded), …
  - Basic depsgraph-logic – mainly just repurposing the existing code to make it work for our needs
  - Mechanisms for handling instance data
- Depsgraph building stuff
  - Code to create the required nodes
    - Scene traversal
    - Node creation/finding
    - Suspected-cycles isolation (group creation/growing)
    - Resolution of group graphs – explode and interleave sets of atomic ops
    - Duplicator/Proxy special handling
  - Convert existing evaluation functions over to evaluation functions which can be attached to atomic eval nodes/tasks
    - Datablocks without subdata will initially just be ported straight across as the simple monolithic blobs they currently are
    - Evaluation of each bone, driver, and relevant RNA callbacks will each be atomic operations. Pre/post steps for armatures will similarly be atomic.
    - Object transform, modifier stack, constraint stack (?), etc. will each be “atomic” operations initially. Further breaking down can be done if/when we discover that this isn't good enough (and really, could be done at any stage in future, as all this is dynamic runtime stuff)
    - Remove the old hardcoded evaluation handlers/loopers
- Filtering + Querying API
  - Low-level filtering API and datatypes
  - Higher-level filtering API
  - Graph subset building technique
- Task scheduler
  - Way to extract the evaluation graphs (graph of inner nodes) from depsgraph – probably via Filtering API
  - Way to prune evaluation graph (i.e. drop unused branches for example) – *OPTIONAL*
  - Way to parition graph into tiered queue (or some other mechanism for ensuring that we only attempt to parallelise groups of nodes which won't depend on other nodes in same group)
  - A basic single-threaded scheduler as fallback and/or initial starting point, to keep things easy to manage (unless other devs are willing/interested at this point in actually building a multi-threaded scheduler which could work).

Useful Links

- http://wiki.blender.org/index.php/User:Aligorith/GSoC2013_Depsgraph