Dependency Graph and Evaluation Engine: Design Doc for Initial Implementation (3rd Attempt)

Joshua Leung

4 October 2014

Where we're at, and what we want/need to accomplish…

Firstly some definitions:

- **Dependency Graph** = The thing which keeps track of relationships between stuff, and which we use to determine which items need recalculating
- **Evaluation Engine** = The code which performs recalculates items/data which need updating (as determined by the Dependency Graph)
- **Depsgraph** = For this document, this is the Dependency Graph + Evaluation Engine

During the GSoC project last year, I worked on figuring out what the Depsgraph needed to be like in order to resolve all the various issues that we'd be running up against with the old system. The key points to note from that work are:

1) The ***Depsgraph*** must be able to be used for **more than just Object + Object-Data** relationships
2) The ***Dependency Graph*** must be able to ***represent more granular entities*** (e.g. drivers, constraints, modifiers) than just Object and Object-Data ID-blocks, and the relationships between these.
3) The ***Evaluation Engine*** must allow us to schedule up complex **webs of interactions between *granular entities***.
    a. For example, this includes being able to schedule up drivers and "other" objects to get evaluated while the bones in an armature haven't all been evaluated yet (provided that all the dependencies for those drivers/objects have been satisfied). In other words, what Ton calls "***Mixed object-bone graph***"
    b. For ***physics systems***, this means being able to ***schedule the simulation*** to be calculated ***after all collision targets*** have been evaluated, but ***before any objects which then use the results*** of the sim. This opens up the door for rigidbody-driven drivers/constraints/modifiers/particle emission for instance, and potential interactions between blackbox-simulations.
    c. By introducing "***time source***" nodes, we can properly prepare for time "offset" effects. However, to have that really working, we also need…
4) The ***Evaluation Engine*** should have some way of ***evaluating instances***/copies/temporary-background-calculations on ***separate copies of the data***.
    a. Viewport duplis, onion skinning in the viewport, baking animation/sims, rendering, background sets, sequencer scene strips, and shared-data with different animation applied to it (i.e. same mesh shape but different materials) all need to this to work properly. Currently, all of these end up

        clobbering state (i.e. this is why unkeyframed properties get lost, duplicates don't always behave).
    b. Proxies for linked data are just a sub-case of the general mechanism here (with some additional settings for what can/cannot be edited, and some way of keeping track of what has/hasn't been edited by users).

5) As optional extras (for later/eventually, but not necessary in the initial version):
    a. The ***ability for tools to actually query the Dependency Graph***, instead of searching for this information themselves. Results in less and faster code.
    b. The ***ability for*** tools/baking-***backends to use a copy of the Depsgraph containing only the operations necessary*** to calculate the data they need. Results in faster baking.

The first two are relatively easy, while the last two are difficult. If necessary, we may need to postpone #4 (for the Gooseberry Pilot), even though having it in place would be a huge boost. That said, the design discussed here outlines a way of achieving this.

Status

During last year's GSoC, I set to work on two attempts to translate these ideas into code. I realised midway through the first attempt that it wasn't going to work. The second attempt, did compile, but ultimately the codebase was not actually in a functional state yet, as there were still a few complications which arose (e.g. with regard to particles, metaballs, and a few other similar areas which had some really bizarre quirks).

The parts that were implemented though included the basic types (including the full set of node types and a sketch of what the evaluation contexts may be like), some core graph manipulation operations/API's, and some code for building these graphs (adapted from the old depsgraph code, but only partially completed due to some tricky issues). Critically though, I hadn't figured out how to exactly operations were going to get the data they needed to do what they needed to do.

Earlier this year, these efforts were later picked up by Lukas. He managed to patch up the code to get through more of building process, simplifying a few of the warts, getting the whole lot actually running (somewhat), and added some GraphViz-based debug drawing. Then, he started reimplementing the lot in C++, with efforts centered around a different way of implementing the graph builder.

Rethinking some assumptions…

Having had the benefit of some time away from this project focussing on other things, and also finally reading about both Pixar's approach (in Presto) in the kinds of things the Bullet people are doing for Bullet 3, I've come to the following conclusions:

1) It is impossible to try to build these kinds of graphs in "single pass" fashion. (Thanks Lukas for convincing me on this, and showing me another way!)
2) We shouldn't try to store the state of data being evaluated in the Dependency Graph nodes.

    a. This added a lot of complexity, resulting in the need to define heaps of node types, while also introducing the tricky problem of vagueness about who/what/where is state handled in the pipeline.
    b. Instead, an approach similar to the one described by Pixar (see the Multithreading and VFX course notes) of using a “Context API” to provide access to necessary copies of the data seems more promising. (Plus, it plays nicely with Sergey’s idea of “Copy on Write”, if we decide to do it that way). The rest of this doc describes the idea for the Context API…

3) The Components and nested-operation-nodes design was used as a way of addressing and identifying nodes, while keeping in mind performance considerations for when we need to look up affected nodes. A “lookup key/handle” approach may be simpler to implement (and better for performance – namely memory usage, etc.). Depending on how this goes, we may then add back these constructs later, if we find that they are necessary to get things working.
4) While I was initially trying to keep in mind issues **5a** and **5b** – causing complications with worrying about how we’d be able to look up and find the info we needed – taking a step back, the “Lookup Key/Handle” approach and some smarter logic (perhaps inspired by the Neo4J engine and its query language) might solve those issues when we get there later.
    a. At worst, we find that it’s too difficult, and we need to implement a parallel set of graph building techniques to handle this case.
5) Having larger memory “buffers”, each filled only with data of the same type, is apparently quite a good way of doing things in this industry, as it fits better with how computers actually work.

New Approach #3 – 3rd Time Lucky (Fingers Crossed)

Approach and Focus

This time around, I decided to try and attack the problem from another perspective: Instead of trying to figure out what the Dependency Graph itself should contain to represent the things we need it to, the approach this time is based on saying "what is required to tell the computer to let us schedule up dependency evaluation operations".

Secondly, we have a much smaller initial deliverable target this time: That is, initially, we'll be targeting "just" the rigging related parts of the depsgraph (i.e. object, bone, constraints, and drivers) running under a new system. By and large, these are also the parts which cause the greatest amount of grief, as well as being the areas which would benefit most from a modernised depsgraph. (Sure, the sequencer, proxies, duplis, and the sequencer system are also critical, but to keep things manageable, our planning should keep those off to the side for now).

Operation Nodes

As in our previous designs, the most "atomic" and important of node types in the Depsgraph is that of the ***Operation Node***. Taking a leaf out of Domain Specific Language (DSL) / programming language interpreter playbook: each Operation Nodes represents an instruction for the Evaluation Engine to execute.

As described by the guys at Pixar [Multithreading and VFX Course Notes], when building the Depsgraph, we're transforming the "scene description" concepts (and "visual programming language", aka "nodes") that artists and riggers work with into a representation optimised for the computer to execute.

The question then is: **What do Operation Nodes look like**?

Our previous answer was that these would be **callbacks** in the form of function-pointers to the evaluation methods that should get run, ***bundled with additional parameters*** attached for determining what they should do. The problem with this though was that there was always the issue of "where to get the data from?", and how we would be able to pass the different types of arguments down the pipeline. This is of course where it all started to devolve into talk of needing C++11 stuff like argument binding and all that stuff.

These days, I instead prefer a simpler, more "low tech" way. Once again, we draw inspiration directly from DSL techniques, and go back to making operation nodes simply **containers for** "**opcodes**". That is, each operation corresponds to some kind of numbered instruction that the operation handler figures out how to dispatch (Note: While OO/extensibility fanatics might not like this approach as much, from a debugging perspective, this is much easier for us to manage). More importantly though, the operation node would only contain a ***lookup key***, which the appropriate opcode handler would use to fetch the data it needs to operate on from the **context datastores**.

For examples of some opnodes and graph setups for some common scenarios (as well as many of our previously troublesome/unsolvable situations), check the examples section.

Operation Node Types

There are several different types of operation nodes (as can be seen in the examples).

Firstly, let's look at some of the generic node types:

- **Root Node** – This is entrypoint into the graph. There is only one of these in the graph. Each operation which can be executed independently of anything else should be hooked up to this. At this point, it doesn't actually do anything, though this may need to change in future.
- **Time Source Node** – This is used so that we can explicitly identify what operations change when the frame changes, and update all of those using standard techniques (instead of having to build a parallel hierarchy of flags and checks for those flags).
    - **Absolute Time Source Node** – This "sets the time" that evaluation takes place at. There should only be one of these in the graph though, and attached directly to the root node.
    - **Relative Time Source Node** – This is mainly used for Time Offset effects, to specify that some data needs to be evaluated on a different frame, relative to the current absolute time. It works in two ways:
        - 1) When evaluated, it reads the absolute time source (Note: it has a dependency relation to that), applies its offset to that, and writes this in an appropriate slot in the datastores,
        - 2) During the building process, time-dependent ancestors of a node (which directly depends on this relative timesrc) get duplicated and made to be dependent on this timesrc instead. Such duplicates get flags set on them to ensure that they are evaluated using temporary copies of the data (or maybe even a separate context). The specifics of how that work should be considered "implementation detail"?
- **Done Node** – The "Done" node is not strictly necessary, though it acts as a nice check mechanism to ensure that we do actually end only when there are no more pending nodes. It could be used for cleaning up temporary context if necessary too…

Besides these generic nodes, there are "type specific" nodes. That is, these nodes are specific to the data "**Component**" that they operate on. Components are blobs of closely-related temporary evaluation data which get stored in datastores in the Evaluation Contexts. In the previous design, they also corresponded to "outer nodes", which were used to group together operation nodes which operated on the same component (for easier lookup operations).

Some of the operation nodes here (grouped by component) are:

- **Data Component** – The "Data Component" corresponds to the properties and settings which you can edit via RNA.
    - **Animation Node** – The Animation opnode evaluates the NLA + Active Action stacks attached to an ID-block's AnimData. The results of evaluating this stack get saved to an "EvalChannelBuffer" (NOTE: this doesn't exist in standard Blender yet, but is basically necessary for resolving some bugs and limitations we have right now). These cached values in the buffer then get flushed to the data [TODO: WHEN EXACTLY THIS HAPPENS STILL NEEDS FIGURING OUT]

    - **Driver Node** – The Driver opnode evaluates a driver relationship, linking one or more “inputs” to a calculation for determining the new value of the driven property. This has implicit dependencies on the animation node (if present), and/or any of the other data nodes which get things into a usable base state
    - **Proxy Inherit Node** (?) – The idea here is that if we’re sharing some data and the user explicitly (proxifies the data) or implicitly (i.e. sharing data, and we animate over the top for some parameters), this is supposed to create a copy, which subsequent refs to the original/base data should use.
- **Transform Component** – This encapsulates object-level transforms data (i.e. the transform matrices + transform properties + constraints temporary data)
    - **Local Node** – This computes the standard loc/rot/scale matrix as the basic transform
    - **Delta Node** – Delta transforms are done separately and then applied on top of the local? [XXX: Review whether this should be rolled back into Local]
    - **Parent Node –** Parenting operation would basically be a separate operation too, like running a ChildOf constraint as part of the transform stack
    - **Constraint Node** – Each constraint in the stack becomes a separate node. [XXX: Review whether we need start/finish nodes to do the setup/cleanup of stack eval]
- **Pose Component** – This encapsulates pose solving (e.g. IK Chains) and bone evaluation data
    - **Pose Init Node** – This computes the IK chains and/or temp data
    - **Pose Done Node** – This frees the IK chains, and is used as a signal to the armature deformers that it’s safe to use the armature [XXX: something may need to change here ensure that the partially done armatures may also work, as long as all vgroups are satisfied]
    - **Bone Local Node** – This computes the standard loc/rot/scale matrix for a bone
    - **Bone Parent Node** – This computes the parenting magic for the bone, in all its complicated variations!
    - **Bone Constraint Node** – Just as for object constraints, except here we do it for bones. (NOTE: For IK and SplineIK constraints, these nodes are instead replaced by IK/SplineIK solver nodes instead. If constraints occur before/after such solvers, they will now be evaluated accordingly)
    - **Bone Done Node** – This computes the deform matrix for the bone, and is used to signal to others that the bone’s values can now be used
    - **IK/Spline IK Solver Node** – These nodes perform the chain solving operations. It is important to note that for each bone in the chain, their “done” nodes are dependent on these solver nodes in addition to just the standard constraints/transform stack. That way, it will all work out.
- **Geometry Component –** This encapsulates all things geometry and/or modifiers related
    - **Basis Node** – Creates the DerivedMesh/DispList/etc. for whatever geometry we’re dealing with
    - **Modifier Node** – Like with constraints, this represents a modifier, which takes a geometry object and produces a new one (or modifies the original). (NOTE: perhaps the Evaluation Context API will have some smarts here to determine how the dataflow here goes, based on the relationships going in/out)

    - **Path Node** – For all the tools that depend on having access to the "Path" data for a curve, this opnode explicitly captures the requirement that we generate that data. It can only be scheduled for after all modifiers have been done (assuming that works).

"All" Components (but mainly Transform/Geometry):

- At this point, I think I should draw special attention to some of the "**Ready**" nodes. The intention here is that, on the Evaluation Contexts, upstream requesters of evaluation results can provide handlers for slurping off the results as they become available. This is particularly relevant when dealing with large numbers of instances, as this means that render exporters or perhaps the viewport drawing code could grab the results as they become available, thus saving memory as we don't need to store all the instances in memory once they're calculated.

We could have other components as needed later. Examples could include sound, physics, sequencer stuff, or nodes?

There are also some experimental flow-control nodes we could add at a later stage, for fixing some of the more pathological rigging setups:

- **Conditional Node** – Only some sets of dependencies get evaluated depending on whether the condition associated with the node is **true** or **false**. Main use case is for those rigs where there are bidirectional switches or chains of controls, where the one in use should depend on which way is used
- **Cycle/Repeat Node** – It is more natural to define certain setups as declarative "constraints". This node could be inserted to instruct certain sets of operations to be repeated until some critical parameter stabilises (or a maximum loop count is met).

Evaluation Contexts and Datastores

One of the biggest problems with the old evaluation engine is that everything is evaluated on the main database. This is not too bad with simple scenes (and indeed, it's quite nice, as it keeps things quite simple to work with). However, as soon as you've got data being shared/reused in different ways, along with a need to allow some stuff to get calculated in the background while still permitting work to continue in the viewport, this model breaks down. Particular problem areas include Scene Strips in the Sequencer, animating properties in multi-user data from the owners, and calculating dupliframes/motionpath baking/onion skin drawing/physics caching.

Evaluation Contexts are the solution to this, and a basic implementation of this concept is partially implemented already. Whenever a dependency graph (including a "subgraph" – see later) gets evaluated, an evaluation context is created. Each evaluation context bundles together a version of the relevant parts of the database *in isolation* from other copies of the data that may exist in other evaluation contexts.

Each context contains a set of ***datastores***. Each datastore stores instances of a particular type of component. For example, there is one for Time Sources, another for Data Components, and others for Transform/Geometry/Pose/etc. Lookup keys are basically used to say which datastore should be searched, and what data to search for (or more accurately, index-into) within the store.

Within operation nodes, ALL data access to data outside the current struct must be mediated through the Evaluation Context API. This restriction allows us to do several important things:

1) Figuring out which instance exactly should be used, and to correct links accordingly to maintain this illusion
2) Implement Copy-On-Write or similar semantics for ensuring that the correct copies of data gets seen and edited

Sub-Graphs, Instancing, and Proxies

The Evaluation Context + Datastore abstraction mechanism provides us with a convenient mechanism for hiding the mechanics of being able to have multiple "outputs" based on the same DNA struct/datablock but with local changes included.

With the evaluation context and datastore systems in place, handling sub-graphs (e.g. sets, dupliframe evaluation, time offset, or other background evaluation processes) and instancing problems (e.g. multiple user data which needs to be animated/modified differently by different users) boil down to simply working on separate copies of the data requested.

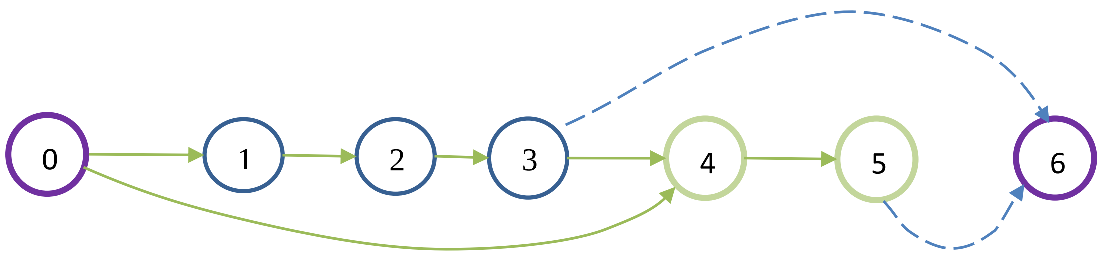
Examples of some Operation Nodes + Setups

Ex 1 – Cube with a Limit Location Constraint

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …}
0	GRAPH	ROOT	
1	TRANSFORM	LOCAL	{TRANSFORM, Cube} + TFM_ALL
2	TRANSFORM	CONSTRAINT	{TRANSFORM, Cube, LimitLoc Con.} + TFM_LOC
3	TRANSFORM	READY	{TRANSFORM, Cube} + TFM_ALL
4	GEOMETRY	MESH_BASIC	{GEOMETRY, Cube}
5	GEOMETRY	READY	{GEOMETRY, Cube}
6	GRAPH	DONE	

Node Graph:

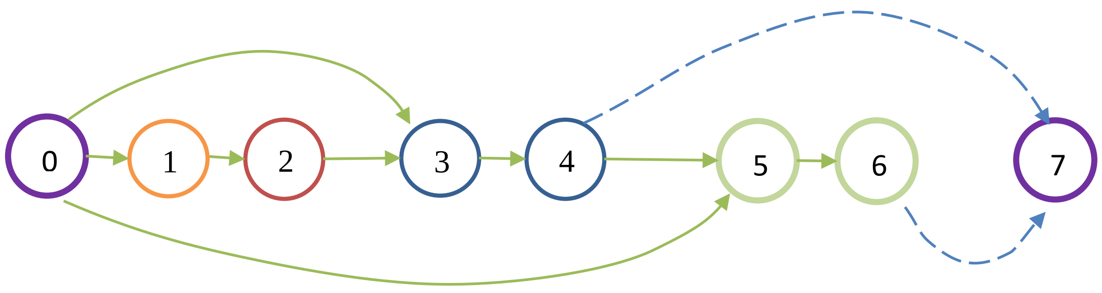


Ex 2 – Animated Cube

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …}
0	GRAPH	ROOT	
1	TIMESRC	TIMESRC_ABS	{TIMESRC, Scene, Primary}
2	DATA	ANIMATION	{DATA, Cube, AnimData}
3	TRANSFORM	LOCAL	{TRANSFORM, Cube} + TFM_ALL
4	TRANSFORM	READY	{TRANSFORM, Cube} + TFM_ALL
5	GEOMETRY	MESH_BASIC	{GEOMETRY, Cube}
6	GEOMETRY	READY	{GEOMETRY, Cube}
7	GRAPH	DONE	

Node Graph:

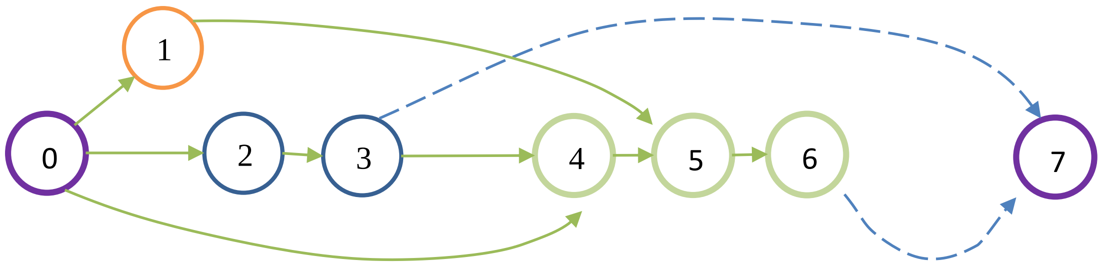


Ex 3 – Cube with Build Modifier

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …}
0	GRAPH	ROOT	
1	TIMESRC	TIMESRC_ABS	{TIMESRC, Scene, Primary}
2	TRANSFORM	LOCAL	{TRANSFORM, Cube} + TFM_ALL
3	TRANSFORM	READY	{TRANSFORM, Cube} + TFM_ALL
4	GEOMETRY	MESH_BASIC	{GEOMETRY, Cube}
5	GEOMETRY	MODIFIER	{MODIFIER, Cube, Build Modifier}
6	GEOMETRY	READY	{GEOMETRY, Cube}
7	GRAPH	DONE	

Node Graph:

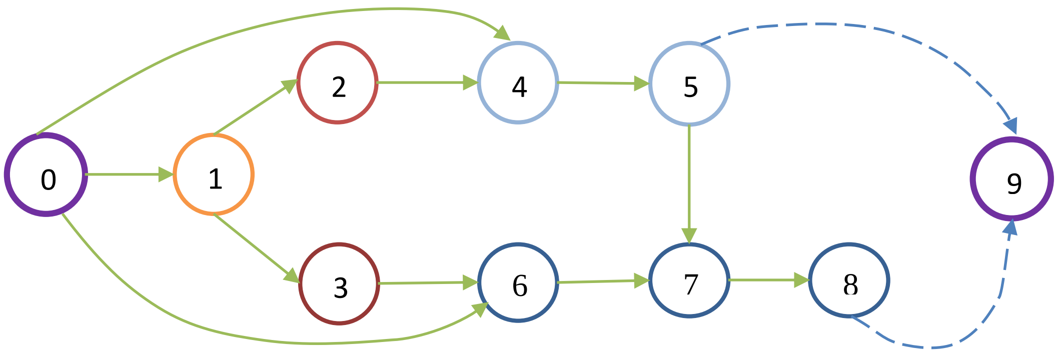


***Ex 4* – Object A parented to Object B, both animated**

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …}
0	GRAPH	ROOT	
1	TIMESRC	TIMESRC_ABS	{TIMESRC, Scene, Primary}
2	DATA	ANIMATION	{DATA, ObjectA, AnimData}
3	DATA	ANIMATION	{DATA, ObjectB, AnimData}
4	TRANSFORM	LOCAL	{TRANSFORM, ObjectB} + TFM_ALL
5	TRANSFORM	READY	{TRANSFORM, ObjectB} + TFM_ALL
6	TRANSFORM	LOCAL	{TRANSFORM, ObjectA} + TFM_ALL
7	TRANSFORM	PARENT	{TRANSFORM, ObjectA} + TFM_ALL
8	TRANSFORM	READY	{TRANSFORM, ObjectA} + TFM_ALL
9	GRAPH	DONE	

Node Graph:



***Ex 5* – Bone-drivers affecting constraint influence in same armature**

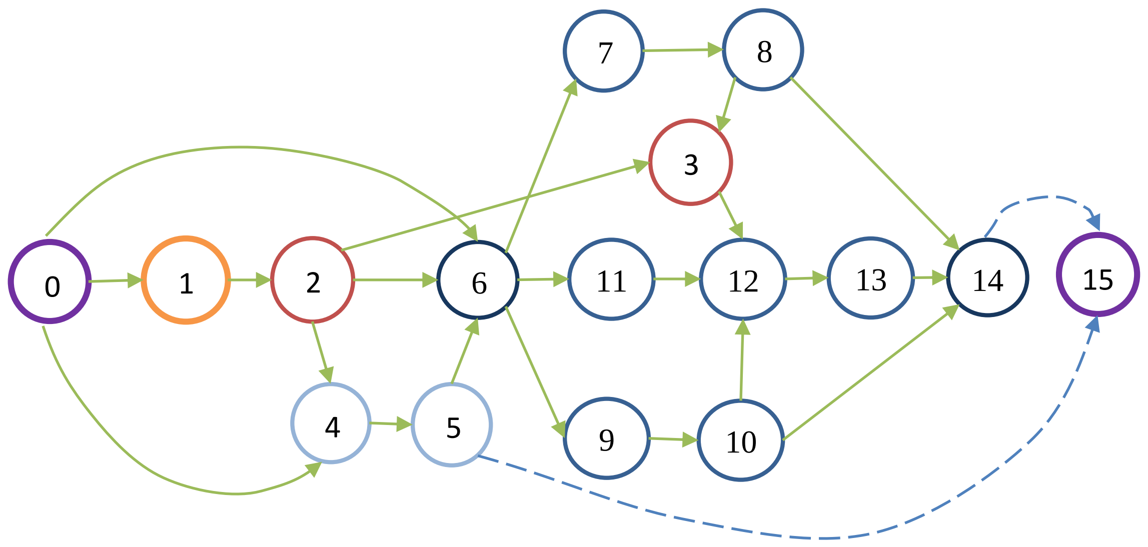
We have 3 bones:

- Bone "A" has a "Copy Rotation" constraint targeting bone "B".
- Bone "C" is the 'control' bone, which uses a driver to adjust the constraint influence on Bone "A".
- Bones "A" and "B" are animated

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …}
0	GRAPH	ROOT	
1	TIMESRC	TIMESRC_ABS	{TIMESRC, Scene, Primary}
2	DATA	ANIMATION	{DATA, ArmatureOb, AnimData}
3	DATA	DRIVER	{DATA, ArmatureOb, Bone A.Constraint}
4	TRANSFORM	LOCAL	{TRANSFORM, ArmatureOb} + TFM_ALL
5	TRANSFORM	READY	{TRANSFORM, ArmatureOb } + TFM_ALL
6	POSE	POSE_INIT	{POSE, ArmatureOb}
7	POSE/BONE	BONE_LOCAL	{BONE, ArmatureOb, Bone C}
8	POSE/BONE	BONE_DONE	{BONE, ArmatureOb, Bone C}
9	POSE/BONE	BONE_LOCAL	{BONE, ArmatureOb, Bone B}
10	POSE/BONE	BONE_DONE	{BONE, ArmatureOb, Bone B}
11	POSE/BONE	BONE_LOCAL	{BONE, ArmatureOb, Bone A}
12	POSE/BONE	CONSTRAINT	{BONE, ArmatureOb, Bone A}
13	POSE/BONE	BONE_DONE	{BONE, ArmatureOb, Bone A}
14	POSE	POSE_DONE	{POSE, ArmatureOb}
15	GRAPH	DONE	

Node Graph:



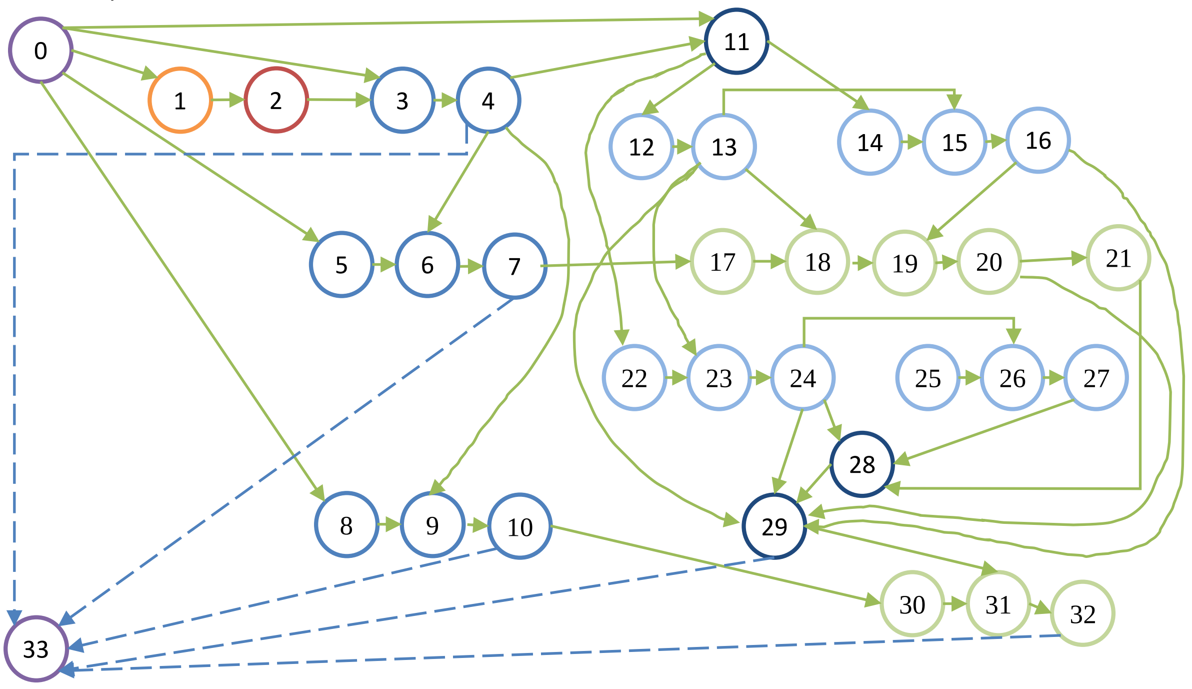
***Ex 6* – Spline IK Rig in a Single Armature**

- We have an armature (**A1**) with 4 bones: **C1** and **C2** are control bones, and **B1** and **B2** are deformed by Spline IK. **C1** is the parent of **C2** and **B1**, **B1** is the parent of **B2**.
- Curve **S1** is used to control **B1** and **B2** (via the Spline IK solver). It is deformed by **C1** and **C2**, which affect its verts.
- Mesh **M1** is deformed by **A1**. Specifically, it has vertex groups for **B1** and **B2**.
- **S1** and **M1** are both parented to **A1**
- **A1** is animated (with animation curves for **C1** and **C2**)

Operation Nodes (each row represents one node):

id	Component Type	OpCode	Lookup Key (Component, ID-Block, …)
0	GRAPH	ROOT	
1	TIMESRC	TIMESRC_ABS	{TIMESRC, Scene, Primary}
2	DATA	ANIMATION	{DATA, A1, AnimData}
3	TRANSFORM	LOCAL	{TRANSFORM, A1} + TFM_ALL
4	TRANSFORM	READY	{TRANSFORM, A1 } + TFM_ALL
5	TRANSFORM	LOCAL	{TRANSFORM, S1} + TFM_ALL
6	TRANSFORM	PARENT	{TRANSFORM, S1} + TFM_ALL
7	TRANSFORM	READY	{TRANSFORM, S1} + TFM_ALL
8	TRANSFORM	LOCAL	{TRANSFORM, M1} + TFM_ALL
9	TRANSFORM	PARENT	{TRANSFORM, M1} + TFM_ALL
10	TRANSFORM	READY	{TRANSFORM, M1} + TFM_ALL
11	POSE	POSE_INIT	{POSE, A1}
12	POSE/BONE	BONE_LOCAL	{BONE, A1, Bone C1}
13	POSE/BONE	BONE_DONE	{BONE, A1, Bone C1}
14	POSE/BONE	BONE_LOCAL	{BONE, A1, Bone C2}
15	POSE/BONE	BONE_PARENT	{BONE, A1, Bone C2}
16	POSE/BONE	BONE_DONE	{BONE, A1, Bone C2}
17	GEOMETRY	CURVE_BASIS	{GEOMETRY, S1}
18	GEOMETRY	MODIFIER	{GEOMETRY, S1, Mod["BoneHook"]}
19	GEOMETRY	MODIFIER	{GEOMETRY, S1, Mod["BoneHook2"]}
20	GEOMETRY	READY	{GEOMETRY, S1}
21	GEOMETRY	PATH	{GEOMETRY, S1}
22	POSE/BONE	BONE_LOCAL	{BONE, A1, Bone B1}
23	POSE/BONE	BONE_PARENT	{BONE, A1, Bone B1}
24	POSE/BONE	BONE_DONE	{BONE, A1, Bone B1}
25	POSE/BONE	BONE_LOCAL	{BONE, A1, Bone B2}
26	POSE/BONE	BONE_PARENT	{BONE, A1, Bone B2}
27	POSE/BONE	BONE_DONE	{BONE, A1, Bone B2}
28	POSE	SPLINE_IK	{POSE, A1, SplineIk1(Bone["B2"]) }
29	POSE	POSE_DONE	{POSE, A1}
30	GEOMETRY	MESH_BASIS	{GEOMETRY, M1}
31	GEOMETRY	MODIFIER	{GEOMETRY, M1, Mod["ArmatureDeform"] }
32	GEOMETRY	READY	{GEOMETRY, M1}
33	GRAPH	DONE	

Node Graph:



…