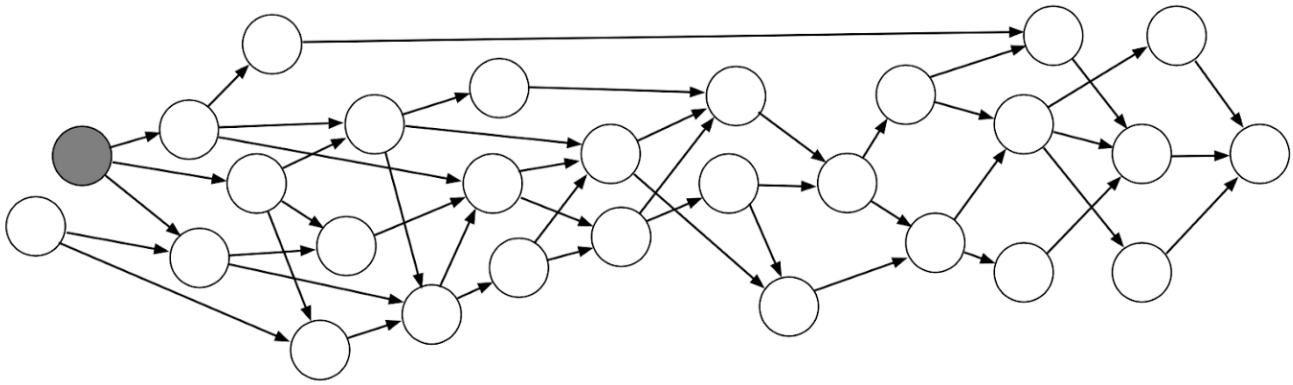
Organizing GPU Work with Directed Acyclic Graphs

Pavlo Muratov Follow

Jul 10, 2020 · 19 min read



What is there to organize?

> *DirectX 11 and OpenGL style APIs require the driver to invoke complex heuristics to determine when and how to perform critical scheduling operations on the GPU. The immediate-mode nature of the interface requires complex bookkeeping and state-tracking to handle every edge case — ultimately, negatively impacting performance and hampering parallelism.*
>
> *Modern graphics APIs, like DirectX 12, Vulkan, and Metal 2, have shifted away from this model, instead opting to shift the burden of low-level GPU management to the application itself.*

That gives us an opportunity to design a system that possesses high-level knowledge of the rendering work in a frame and determines optimal way to execute it beforehand. That includes optimizing GPU and memory states, memory reuse, queue synchronization. Such system can also be used to validate correctness of frame

workflow and cull redundant jobs, re-optimize itself in response to user actions. And **graph** is a natural candidate for underlying data structure.

In this article we'll touch several subjects: building the graph, using it to batch work and insert proper synchronization points, arranging resource barriers.

Design of the algorithms below was inspired by DirectX 12 API. Code examples with implementation details will be provided at the very end.

Building the Graph

Let's start from the basics. We want to organize work, so let's define a unit of work and call it **Render Pass.** Work is no good if we can't use its results, so we need to define its **outputs** and **inputs**, which are represented by read and written GPU resources. And this is all the information required to build a graph, nodes of which represent render passes.

```
class Node
{
  void AddReadDependency(Foundation::Name resourceName, const SubresourceList& subresources);
  void AddWriteDependency(Foundation::Name resourceName, const SubresourceList& subresources);

  std::unordered_set<SubresourceName> ReadSubresources;
  std::unordered_set<SubresourceName> WrittenSubresources;
}
```

Simplest node example

```
class RenderPassGraph
{
  void AddPass(const RenderPassMetadata& passMetadata);
  void RemovePass(NodeListIterator it);

  NodeList Nodes;
}
```

simplest_graph.hpp hosted with ♡ by GitHub view raw

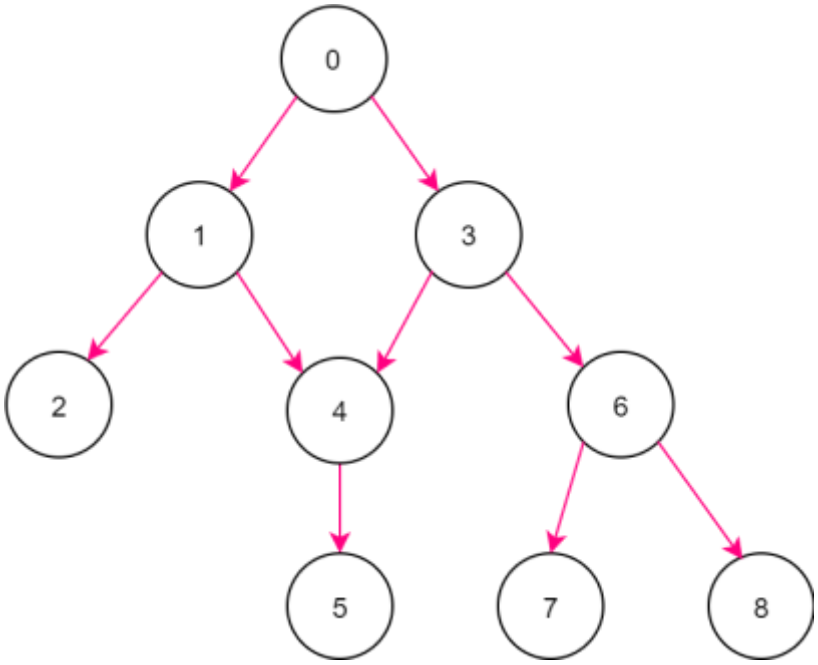
Graph can be a simple flat list of nodes

```
class ResourceScheduler
{
  void NewRenderTarget(Foundation::Name resourceName);
```

```
    void NewDepthStencil(Foundation::Name resourceName);
    void NewTexture(Foundation::Name resourceName);
    void UseRenderTarget(Foundation::Name resourceName, const MipList& mips = { 0 });
    void UseDepthStencil(Foundation::Name resourceName);
    void ReadTexture(Foundation::Name resourceName, const MipList& mips = { 0 });
    void WriteTexture(Foundation::Name resourceName, const MipList& mips = { 0 });
    template <class T>
    void NewBuffer(Foundation::Name resourceName, const NewBufferProperties<T>& bufferProperties
    void ReadBuffer(Foundation::Name resourceName, BufferReadContext readContext);
    void WriteBuffer(Foundation::Name resourceName);
    void ExecuteOnQueue(RenderPassExecutionQueue queue);
    void UseRayTracing();
}
```

scheduler_example.hpp hosted with ♡ by GitHub view raw

An example of an interface of a class used to register dependencies in graph's nodes



A bunch of render passes with various dependencies and no loops

Now that we established basic definitions, let's take an unordered, but dependent set of render passes and order them in a deterministic, non-contradictory manner.

Those are important properties required to make graph build fully automatic.

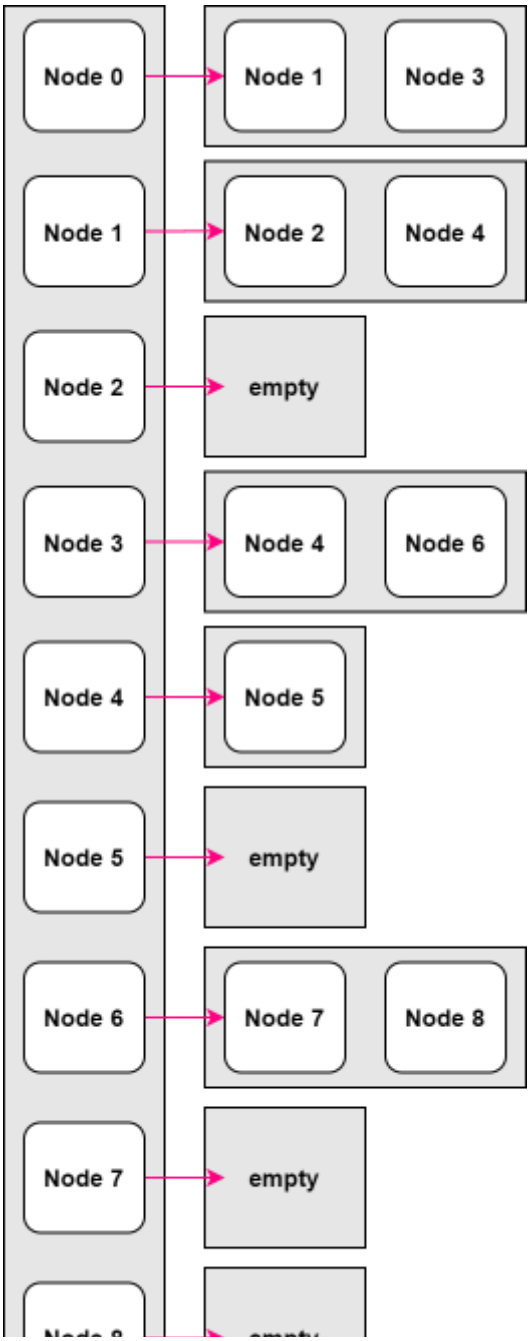
That's where acyclicity comes in. It imposes a couple of restriction we must consider before building the graph.

Any cyclic dependency will bring ambiguity when the algorithm won't be able to tell which of the cycled nodes has to come first and which has to come last. So, firstly, we

have to forbid cyclic dependencies.

Another caveat is multiple write dependencies for the same resource. If two or more render passes register same resource as their output (write) dependency the algorithm will also fail to establish a single node order because there will be more than one possible solution. Hence, secondly, we forbid multiple write dependencies for the same resource and so each render pass must register its own new output. *Requiring single write dependency is debatable though and will be discussed additionally at the end, but for now we'll accept it.*

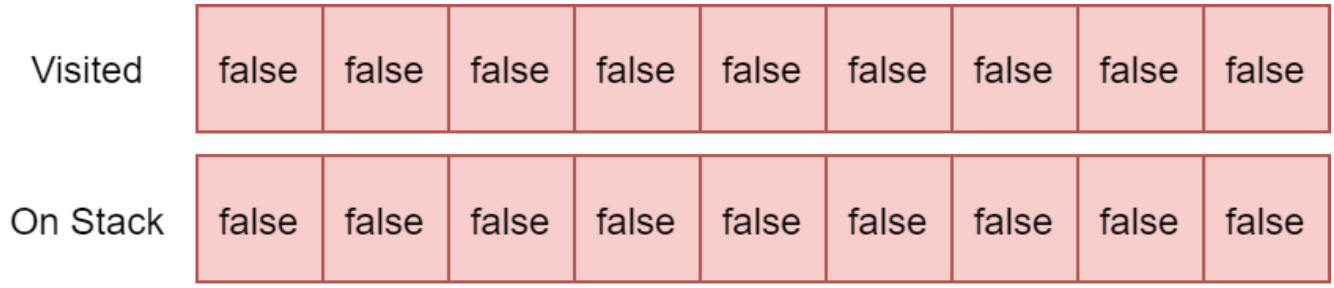
Now with these restrictions in mind we'll order the example graph by performing a **topological sort** based on a modified version of **depth-first search** algorithm. It can order a directed acyclic graph in linear time.



Adjacency lists for the graph

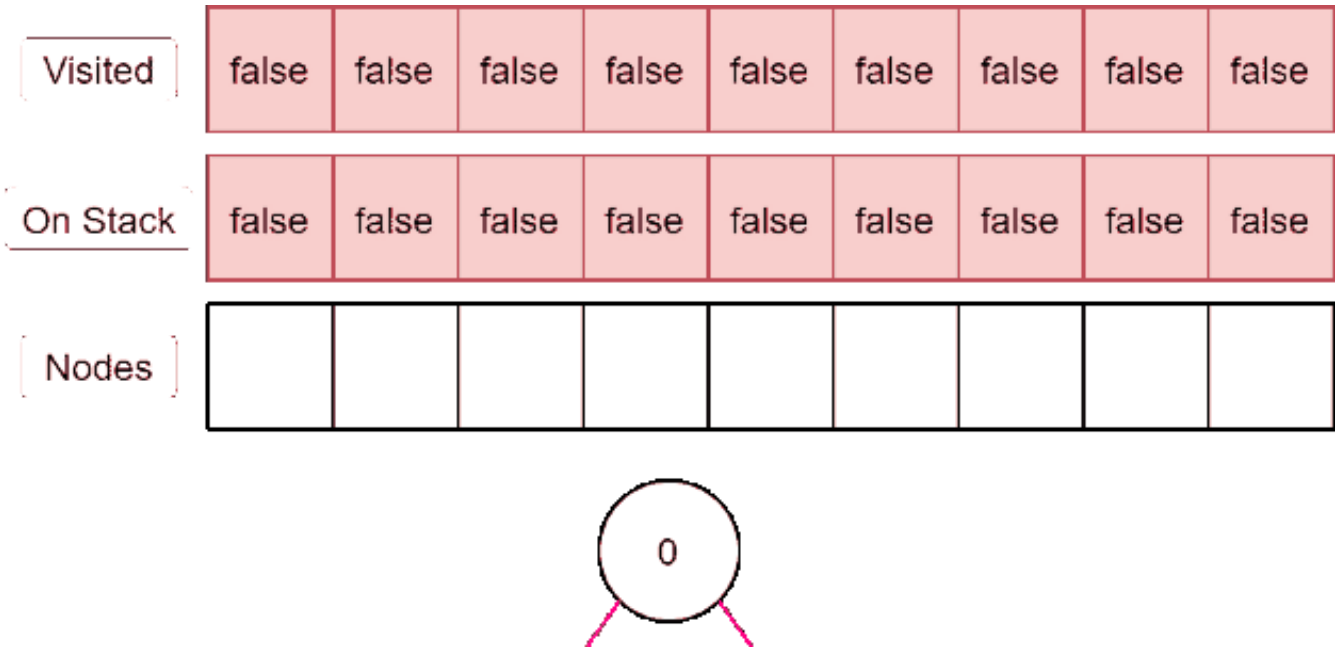
The algorithm requires nodes and edges as its inputs. We have our render pass nodes, but we don't actually have edges yet, we only know which resources are read and written by each render pass. We can use that information to build a data structure called **adjacency list**, which is just an array of arrays, where N arrays is created for N nodes in a graph and each array contains references to connected nodes. We do this by checking for write-to-read dependencies between every possible pair of nodes.

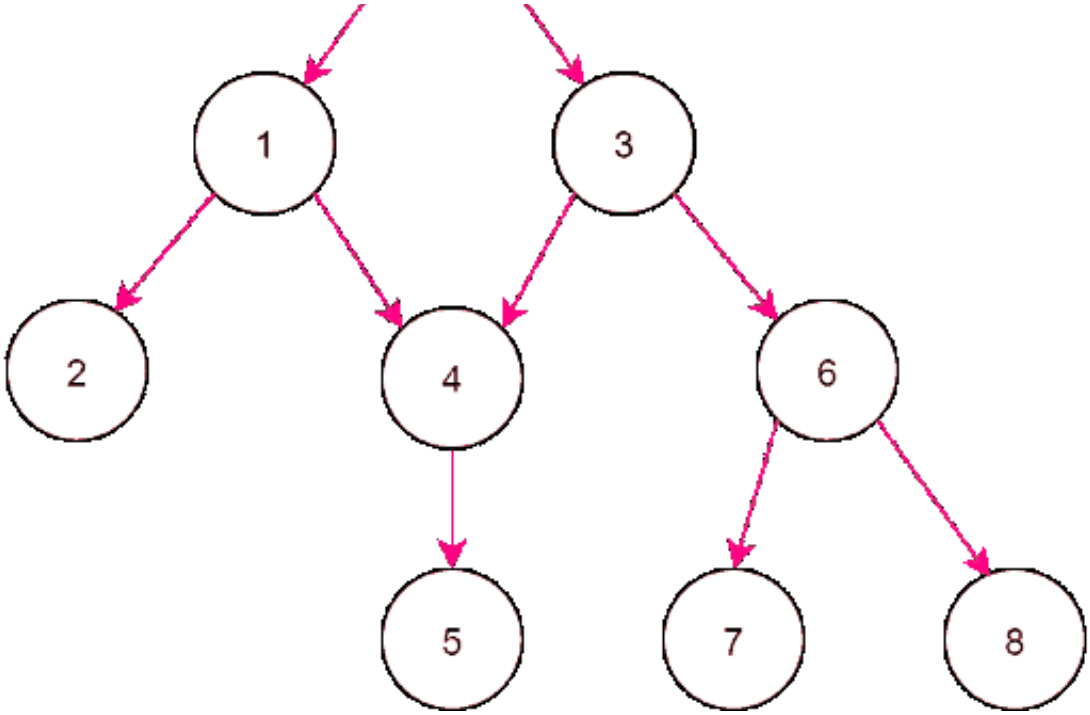
Now with adjacency list constructed we can order the graph. To do that we'll need two helper arrays of booleans, each of size N, where N is the total number of nodes in the graph. One is used to mark "visited" nodes and the other to mark nodes "on stack" to detect circular dependencies.



Helper arrays for the graph

We iterate through every node in the graph, and for each node we start a recursive search through descendants, marking encountered nodes as "on stack" and "visited".





Topological sort

We add node to the final array when no more descendants could be found. When stack is unrolled we remove the “on stack” flag, but leave the “visited” flag, which will help us avoid processing a node more than once. If during the search we encountered a node that is already visited and also is on the stack, then we found a circular dependency and must abort.

In example above we started from node 0, so we were able to order the whole graph with just one recursive search, but in a general case we might need more, that’s why descendant search should be issued for every node.

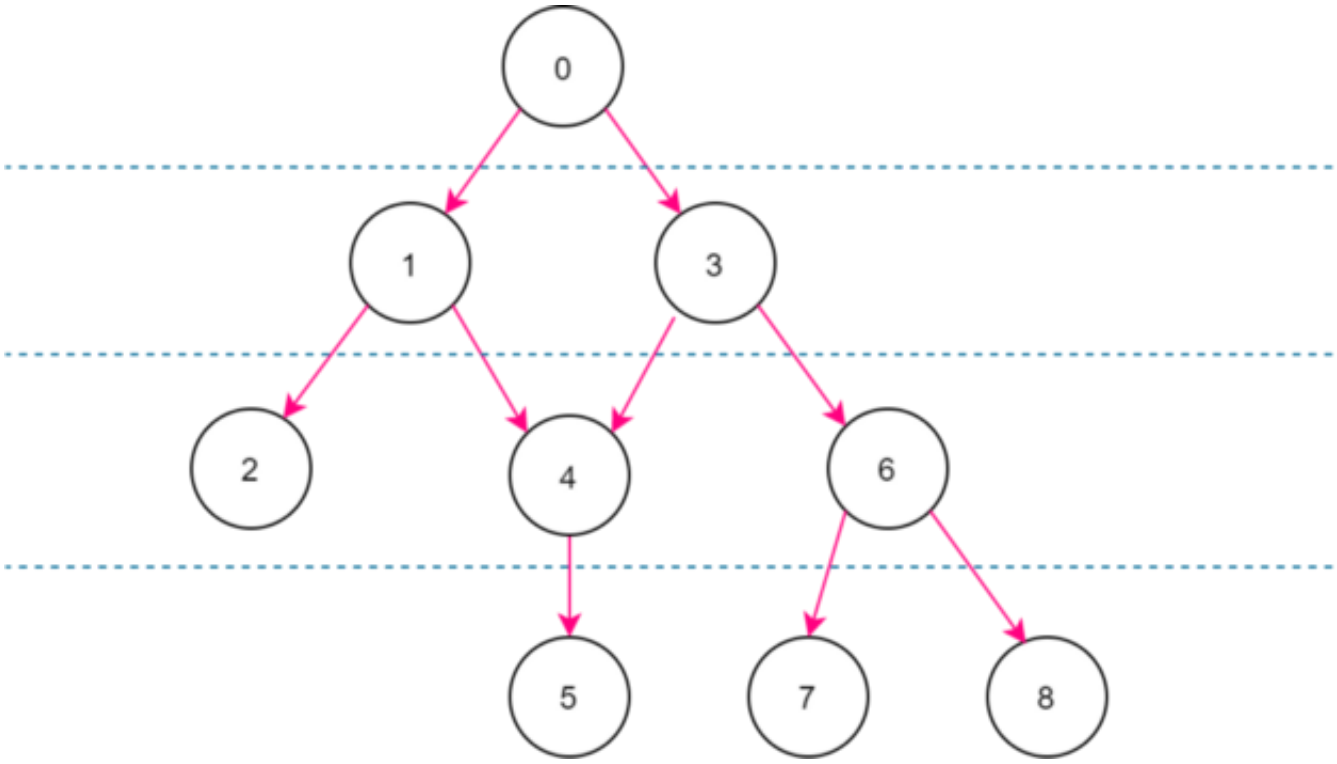
As a last step we reverse the resulting array.

0	3	6	8	7	1	4	5	2

Topologically sorted nodes

We can now execute render passes in a determined order and dependencies will be respected.

But there is one more thing to consider before we move on. Let’s look at the graph again.



We can see that there exist groups of nodes that do not depend on each other. Let's call those groups **dependency levels**. Render passes inside of a single dependency level can be executed in an arbitrary order which is useful for parallel work execution and certain optimizations we'll discuss later on.

An important relation here is that passes in a dependency level share the same recursion depth, or rather *maximum* recursion depth also known as **longest path in a DAG** from a root node. We can use **longest path search** algorithm to find dependency level of each node.

To do that we define an array of distances to each node of size N, where N is the number of nodes in the graph and initialize it with zeroes. We also need a topological sorting of the graph, but that we already have. We then iterate through nodes in topological order and for every node processed, we update distances of its adjacent nodes using distance of current node. Distances determined for each node will correspond to their dependency level indices. Nodes are then dispatched to their dependency levels.

Algorithm is simple enough to be explained with just four lines of pseudo code:

```
1  Do following for every vertex u in topological order.
2    Do following for every adjacent vertex v of u
3      if (dist[v] < dist[u] + 1)
4        dist[v] = dist[u] + 1
```

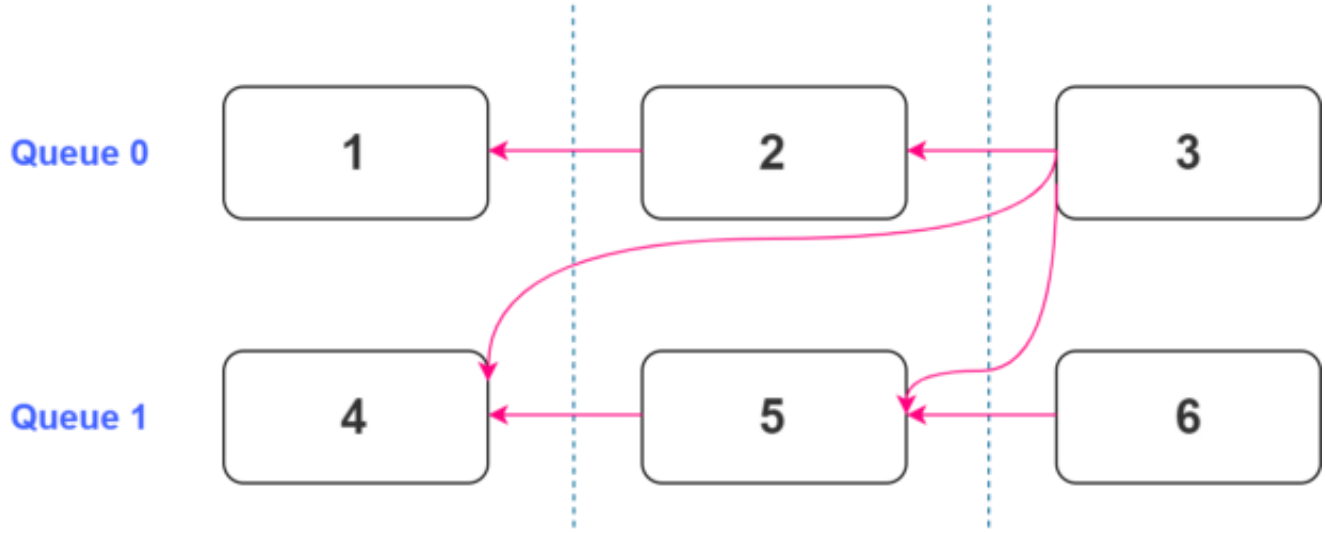
longest_path_search_preudo.hpp hosted with ♡ by GitHub view raw

Longest path search

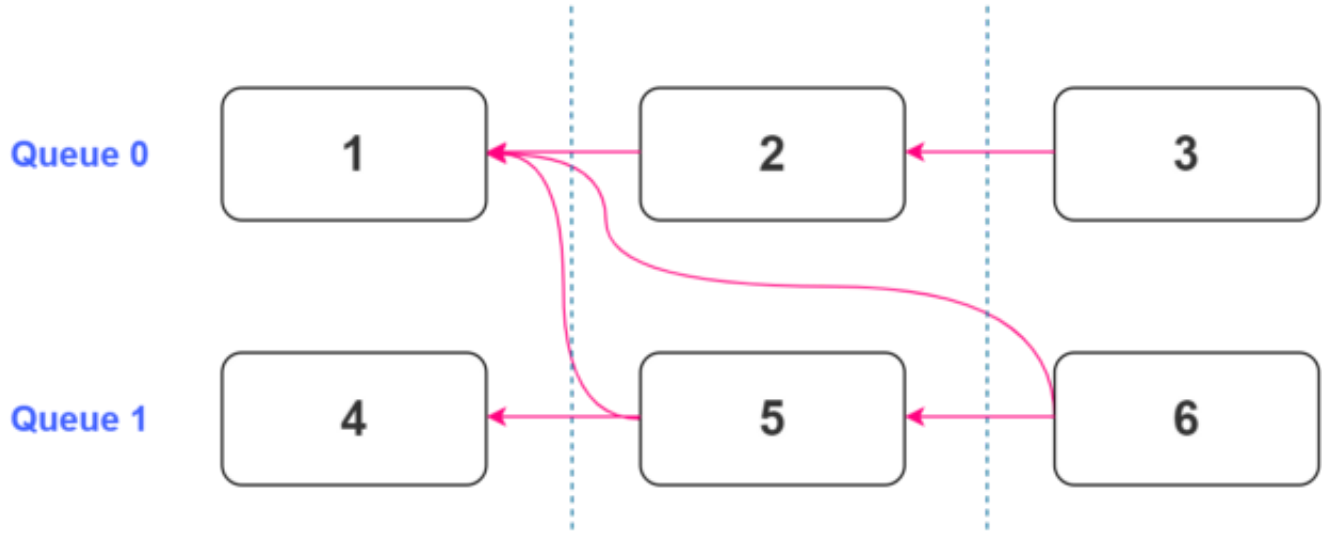
Using the Graph: Synchronization

We just built a graph of work, but that work can be executed in parallel using command queues exposed by modern graphic APIs. That leaves us with a challenge of finding proper synchronization points to respect dependencies between render passes executed on separate queues and also minimizing the amount of synchronizations to batch more work together and minimize the number of Execute…() calls to improve performance.

In this section we'll develop an algorithm to achieve optimal synchronization between queues by examining several scenarios and by trying to find a universal solution for all of them.



Synchronization scenario 1



Let's start by looking at two cases. We have two queues and three passes on each queue (we'll focus on cross-queue work dependencies, because otherwise things are trivial). A monotonically increasing index is assigned to each pass (index could've started from 0 on each queue, but that doesn't matter, only monotonic increase matters).

In first case we have a pass **(3)** that needs to sync with two other passes from another queue **(4,5)**. In the second case we have two passes **(5,6)** that need to sync with one common pass **(1).**

Just by looking at the pictures we can tell that in first case synchronizing **(3)** with **(4)** is redundant because that sync is automatically covered by synchronization between **(3)** and **(5)**. So what's the algorithm here? Take **(3)**, iterate through dependencies on other queue **(4,5)** and take only the closest one, with highest index — **(5)**, discard the rest.

So, is it good enough for both cases? Actually no, second picture tells us it's not.

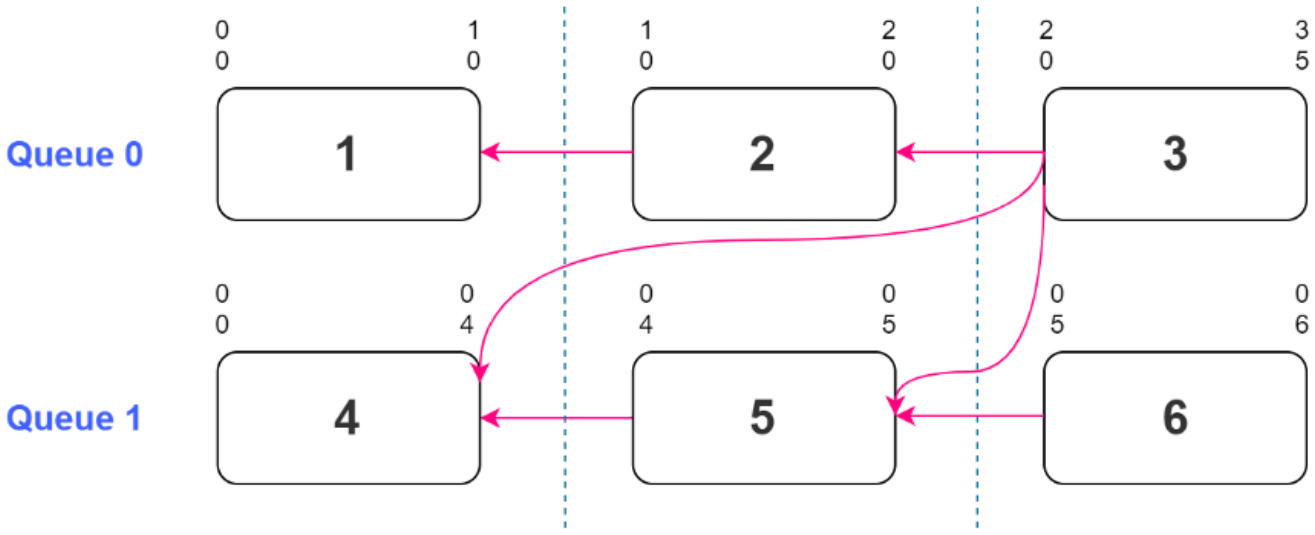
If we'd only look for closest dependencies in second case, we'd end up with the same picture: **(5)** would be synced with its closest dependency **(1)** and so would **(6),** since its closest dependency is also **(1)**. But we can clearly see that synchronizing **(6)** with **(1)** is redundant because of sync between **(5)** and **(1)**. A solution for second case is not immediately obvious.

So what do we do now?

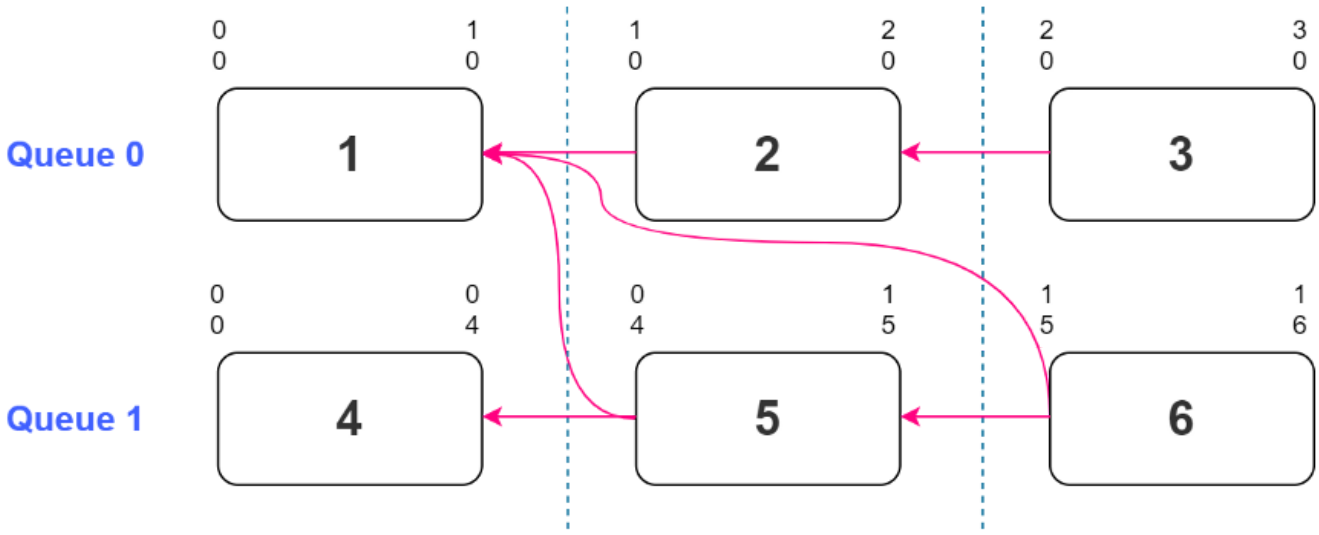
We proceed by introducing a notion of **sufficient synchronization index set** (SSIS) first, which is a set of numbers assigned to each node in the graph containing indices of closest nodes on each queue a node needs to synchronize with.

i0 - Index of closest node to sync with on queue 0
i1 - Index of closest node to sync with on queue 1
i2 - Index of closest node to sync with on queue 2
i3 - Index of closest node to sync with on queue 3
...
in - Index of closest node to sync with on queue N

Now we take two previous examples and assign SSIS to each node.



Synchronization scenario 1 with assigned SSIS values



Synchronization scenario 2 with assigned SSIS values

We start from all zeros in SSIS and then go through each node and for each node iterate through its dependencies on another queues and store indices of closest dependencies into its SSIS. SSIS values for the queue a node corresponds to are set a bit differently (for nodes on queue 0 that would be SSIS[0] and for nodes on queue 1 that'd be SSIS[1] and so forth). In those cases we assign not the index of a dependency (which is a previous node on that queue), but the index of the node itself. Current and previous nodes on the same queue might not be dependent through resource reads/writes, but for the algorithm to work we need to add previous node as a dependency of current one regardless. That's of course only if previous node exists.

Following those rules, in first example SSIS for **(3)** becomes (3,5), and in second example SSIS for **(5)** becomes (1,5) and SSIS for **(6)** becomes (1,6).

Now that we've built SSIS for each node we can look for indirect dependencies to cull redundant syncs and solve the problem we have in the second example: node **(6)** is synced with **(1)** indirectly through **(5)** and sync **(6) -> (1)** is redundant.

To find indirect dependencies we introduce second pass of the algorithm.

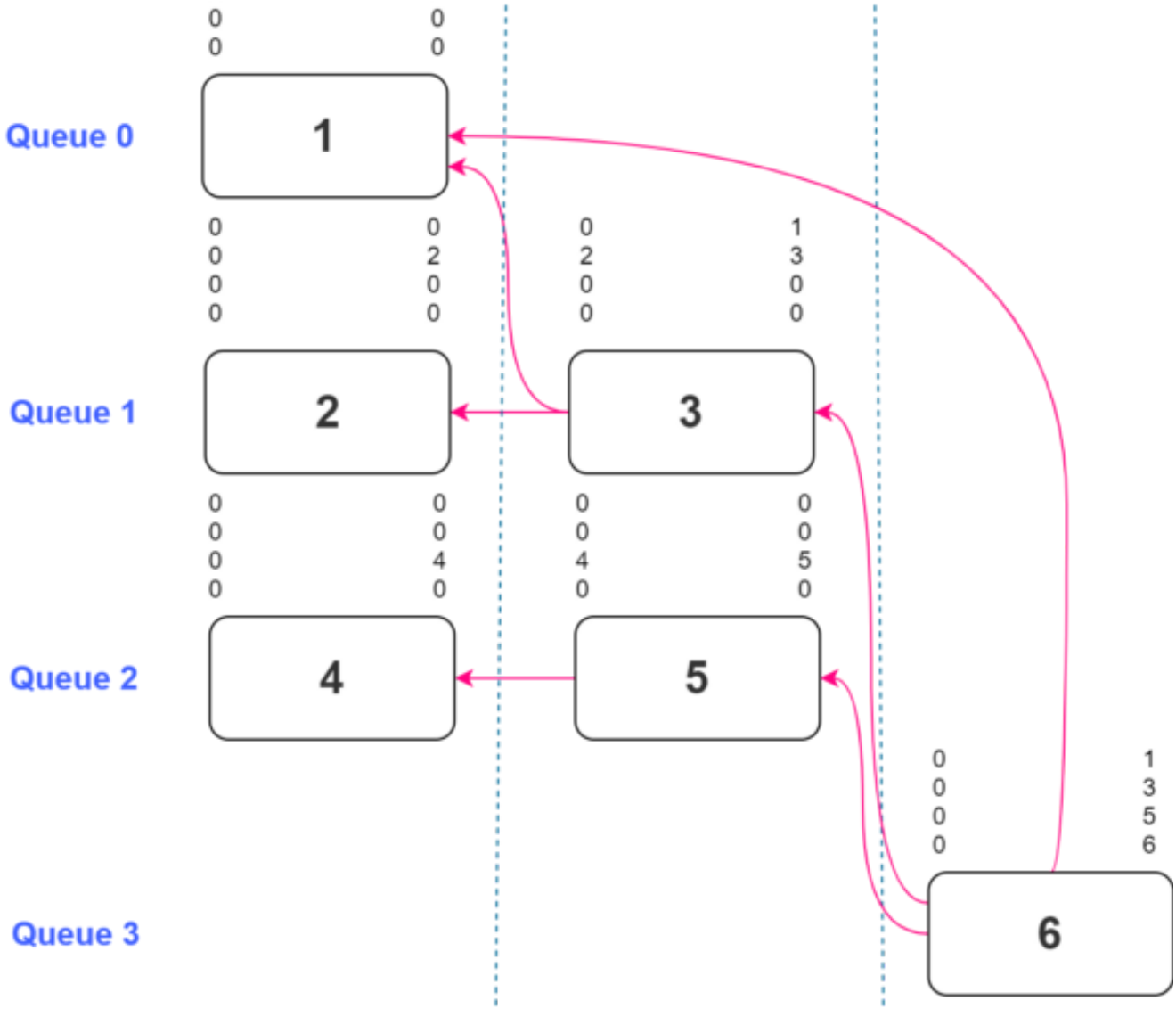
Iterate through each node again and for each node look at its dependencies, again. But this time compare values in node's SSIS with values in SSIS of the dependency.

By comparing current SSIS with SSIS of dependency we can *determine a number of queues we would synchronize with if we'd only synchronize with this dependency only*. We compare values for "other" queues directly and for "current" queue by first subtracting 1 from SSIS of current node, to account for different SSIS assignment rule for dependencies on the same queue that we discussed above.

So let's take node **(6)** and its first dependency node **(1)**, and compare SSIS values for queue 0. SSIS(Node 6)[0] <= SSIS(Node 1)[0] (1 <= 1) is **true** so we know sync with queue 0 is covered by synchronization with node **(1)**. Now compare values for second queue. SSIS(Node 6)[1] <= SSIS(Node 1)[1] ((6–1) <= 0) is **false** and that means we would **not** synchronize with queue 1 by synchronizing with node **(1)** exclusively and so we must consider other dependencies to achieve full set of required synchronizations. Now take next dependency of node **(6)** which is node **(5)**. By comparing SSIS values we see that synchronizations with both queues are covered by synchronizing with only **(5)**, because both 1 <= 1 and (6–1) <= 5 are **true**. Sync **(6) -> (1)** is now redundant and can be culled.

In a more general case with N queues we build a list of queues we need to sync with and a list of dependency nodes with number of synchronizations they cover from this queue list. Then we pick a node with maximum number of covered syncs and discard the rest.

This generalization is sufficient to minimize synchronizations for the cases we examined, but still not full enough to cover all scenarios. Let's examine another case where this algorithm doesn't hold up and develop an improvement for it.



A more sophisticated synchronization scenario when one pass of search for indirect dependencies is not enough

Here we can see that node **(6)** needs to be synced with queue 0 after node **(1)**, but sync with **(1)** is redundant, because it can be achieved through synchronization with node **(3)**. Let's try to cull redundant synchronizations using the algorithm we had up to this point.

Build list of queues to synchronize with: 0, 1, 2. Find dependency nodes that will synchronize us with maximum number of required queues by comparing SSIS values.

Possible synchronizations:

6 -> 1: 1 queue covered, because 1<=1 **true**, 3<=0 **false**, 5<=0 **false**

6 -> 3: 2 queues covered, because 1<=1 **true**, 3<=3 **true**, 5<=0 **false**

6 -> 5: 1 queue covered, because 1<=0 **false**, 3<=0 **false**, 5<=5 **true**

According to our algorithm we take sync with maximum queues covered (**6 -> 3)** and discard the rest, but now we have a problem because only syncs with queues 0 and 1 are performed, queue 2 is left out.

We can fix this by extending the algorithm.

After first pass of searching for dependencies covering maximum amount of queue syncs, remove queues that were covered from the list of queues we need to synchronize with and remove such dependency nodes from the list of dependencies to consider on next iteration (if there is one). Also update SSIS values of current node with SSIS values of accepted dependencies for queues that were covered, because SSIS value of dependency can actually be larger than what is expected by current node (which is fine). Now check if we have any queues left we want to sync with. If so, repeat the search of best fit nodes in the list of nodes left from previous iteration. Repeat the cycle until no queues are left in the list. Now nodes that are left as dependencies are the nodes synchronizations with which can be safely culled.

And that's it, final piece of an algorithm to find optimal synchronization points given an arbitrary queue/node combination.

Note: having nodes from the same queue as dependencies is only useful as a tool to find indirect synchronizations, they do not actually translate to API synchronizations via fences, because work on the same queue is synced implicitly.

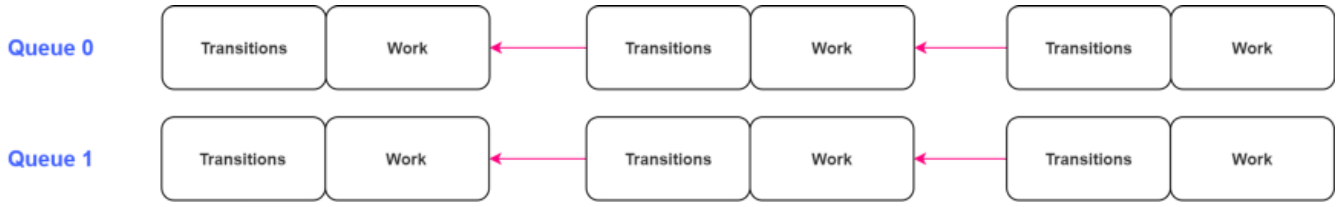
Let's recap.

> *Perform first pass by going through each node and determine their SSIS by looking at closest dependencies on each queue.*
>
> *Perform second pass to cull redundant synchronizations by determining a list of relevant queues to sync with and iteratively finding smallest set of nodes synchronizations with which will satisfy all required queue synchronizations, using SSIS comparisons.*

Using the Graph: Resource State Transitions

We're done with the easier part and move on to resource state transitions which add a thick layer of complications on top of what we already have. Let's begin by looking at simple cases. The simplest one would be a single queue of render passes or multiple independent queues.

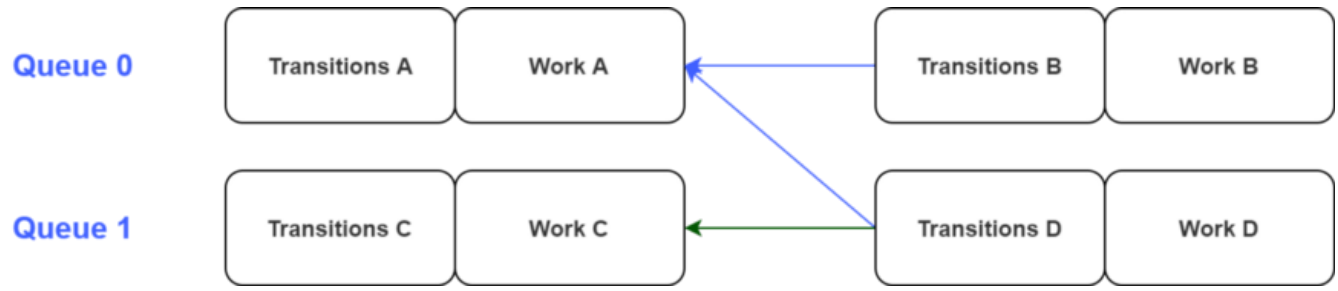
Single queue



Couple of independent queues

Transitions are straightforward then: render, transition, repeat. Boring.

Things start to get interesting when we have a render pass on one queue that uses a resource produced on another.



Cross-queue resource read

There are two restrictions we have to deal with.

First, resource transition may contain states unsupported by receiving queue. Example could be a **RenderTarget** produced on graphics queue and consumed on compute queue as a **NonPixelShaderAccess**.

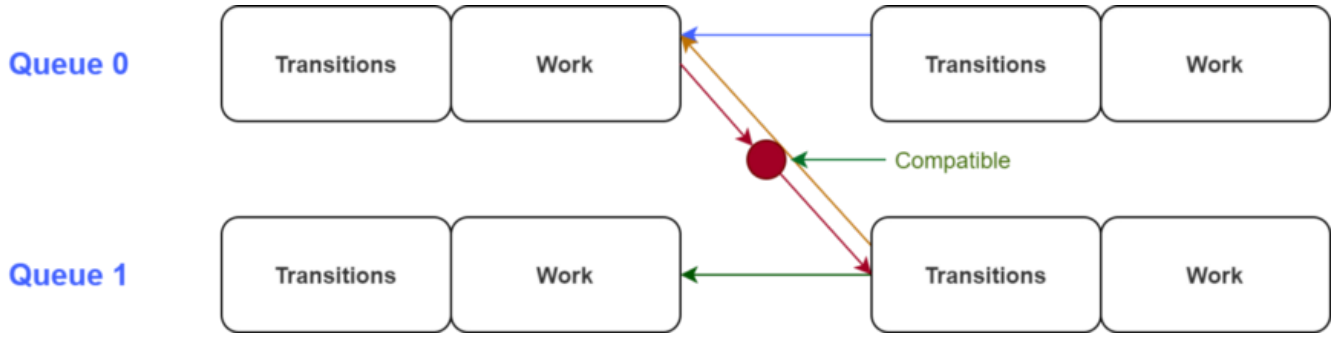
Second, a common resource can be read by multiple queues, so transitions must be put somewhere outside of work executed on those queues and synchronizations must be adjusted accordingly.

We can also have a combination of two cases, when resource is read by multiple queues, but transition to a common read state is unsupported on some or even all of the receiving queues.

An important note here is that even if queue does not support a state transition it still may be able to use resource in that state, it's just that transition must happen somewhere else. For example, if an **UnorderedAccess** resource produced on graphics

queue is read later on both graphics and compute queues as **PixelShaderAccess** on graphics and **NonPixelShaderAccess** on compute, we must transition **UnorderedAccess** into a combined read state suitable for both queues (**AnyShaderAccess** = **PixelShaderAccess** | **NonPixelShaderAccess**). Such transition can only be made on graphics queue, because compute doesn't know anything about pixel shaders. But, it will happily use **AnyShaderAccess** state regardless, after graphics performed the transition, because it only cares for **NonPixelShaderAccess** being present inside a combined read state, other states, known or unknown to it, are ignored.

Now let's look at several examples of challenges we just discussed and try to come up with a strategy (or strategies) to overcome them.



Compatible cross-queue read

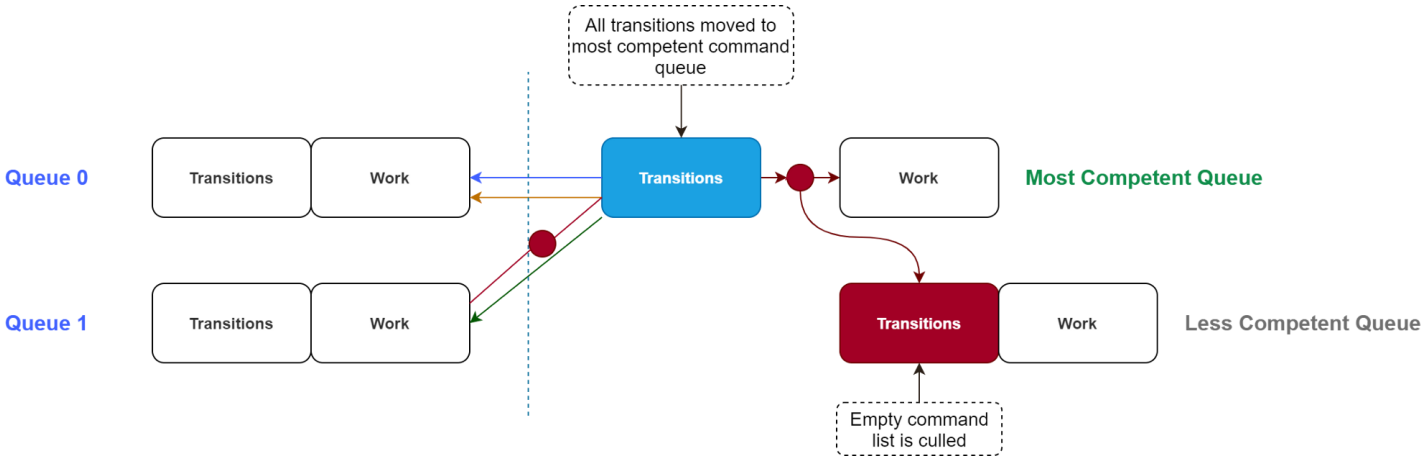
Here a queue reads a resource produced by another and can perform the state transition because all states are compatible. In this case sync between passes found by algorithm discussed earlier stays in the same place and transition can be performed normally before work on receiving queue.

Now consider same cross-queue dependencies, but with a non-supported state transition.

Queue 1 cannot transition a resource it reads so we must find a queue that can. Let's abstract a bit from the graphics API and call it **most competent queue**. If we assume that queue 0 is a graphics queue and queue 1 is compute, then the most competent is queue 0. Now the sync and transition strategy must deviate. We must **reroute** transitions into a separate command list on the most competent queue and rearrange synchronizations to be "plugged" into this command list from both sides.

Now **transitions** must wait for queues involved in state transition rerouting, so we have additional syncs not accounted by sync optimizing algorithm. We also have to

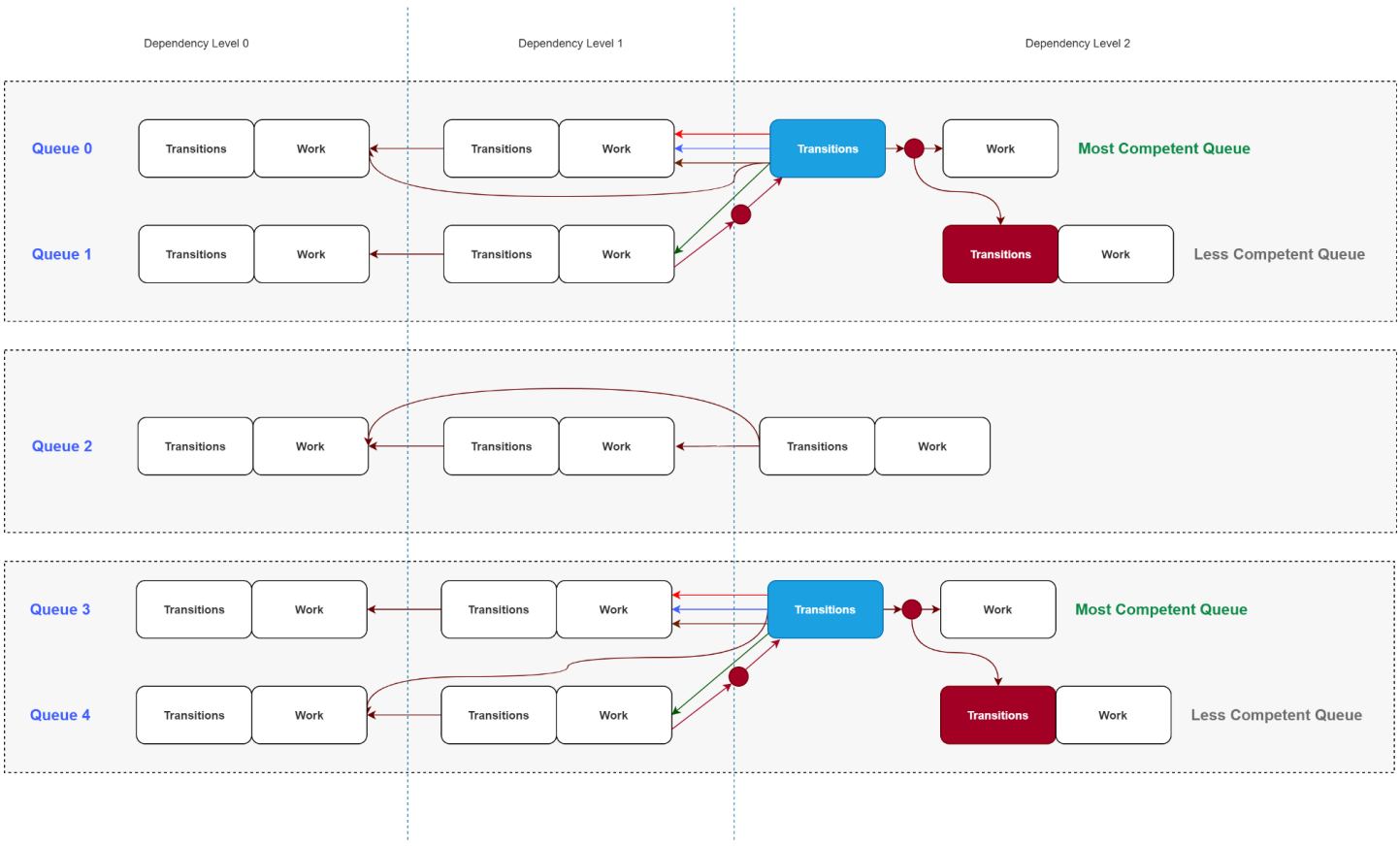
insert a sync after **transitions** and sync our queues with it.



Rerouted transitions. Syncs are “plugged” into a common transitions command list.

What about reading a common resource by multiple queues? As it turns out the strategy is pretty much the same: move transitions out into a separate command list on the most competent queue (even if most competent queue does not actually read this resource) and synchronize queues to those transitions. This way we handle multi-queue reads and unsupported transitions together.

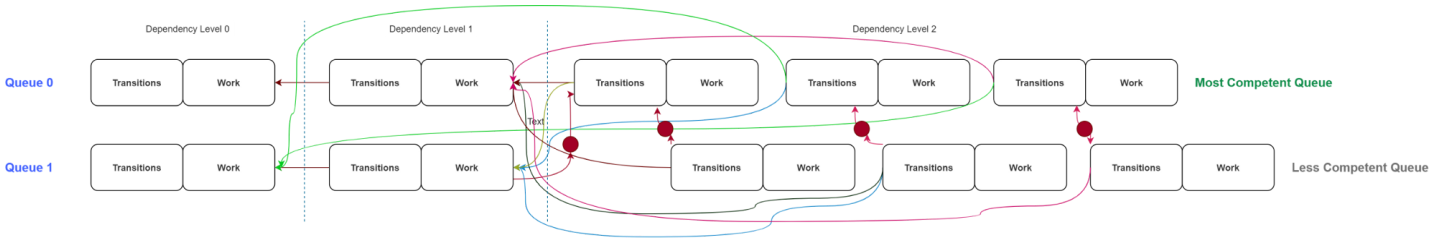
Now let’s address an elephant in the room. What happens if we have cross-queue reads in separate groups of queues?



In a general case with N queues and M passes we could have more sophisticated dependencies with separate collections of mutually dependent queues and most competent queue would have to be determined inside each collection. We now have to ask ourselves, do we *really* need to complicate things even more by introducing such collections? And what happens if more than one collection does not contain a queue competent enough to perform a transition required? The answers are complicated and the whole thing can quickly turn into a synchronization hell, so we need to settle down on one algorithm that would be reasonably manageable and good for performance. But before we do that let's look into render pass reordering and a necessity of moving every single transition into a common command list instead of leaving some of them close to render pass work.

An arbitrary order of render passes inside a dependency level is actually not optimal, because API provides us an ability to kickoff transitions early with split barriers. A metric that would determine this optimality is a distance between Begin and End barriers for each resource transition. It would be nice (performance-wise) to make those distances as large as possible to give GPU more time to hide transition and related stalls. So, do we figure out a heuristic to space out render passes as much as possible and celebrate a performance win? In a vacuum, yes, we do. But upon closer inspection, if we consider cases with multi-queue resource reads and synchronizations they bring, we can see that this transition "spacing" and the transition separation it brings can actually hurt more performance than it wins.

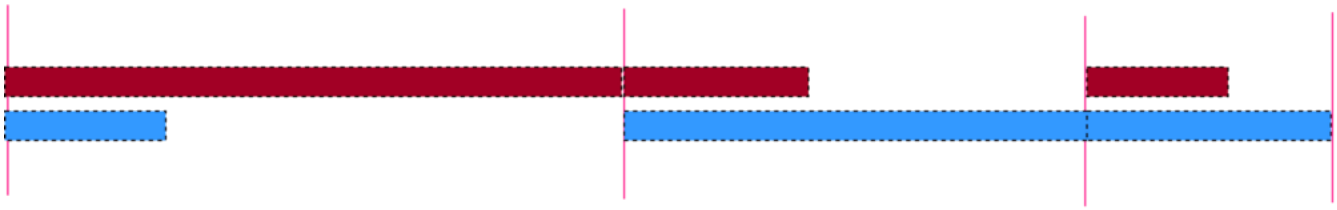
Let's look at example of multi-queue reads by multiple passes in a dependency level.



Separately synced passes in a dependency level due to requirement of separating state transitions

Because we want to increase time distance between transitions we have to perform transitions relevant to each pass separately. That means that we can no longer have a single **rerouted** command list that performs all transitions, we have multiple of those

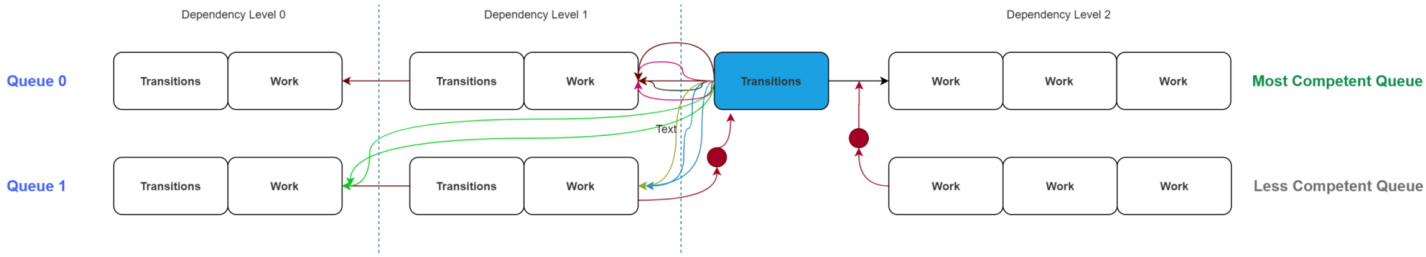
and each pair of work on queue 0 and 1 must be synced separately. And that can kill parallelism.



Bad parallelism

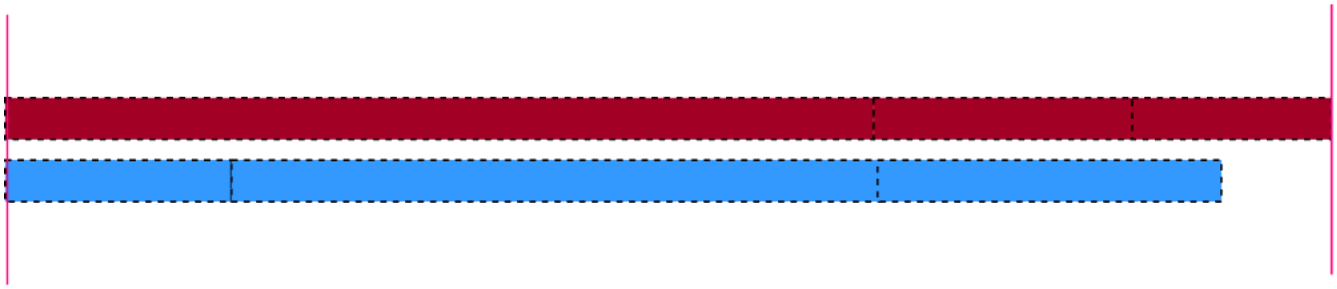
This is how work in a dependency level may look like when we have multiple synchronization points due to cross-queue resource reads. We have nasty stalls on each queue.

Let's see what happens if we move all transitions into a single **rerouted** command list as proposed earlier.



Commonly transitioned and synced passes in a dependency level

In this example we ditched the idea of smart pass reordering and also performing transitions not in a common dedicated command list.



Good parallelism

Now with syncs only at the dependency level beginning we have a nicely paralleled work.

So, when cross-queue dependencies take place in a single dependency level, it looks like moving transitions into a single command list is a better approach than alternatives. But what about queues not involved in complex relations or dependency levels that contain work on only one queue? For those it looks like there are no obstacles for intelligent pass reordering and resource transition separation. And that's true, so we should actually separate the two cases, which leads us to the final algorithm to manage transitions.

> *Gather set of queues involved in multi-queue resource reads using a single condition: if resource is read by more than one queue in a dependency level, add those queues to the set. Then, additionally, detect queues to which we move state transitions due to state incompatibilities and add those queues to the same set. Now take the most competent queue in the set and* ***reroute*** *all transitions and synchronizations to that queue as discussed earlier. We may have queues left that're not involved in such complex relations. We process those separately without sync and transition rerouting.*

Using the Graph: API-specific nuances

Now that the "big" part of the algorithm is behind us let's dive into smaller, but still important topics.

Split barriers

They are a good optimization, but not always applicable. Split barrier is actually two barriers with same Before and After states, so when scheduling a frame we need to check whether both receiving and transmitting queues support transitions with those states. If they do, then we use split barriers, if not, use normal barrier on the receiving queue only.

Command list batching

We gather arrays of command lists that are surrounded by fence waits and/or signals. Each array (batch) is a separate ExecuteCommandLists call.

Going through graph's dependency levels we first deal with passes on queues that require transition rerouting. We move the common transitions command list into its own batch and surround it with waits and signals as discussed earlier. Then we batch passes in that dependency level by adding passes on the same queue to the same batch until a fence signal requirement is detected. If signal is detected, we create a new batch to which subsequent render passes will be added.

Then we batch passes on queues that **do not** require state transition rerouting. Here we add passes to batches similarly, by creating new batch when signal is required, but also we create a new batch when wait is required, the only difference is that pass that requires wait will go into a new batch instead of staying in the old one.

GPU memory load

There is also a concern of GPU memory load. Because we have a restriction of producing (writing) the same resource only once in a frame, we might create more resources than we might have in other renderer designs. We could've gone a DX11-style of solving this problem by introducing pools of resources with identical properties and use some name aliasing mechanism, but that's not actually required nowadays. In newer APIs this problem is solved by memory aliasing achieved through **placed resources** and an ability to overlap memory regions of resources used in disjoint periods of time. An algorithm to find optimal overlapping memory regions using resource usage timelines is a topic on its own and is discussed in a separate article.

Using the Graph: Corner cases

A note on circular dependencies. We might have situations when circular dependency might seem necessary, like when performing temporal re-projection from previous frame and reading a texture before it's written to re-project, then write current frame data into it, then maybe read again in the same frame for different purposes. If we accept such scenarios then we have to engineer manual mechanisms for resolving circular dependencies which is the worse alternative. A better one is to just design render pipeline to avoid circular dependencies at all times. The above example can be solved by introducing two textures instead of one and ping-ponging them between frames without scheduling texture from previous frame for writing in current one.

Also there are read-write resource accesses we might want to have, when blurring an image separably in one render pass, for example. In a graph this looks like a node that has a cycle with itself. The solution is to treat such cases as write only dependencies when building a graph, nothing is actually stopping us from writing and reading a resource inside of the same render pass afterwards.

Another thing that we left out is multiple write dependencies. Writing to the same resource in multiple passes might be desired to reduce memory bandwidth consumption, when render pass writes only to a portion of an existing target. In such case we might have to bite the bullet and engineer additional mechanism for resolving render pass order when multiple write dependencies are encountered. Take a simple

example: **write A**, **read A**, **write A**. Where should reading render pass be put? After first write or after second? One way is to look at initial list of "unordered" render passes and use it as a hint. This of course means that we cannot initially throw render passes into the graph in a random order, the order in which we add them must resemble their final execution order. Other solution might be implementing some sort of list of rules, that graph consults when it encounters an ambiguity.

Conclusion

We worked through quite a few challenges to actually leverage low-level frame control provided by modern graphic APIs and by now can probably imagine what hoops drivers had previously jump through to make everything coherent with immediate-mode approach and without high-level knowledge of frame workload. I don't claim to present best possible solutions, because each engine might require different designs, but I think approaches discussed above are a good compromise.

Thanks

I thank Oleksandr Novytskyi from Ubisoft Kyiv for helping me develop an intuition behind Sufficient Synchronization Index Set idea.

Code

For complete implementation please see my engine PathFinder.

Selected code references:

```
1  void RenderPassGraph::BuildAdjacencyLists()
2  {
3      mAdjacencyLists.resize(mPassNodes.size());
4
5      for (auto nodeIdx = 0; nodeIdx < mPassNodes.size(); ++nodeIdx)
6      {
7          Node& node = mPassNodes[nodeIdx];
8          std::vector<uint64_t>& adjacentNodeIndices = mAdjacencyLists[nodeIdx];
9
10         for (auto otherNodeIdx = 0; otherNodeIdx < mPassNodes.size(); ++otherNodeIdx)
11         {
12             // Do not check dependencies on itself
13             if (nodeIdx == otherNodeIdx) continue;
14
15             Node& otherNode = mPassNodes[otherNodeIdx];
16
17             for (SubresourceName otherNodeReadResource : otherNode.ReadSubresources())
18             {
```

```
                // If other node reads a subresource written by the current node, then it deper
                bool otherNodeDependsOnCurrentNode = node.WrittenSubresources().find(otherNodeF

                if (otherNodeDependsOnCurrentNode)
                {
                    adjacentNodeIndices.push_back(otherNodeIdx);

                    if (node.ExecutionQueueIndex != otherNode.ExecutionQueueIndex)
                    {
                        node.mSyncSignalRequired = true;
                        otherNode.mNodesToSyncWith.push_back(&node);
                    }

                    break;
                }
            }
        }
    }
}
```

build_adjacency_lists.cpp hosted with ♡ by GitHub view raw

Building adjacency lists

```
void RenderPassGraph::DepthFirstSearch(uint64_t nodeIndex, std::vector<bool>& visited, std::vec
{
    if (isCyclic) return;

    visited[nodeIndex] = true;
    onStack[nodeIndex] = true;

    uint64_t adjacencyListIndex = mPassNodes[nodeIndex].mIndexInUnorderedList;

    for (uint64_t neighbour : mAdjacencyLists[adjacencyListIndex])
    {
        if (visited[neighbour] && onStack[neighbour])
        {
            isCyclic = true;
            return;
        }

        if (!visited[neighbour])
        {
            DepthFirstSearch(neighbour, visited, onStack, isCyclic);
        }
    }

```

```
    onStack[nodeIndex] = false;
    mTopologicallySortedNodes.push_back(&mPassNodes[nodeIndex]);
}

void RenderPassGraph::TopologicalSort()
{
    std::vector<bool> visitedNodes(mPassNodes.size(), false);
    std::vector<bool> onStackNodes(mPassNodes.size(), false);

    bool isCyclic = false;

    for (auto nodeIndex = 0; nodeIndex < mPassNodes.size(); ++nodeIndex)
    {
        if (!visitedNodes[nodeIndex])
        {
            DepthFirstSearch(nodeIndex, visitedNodes, onStackNodes, isCyclic);
            assert_format(!isCyclic, "Detected cyclic dependency in pass: ", mPassNodes[nodeInd
        }
    }

    std::reverse(mTopologicallySortedNodes.begin(), mTopologicallySortedNodes.end());
}
```

Topological sort

```
void RenderPassGraph::BuildDependencyLevels()
{
    std::vector<int64_t> longestDistances(mTopologicallySortedNodes.size(), 0);

    uint64_t dependencyLevelCount = 1;

    // Perform longest node distance search
    for (auto nodeIndex = 0; nodeIndex < mTopologicallySortedNodes.size(); ++nodeIndex)
    {
        uint64_t originalIndex = mTopologicallySortedNodes[nodeIndex]->mIndexInUnorderedList;
        uint64_t adjacencyListIndex = originalIndex;

        for (uint64_t adjacentNodeIndex : mAdjacencyLists[adjacencyListIndex])
        {
            if (longestDistances[adjacentNodeIndex] < longestDistances[originalIndex] + 1)
            {
                int64_t newLongestDistance = longestDistances[originalIndex] + 1;
                longestDistances[adjacentNodeIndex] = newLongestDistance;
                dependencyLevelCount = std::max(uint64_t(newLongestDistance + 1), dependencyLev
            }
        }
```

```
    }

    mDependencyLevels.resize(dependencyLevelCount);
    mDetectedQueueCount = 1;

    // Dispatch nodes to corresponding dependency levels.
    // Iterate through unordered nodes because adjacency lists contain indices to
    // initial unordered list of nodes and longest distances also correspond to them.
    for (auto nodeIndex = 0; nodeIndex < mPassNodes.size(); ++nodeIndex)
    {
        Node& node = mPassNodes[nodeIndex];
        uint64_t levelIndex = longestDistances[nodeIndex];
        DependencyLevel& dependencyLevel = mDependencyLevels[levelIndex];
        dependencyLevel.mLevelIndex = levelIndex;
        dependencyLevel.AddNode(&node);
        node.mDependencyLevelIndex = levelIndex;
        mDetectedQueueCount = std::max(mDetectedQueueCount, node.ExecutionQueueIndex + 1);
    }
}
```

longest_path_search.cpp hosted with ♡ by GitHub view raw

Building dependency levels using longest path search

```
void RenderPassGraph::CullRedundantSynchronizations()
{
    // Initialize synchronization index sets
    for (Node& node : mPassNodes)
    {
        node.mSynchronizationIndexSet.resize(mDetectedQueueCount, Node::InvalidSynchronizatior
    }

    for (DependencyLevel& dependencyLevel : mDependencyLevels)
    {
        // First pass: find closest nodes to sync with, compute initial SSIS (sufficient synch
        for (Node* node : dependencyLevel.mNodes)
        {
            // Closest node to sync with on each queue
            std::vector<const Node*> closestNodesToSyncWith{ mDetectedQueueCount, nullptr };

            // Find closest dependencies from other queues for the current node
            for (const Node* dependencyNode : node->mNodesToSyncWith)
            {
                const Node* closestNode = closestNodesToSyncWith[dependencyNode->ExecutionQueu

                if (!closestNode || dependencyNode->LocalToQueueExecutionIndex() > closestNode
                {
```

```
                    closestNodesToSyncWith[dependencyNode->ExecutionQueueIndex] = dependencyNo
                }
            }

            // Get rid of nodes to sync that may have had redundancies
            node->mNodesToSyncWith.clear();

            for (const Node* closestNode : closestNodesToSyncWith)
            {
                if (!closestNode)
                {
                    continue;
                }

                // Update SSIS using closest nodes' indices
                if (closestNode->ExecutionQueueIndex != node->ExecutionQueueIndex)
                {
                    node->mSynchronizationIndexSet[closestNode->ExecutionQueueIndex] = closest
                }

                // Store only closest nodes to sync with
                node->mNodesToSyncWith.push_back(closestNode);
            }

            // Use node's execution index as synchronization index on its own queue
            node->mSynchronizationIndexSet[node->ExecutionQueueIndex] = node->LocalToQueueExec
        }

        // Second pass: cull redundant dependencies by searching for indirect synchronizations
        for (Node* node : dependencyLevel.mNodes)
        {
            // Keep track of queues we still need to sync with
            std::unordered_set<uint64_t> queueToSyncWithIndices;

            // Store nodes and queue syncs they cover
            std::vector<SyncCoverage> syncCoverageArray;

            // Final optimized list of nodes without redundant dependencies
            std::vector<const Node*> optimalNodesToSyncWith;

            for (const Node* nodeToSyncWith : node->mNodesToSyncWith)
            {
                queueToSyncWithIndices.insert(nodeToSyncWith->ExecutionQueueIndex);
            }

            while (!queueToSyncWithIndices.empty())
            {
                uint64_t maxNumberOfSyncsCoveredBySingleNode = 0;
```

```
72
73                  for (auto dependencyNodeIdx = 0u; dependencyNodeIdx < node->mNodesToSyncWith.s
74                  {
75                      const Node* dependencyNode = node->mNodesToSyncWith[dependencyNodeIdx];
76
77                      // Take a dependency node and check how many queues we would sync with
78                      // if we would only sync with this one node. We very well may encounter a
79                      // where by synchronizing with just one node we will sync with more then c
80                      // or even all of them through indirect synchronizations,
81                      // which will make other synchronizations previously detected for this noc
82
83                      std::vector<uint64_t> syncedQueueIndices;
84
85                      for (uint64_t queueIndex : queueToSyncWithIndices)
86                      {
87                          uint64_t currentNodeDesiredSyncIndex = node->mSynchronizationIndexSet[
88                          uint64_t dependencyNodeSyncIndex = dependencyNode->mSynchronizationInd
89
90                          assert_format(currentNodeDesiredSyncIndex != Node::InvalidSynchronizat
91                              "Bug! Node that wants to sync with some queue must have a valid sy
92
93                          if (queueIndex == node->ExecutionQueueIndex)
94                          {
95                              currentNodeDesiredSyncIndex -= 1;
96                          }
97
98                          if (dependencyNodeSyncIndex != Node::InvalidSynchronizationIndex &&
99                              dependencyNodeSyncIndex >= currentNodeDesiredSyncIndex)
100                         {
101                             syncedQueueIndices.push_back(queueIndex);
102                         }
103                     }
104
105                     syncCoverageArray.emplace_back(SyncCoverage{ dependencyNode, dependencyNod
106                     maxNumberOfSyncsCoveredBySingleNode = std::max(maxNumberOfSyncsCoveredBySi
107                 }
108
109                 for (const SyncCoverage& syncCoverage : syncCoverageArray)
110                 {
111                     auto coveredSyncCount = syncCoverage.SyncedQueueIndices.size();
112
113                     if (coveredSyncCount >= maxNumberOfSyncsCoveredBySingleNode)
114                     {
115                         // Optimal list of synchronizations should not contain nodes from the
116                         // because work on the same queue is synchronized automatically and im
117                         if (syncCoverage.NodeToSyncWith->ExecutionQueueIndex != node->Executio
118                         {
```

```
                            optimalNodesToSyncWith.push_back(syncCoverage.NodeToSyncWith);

                            // Update SSIS
                            auto& index = node->mSynchronizationIndexSet[syncCoverage.NodeToSy
                            index = std::max(index, node->mSynchronizationIndexSet[syncCoverag
                        }

                        // Remove covered queues from the list of queues we need to sync with
                        for (uint64_t syncedQueueIndex : syncCoverage.SyncedQueueIndices)
                        {
                            queueToSyncWithIndices.erase(syncedQueueIndex);
                        }
                    }
                }

                // Remove nodes that we synced with from the original list. Reverse iterating
                for (auto syncCoverageIt = syncCoverageArray.rbegin(); syncCoverageIt != syncC
                {
                    node->mNodesToSyncWith.erase(node->mNodesToSyncWith.begin() + syncCoverage
                }
            }

            // Finally, assign an optimal list of nodes to sync with to the current node
            node->mNodesToSyncWith = optimalNodesToSyncWith;
        }
    }
}
```

```
void RenderDevice::GatherResourceTransitionKnowledge(const RenderPassGraph::DependencyLevel& de
{
    mDependencyLevelQueuesThatRequireTransitionRerouting = dependencyLevel.QueuesInvoledInCross

    bool backBufferTransitioned = false;

    for (const RenderPassGraph::Node* node : dependencyLevel.Nodes())
    {
        auto requestTransition = [&](RenderPassGraph::SubresourceName subresourceName, bool isR
        {
            auto [resourceName, subresourceIndex] = mRenderPassGraph->DecodeSubresourceName(sub
            PipelineResourceStorageResource* resourceData = mResourceStorage->GetPerResourceDat

            HAL::ResourceState newState = isReadDependency ?
                resourceData->SchedulingInfo.GetSubresourceCombinedReadStates(subresourceIndex)
                resourceData->SchedulingInfo.GetSubresourceWriteState(subresourceIndex);

            std::optional<HAL::ResourceTransitionBarrier> barrier =
```

```
19                  mResourceStateTracker->TransitionToStateImmediately(resourceData->GetGPUResourc
20
21              if (node->ExecutionQueueIndex == 0 && !backBufferTransitioned)
22              {
23                  std::optional<HAL::ResourceTransitionBarrier> backBufferBarrier =
24                      mResourceStateTracker->TransitionToStateImmediately(mBackBuffer->HALResourc
25
26                  if (backBufferBarrier)
27                  {
28                      mDependencyLevelTransitionBarriers[node->LocalToDependencyLevelExecutionInd
29                  }
30
31                  backBufferTransitioned = true;
32              }
33
34              // Redundant transition
35              if (!barrier)
36              {
37                  return;
38              }
39
40              mDependencyLevelTransitionBarriers[node->LocalToDependencyLevelExecutionIndex()].pu
41
42              // Another reason to reroute resource transitions into another queue is incompatibi
43              // of resource state transitions with receiving queue
44              if (!IsStateTransitionSupportedOnQueue(node->ExecutionQueueIndex, barrier->BeforeSt
45              {
46                  mDependencyLevelQueuesThatRequireTransitionRerouting.insert(node->ExecutionQueu
47              }
48          };
49
50          for (RenderPassGraph::SubresourceName subresourceName : node->ReadSubresources())
51          {
52              requestTransition(subresourceName, true);
53          }
54
55          for (RenderPassGraph::SubresourceName subresourceName : node->WrittenSubresources())
56          {
57              requestTransition(subresourceName, false);
58          }
59
60          for (Foundation::Name resourceName : node->AllResources())
61          {
62              const PipelineResourceStorageResource* resourceData = mResourceStorage->GetPerResou
63              const PipelineResourceSchedulingInfo::PassInfo* passInfo = resourceData->Scheduling
64
65              if (passInfo->NeedsAliasingBarrier)
```

```
            {
                mPerNodeAliasingBarriers[node->GlobalExecutionIndex()].AddBarrier(HAL::Resource
            }

            if (passInfo->NeedsUnorderedAccessBarrier)
            {
                mPassHelpers[node->GlobalExecutionIndex()].UAVBarriers.AddBarrier(HAL::Unordere
            }
        }
    }
}
```

queues

```
void RenderDevice::BatchCommandListsWithTransitionRerouting(const RenderPassGraph::DependencyLe
{
    if (mDependencyLevelQueuesThatRequireTransitionRerouting.empty())
    {
        return;
    }

    uint64_t mostCompetentQueueIndex = FindMostCompetentQueueIndex(mDependencyLevelQueuesThatRe
    mReroutedTransitionsCommandLists[dependencyLevel.LevelIndex()] = AllocateCommandListForQueu
    CommandListPtrVariant& commandListVariant = mReroutedTransitionsCommandLists[dependencyLeve
    HAL::ComputeCommandListBase* transitionsCommandList = GetComputeCommandListBase(commandList

    std::vector<CommandListBatch>& mostCompetentQueueBatches = mCommandListBatches[mostCompeter
    CommandListBatch* reroutedTransitionsBatch = &mostCompetentQueueBatches.emplace_back();
    reroutedTransitionsBatch->FenceToSignal = &FenceForQueueIndex(mostCompetentQueueIndex);
    reroutedTransitionsBatch->CommandLists.emplace_back(GetHALCommandListVariant(commandListVar
    uint64_t reroutedTransitionsBatchIndex = mostCompetentQueueBatches.size() - 1;

    HAL::ResourceBarrierCollection reroutedTransitionBarrires;

    std::vector<CommandListBatch*> dependencyLevelPerQueueBatches{ mQueueCount, nullptr };

    for (RenderPassGraph::Node::QueueIndex queueIndex : mDependencyLevelQueuesThatRequireTransi
    {
        // Make rerouted transitions wait for fences from involved queues
        if (queueIndex != mostCompetentQueueIndex)
        {
            mostCompetentQueueBatches[reroutedTransitionsBatchIndex].FencesToWait.insert(&Fence
        }

        for (const RenderPassGraph::Node* node : dependencyLevel.NodesForQueue(queueIndex))
```

```
32          {
33              // A special case of waiting for BVH build fence, if of course pass is not executed
34              if (mRenderPassGraph->FirstNodeThatUsesRayTracing() == node && mBVHBuildsQueueIndex
35              {
36                  mostCompetentQueueBatches[reroutedTransitionsBatchIndex].FencesToWait.insert(&F
37              }
38
39              if (!dependencyLevelPerQueueBatches[node->ExecutionQueueIndex])
40              {
41                  dependencyLevelPerQueueBatches[node->ExecutionQueueIndex] = &mCommandListBatche
42              }
43
44              CommandListBatch* currentBatchInCurrentDependencyLevel = dependencyLevelPerQueueBat
45
46              // Make command lists in a batch wait for rerouted transitions
47              currentBatchInCurrentDependencyLevel->FencesToWait.insert(mostCompetentQueueBatches
48              currentBatchInCurrentDependencyLevel->CommandLists.push_back(GetHALCommandListVaria
49
50              uint64_t currentCommandListBatchIndex = mCommandListBatches[queueIndex].size() - 1;
51              CollectNodeTransitions(node, currentCommandListBatchIndex, reroutedTransitionBarrir
52
53              if (node->IsSyncSignalRequired())
54              {
55                  currentBatchInCurrentDependencyLevel->FenceToSignal = &FenceForQueueIndex(node-
56                  dependencyLevelPerQueueBatches[node->ExecutionQueueIndex] = &mCommandListBatche
57              }
58          }
59
60          // Do not leave empty batches
61          if (mCommandListBatches[queueIndex].back().CommandLists.empty())
62          {
63              mCommandListBatches[queueIndex].pop_back();
64          }
65      }
66
67      transitionsCommandList->InsertBarriers(reroutedTransitionBarrires);
68  }
```

```
1   void RenderDevice::BatchCommandListsWithoutTransitionRerouting(const RenderPassGraph::Dependenc
2   {
3       for (auto queueIdx = 0u; queueIdx < mRenderPassGraph->DetectedQueueCount(); ++queueIdx)
4       {
5           if (mDependencyLevelQueuesThatRequireTransitionRerouting.find(queueIdx) != mDependencyL
6           {
```

```
            continue;
        }

        auto& nodesForQueue = dependencyLevel.NodesForQueue(queueIdx);

        if (nodesForQueue.empty())
        {
            continue;
        }

        for (const RenderPassGraph::Node* node : nodesForQueue)
        {
            if (mCommandListBatches[queueIdx].empty())
            {
                mCommandListBatches[queueIdx].emplace_back();
            }

            CommandListBatch* currentBatch = &mCommandListBatches[queueIdx].back();

            bool usesRT = mRenderPassGraph->FirstNodeThatUsesRayTracing() == node;

            if (!node->NodesToSyncWith().empty() || usesRT)
            {
                if (!currentBatch->CommandLists.empty())
                {
                    currentBatch = &mCommandListBatches[queueIdx].emplace_back();
                }

                for (const RenderPassGraph::Node* nodeToWait : node->NodesToSyncWith())
                {
                    currentBatch->FencesToWait.insert(&FenceForQueueIndex(nodeToWait->Execution
                }

                if (usesRT)
                {
                    currentBatch->FencesToWait.insert(&FenceForQueueIndex(mBVHBuildsQueueIndex)
                }
            }

            // On queues that do not require transition rerouting each node will have its own t
            HAL::ResourceBarrierCollection nodeBarriers{};
            uint64_t currentCommandListBatchIndex = mCommandListBatches[queueIdx].size() - 1;

            CollectNodeTransitions(node, currentCommandListBatchIndex, nodeBarriers);

            if (nodeBarriers.BarrierCount() > 0)
            {
                mPassCommandLists[node->GlobalExecutionIndex()].TransitionsCommandList = Alloca
```

```
                CommandListPtrVariant& cmdListVariant = mPassCommandLists[node->GlobalExecutio
                HAL::ComputeCommandListBase* transitionsCommandList = GetComputeCommandListBas
                transitionsCommandList->InsertBarriers(nodeBarriers);
                currentBatch->CommandLists.push_back(GetHALCommandListVariant(cmdListVariant));
            }

            currentBatch->CommandLists.push_back(GetHALCommandListVariant(mPassCommandLists[no

            if (node->IsSyncSignalRequired())
            {
                currentBatch->FenceToSignal = &FenceForQueueIndex(queueIdx);
                mCommandListBatches[queueIdx].emplace_back();
            }
        }

        // Do not leave empty batches
        if (mCommandListBatches[queueIdx].back().CommandLists.empty())
        {
            mCommandListBatches[queueIdx].pop_back();
        }
    }
}
```

```
void RenderDevice::CollectNodeTransitions(const RenderPassGraph::Node* node, uint64_t currentC
{
    const std::vector<SubresourceTransitionInfo>& nodeTransitionBarriers = mDependencyLevelTra
    const HAL::ResourceBarrierCollection& nodeAliasingBarriers = mPerNodeAliasingBarriers[node

    collection.AddBarriers(nodeAliasingBarriers);

    for (const SubresourceTransitionInfo& transitionInfo : nodeTransitionBarriers)
    {
        auto previousTransitionInfoIt = mSubresourcesPreviousTransitionInfo.find(transitionInf
        bool foundPreviousTransition = previousTransitionInfoIt != mSubresourcesPreviousTransit
        bool subresourceTransitionedAtLeastOnce = foundPreviousTransition && previousTransitio

        if (!subresourceTransitionedAtLeastOnce)
        {
            bool implicitTransitionPossible = Memory::ResourceStateTracker::CanResourceBeImpli
                *transitionInfo.Resource, transitionInfo.TransitionBarrier.BeforeStates(), tra

            if (implicitTransitionPossible)
            {
                continue;
```

```
22              }
23          }
24
25          if (foundPreviousTransition)
26          {
27              const SubresourcePreviousTransitionInfo& previousTransitionInfo = previousTransitic
28
29              // Split barrier is only possible when transmitting queue supports transitions for
30              bool isSplitBarrierPossible = IsStateTransitionSupportedOnQueue(
31                  previousTransitionInfo.Node->ExecutionQueueIndex, transitionInfo.TransitionBarr
32              );
33
34              // There is no sense in splitting barriers between two adjacent render passes.
35              // That will only double the amount of barriers without any performance gain.
36              bool currentNodeIsNextToPrevious = node->LocalToQueueExecutionIndex() - previousTra
37
38              if (isSplitBarrierPossible && !currentNodeIsNextToPrevious)
39              {
40                  auto [beginBarrier, endBarrier] = transitionInfo.TransitionBarrier.Split();
41                  collection.AddBarrier(endBarrier);
42                  mPerNodeBeginBarriers[previousTransitionInfo.Node->GlobalExecutionIndex()].AddE
43              }
44              else
45              {
46                  collection.AddBarrier(transitionInfo.TransitionBarrier);
47              }
48          }
49          else
50          {
51              collection.AddBarrier(transitionInfo.TransitionBarrier);
52          }
53
54          mSubresourcesPreviousTransitionInfo[transitionInfo.SubresourceName] = { node, currentCo
55      }
56  }
```

Selecting and using barriers for render pass

Executing and synchronizing batched command lists

Queue Synchronization | Resource Barriers | Acyclic Graph | Directx 12 | Frame Graph


Download on the
App Store
GET IT ON
Google Play