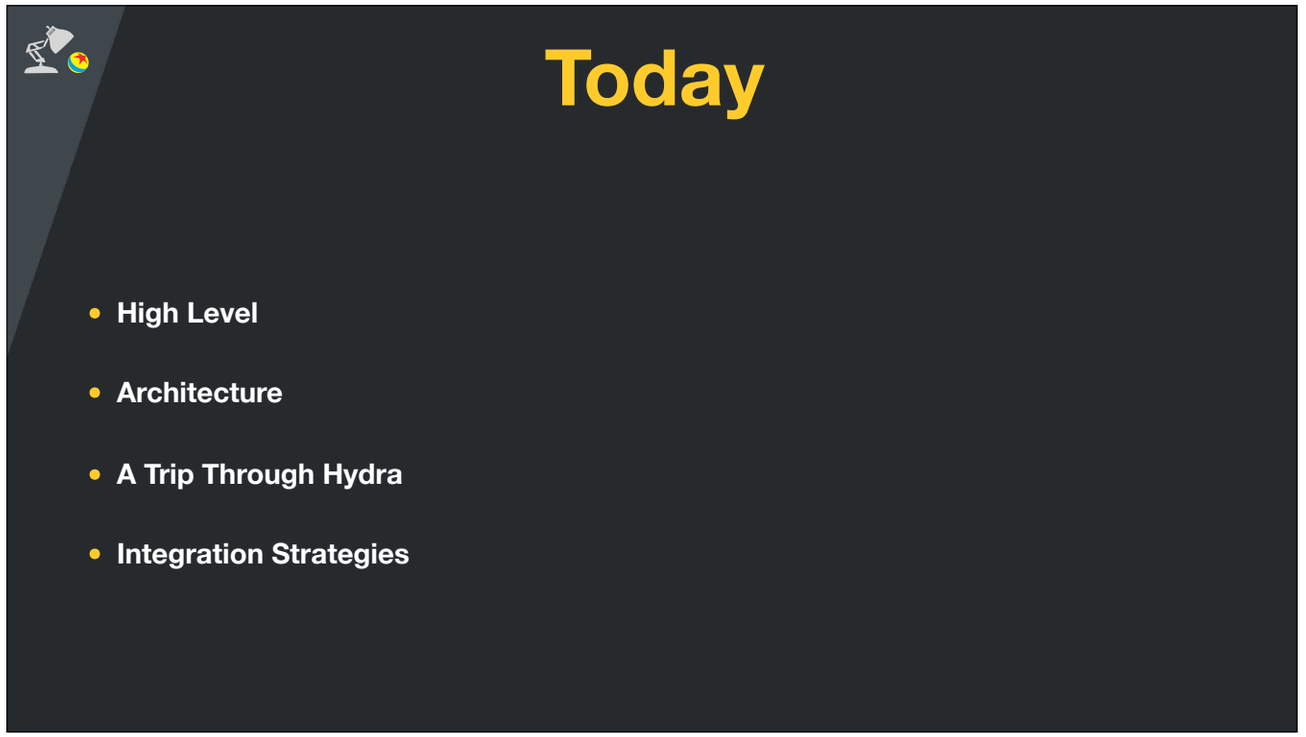
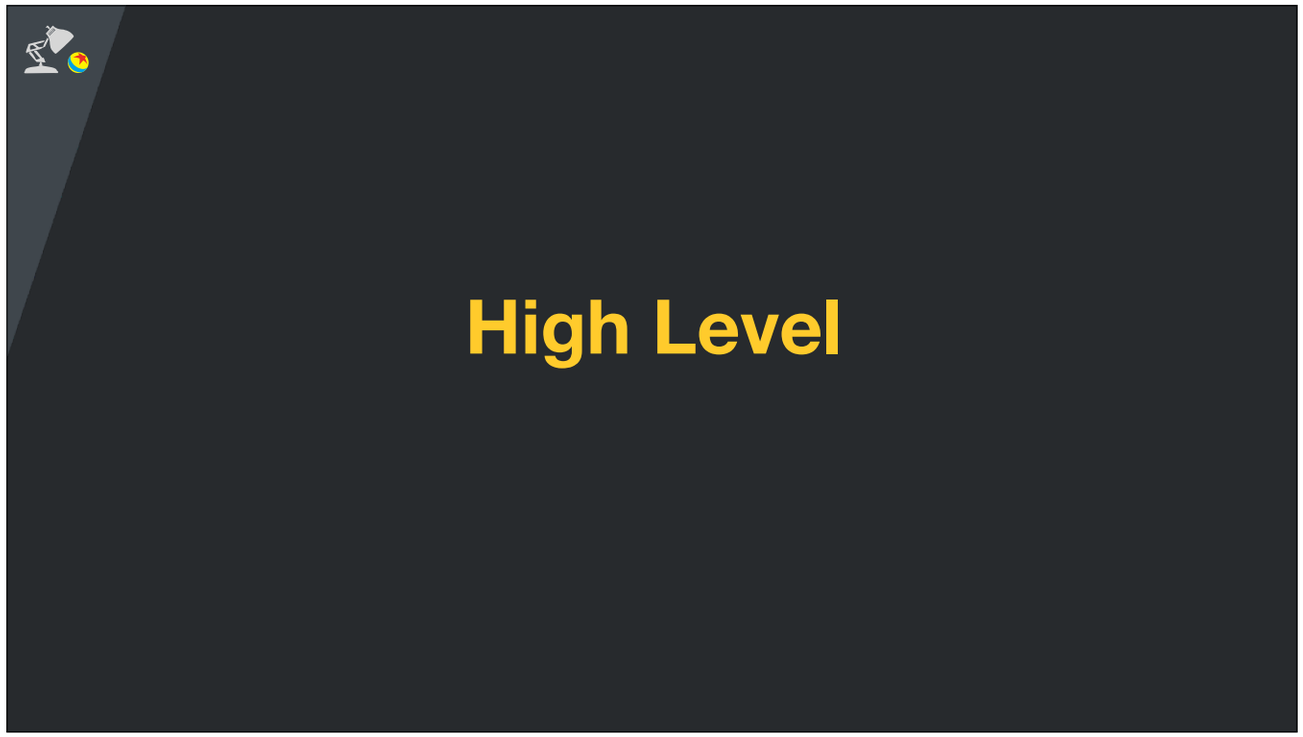

PIXAR
SOFTWARE R&D
Hydra
Pol Jeremias-Vila
©Disney/Pixar

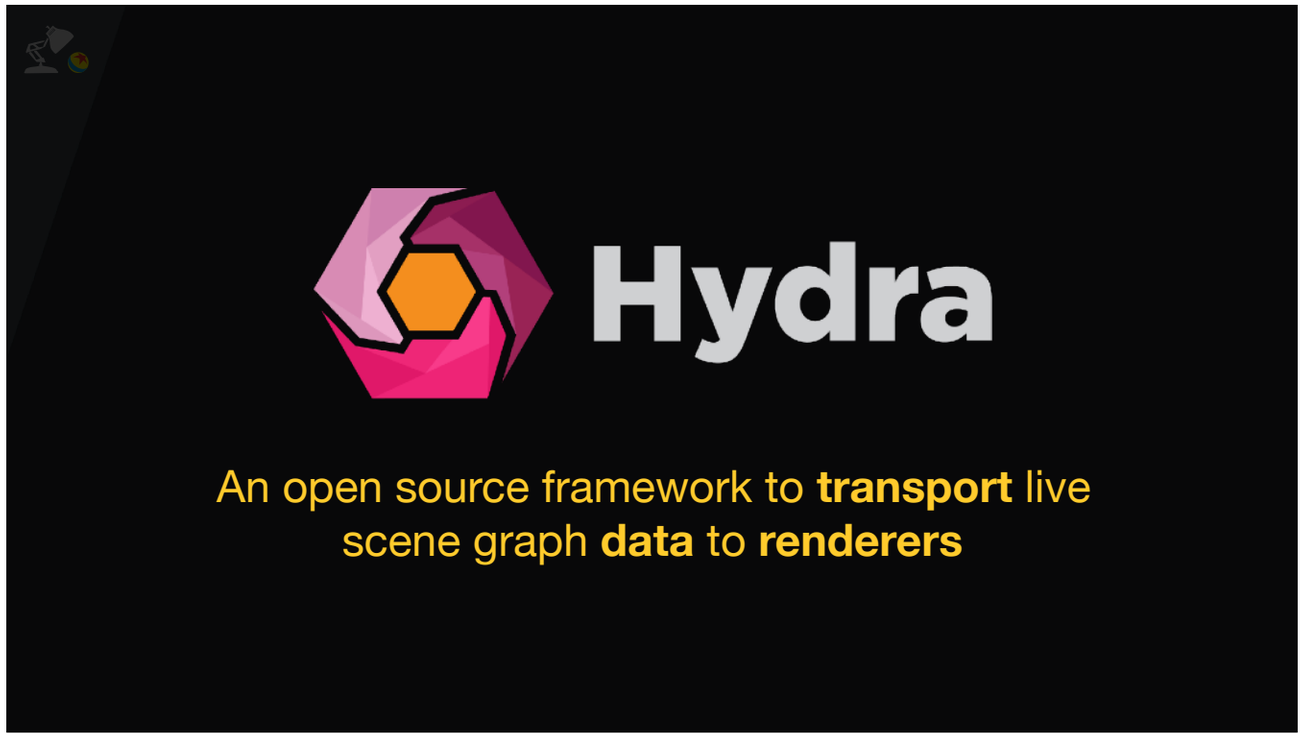


Hi! My name is Pol and I will be talking about Hydra today.

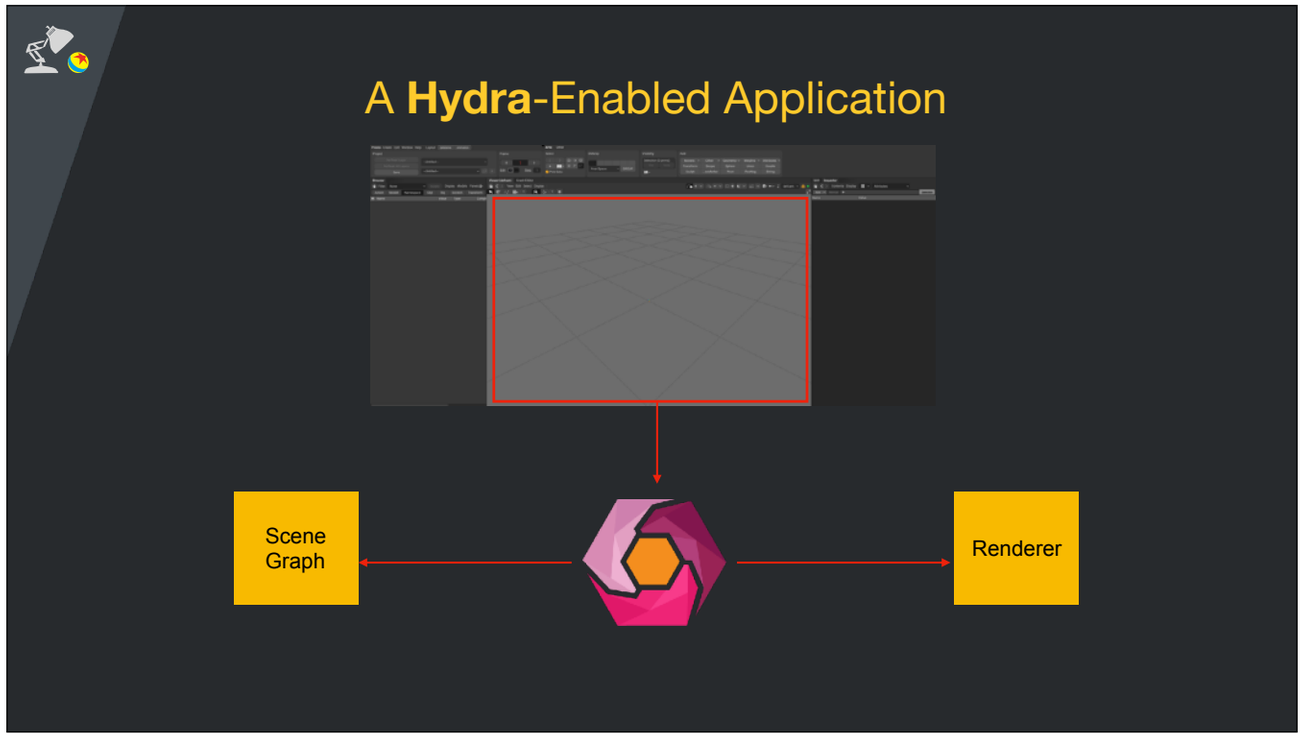
- We will start discussing the **high level** of Hydra, this will be short section.
- Then we will be talking about the details of the **Architecture**.
- After we will walk through a simple scene graph and renderer to showcase how Hydra works.
- Finally we will talk about how to **integrate** Hydra in your application.



Let’s begin with the High Level.



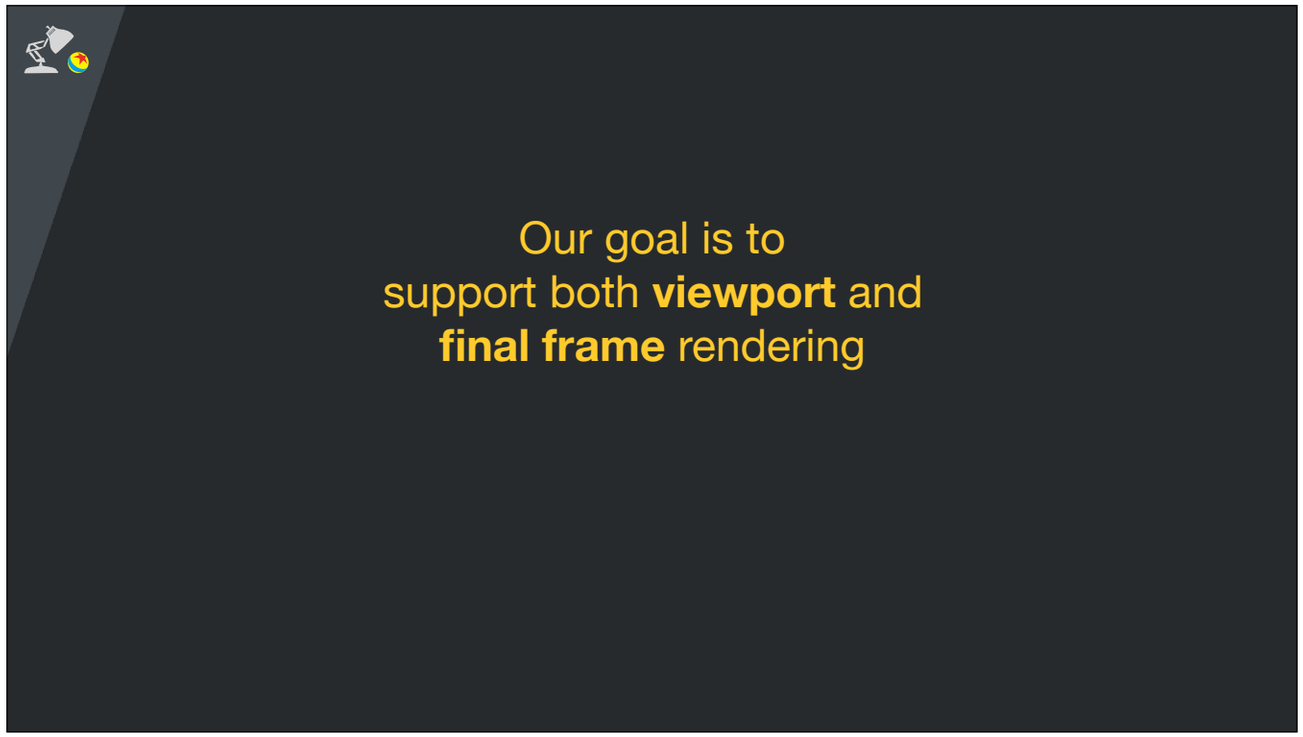
You might be thinking… what is Hydra? Well… Hydra is a project that started years ago as a **high performance rendering engine** that used OpenGL to image USD stages as fast a possible.
Over the years though, we have **abstracted** away both! OpenGL and USD.
(next) — —
Today, Hydra is an open source framework to transport live scene graph data to renderers. But, you may be thinking… this is very abstract, what does this even mean?



Let’s look at an example.

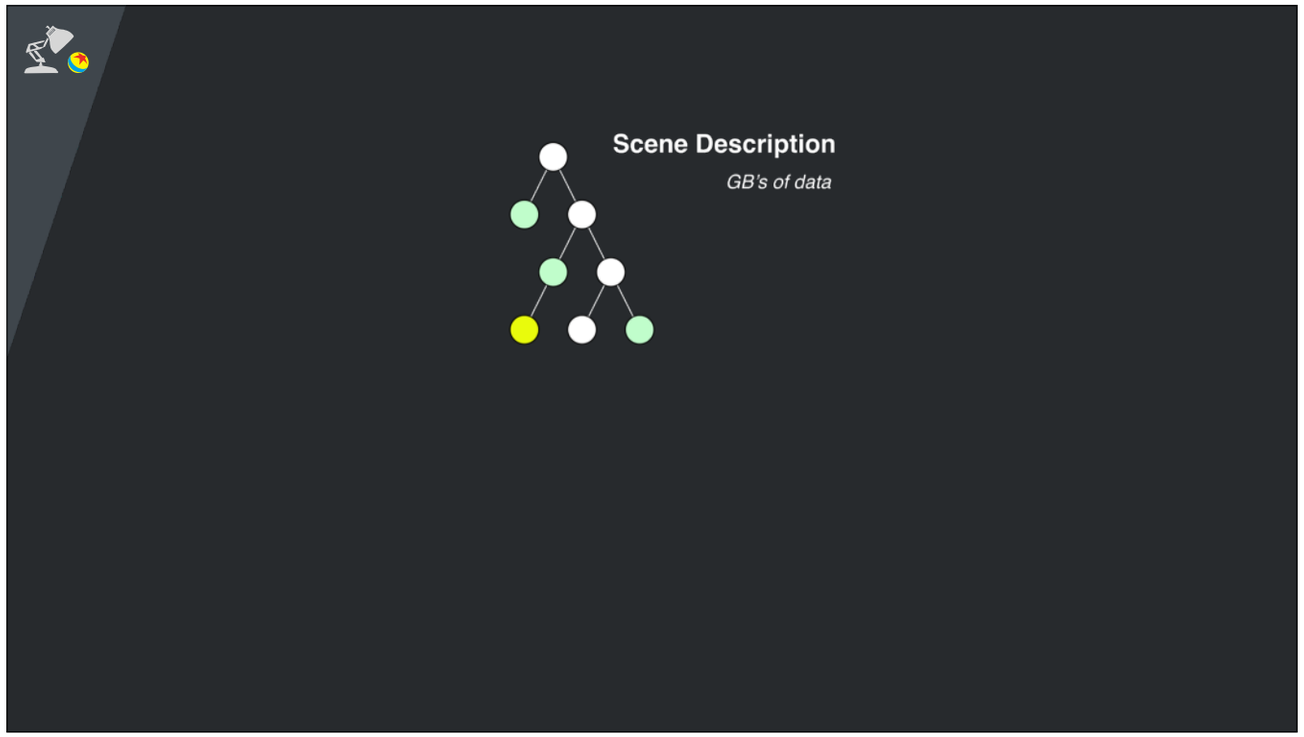
Imagine you have this application which has a viewport. If you integrate Hydra into this viewport, you have suddenly separate the concepts of scene graphs and renderers.

And you have done that in such a way that you just created plugin points in your application to connect a scene graph and a renderer.



We have just looked at an example where Hydra is connected to the viewport, but that does not need to be the case.

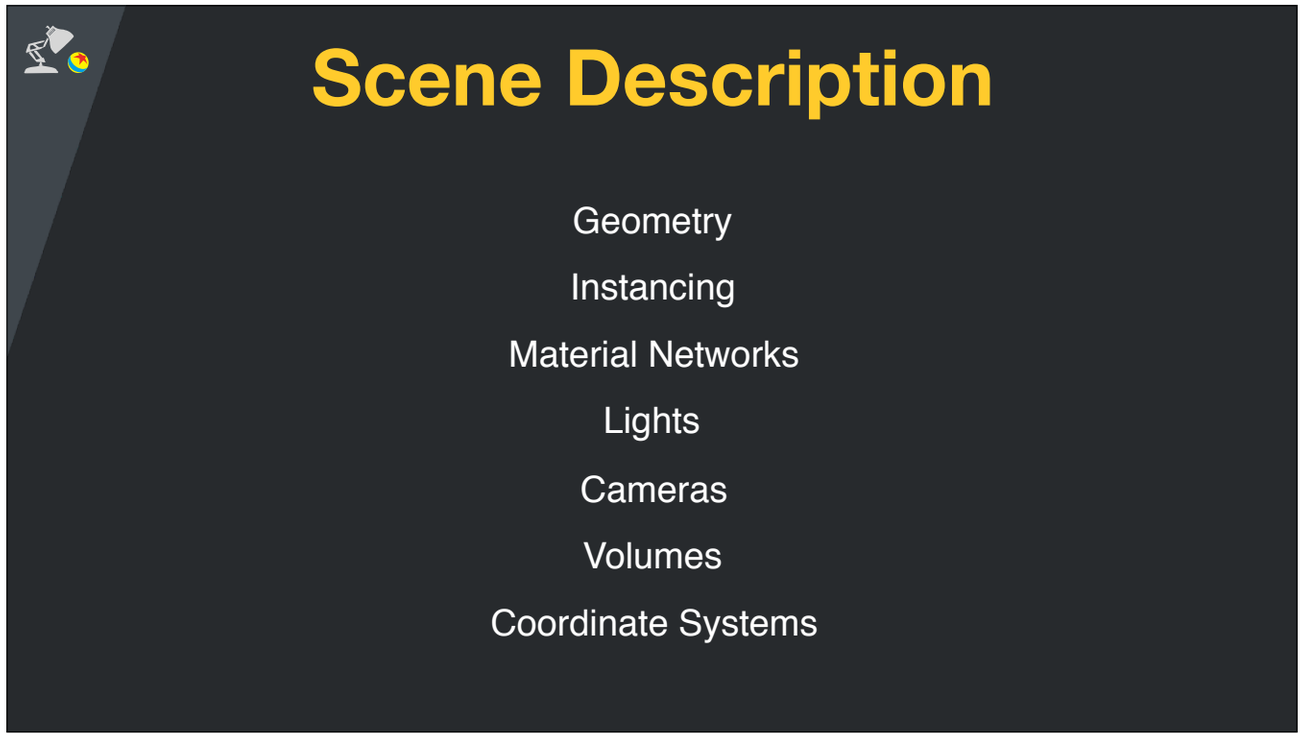
Our goal is to support both, viewport and final frame. Now, let’s talk about rendering for a second.



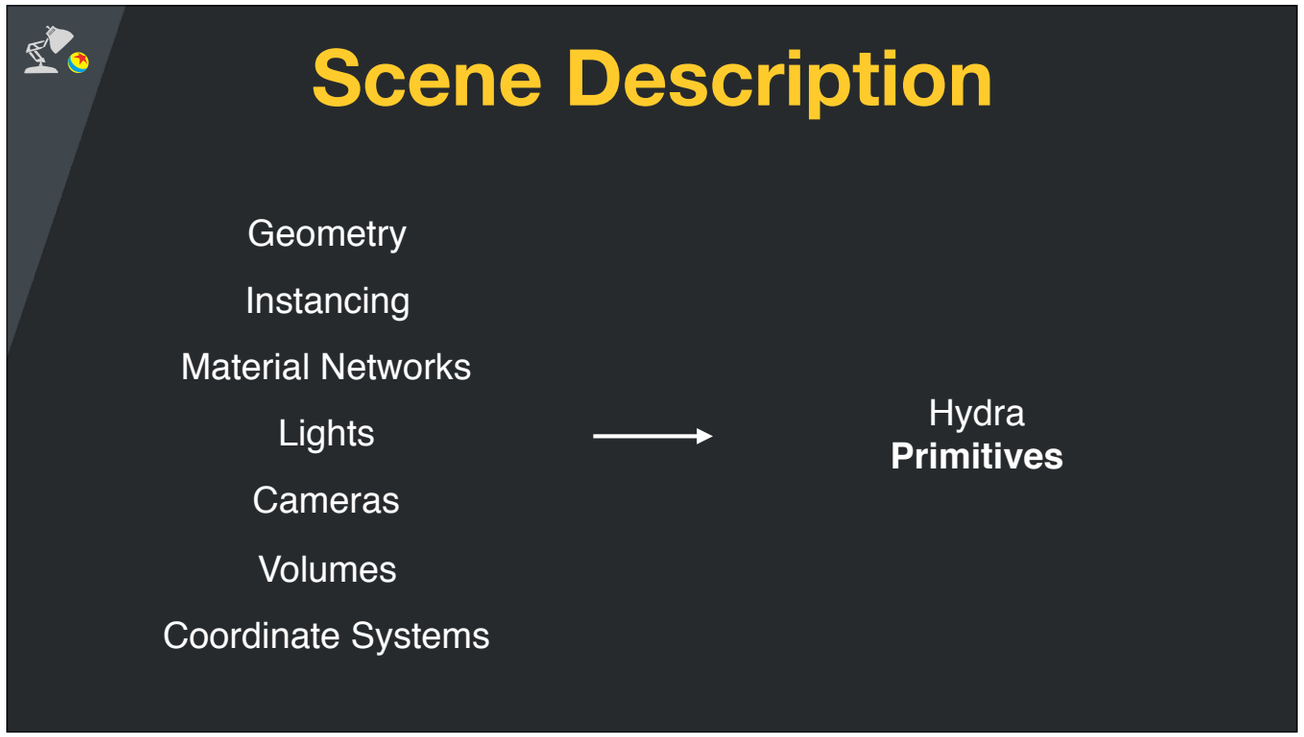
This is a scene description. It has many nodes and of course, if this was a real movie scene, it is most likely huge, as in gigabytes of data.

Each of these nodes can describe different characteristics of the scene.

There are many things you can put on those nodes, for instance …

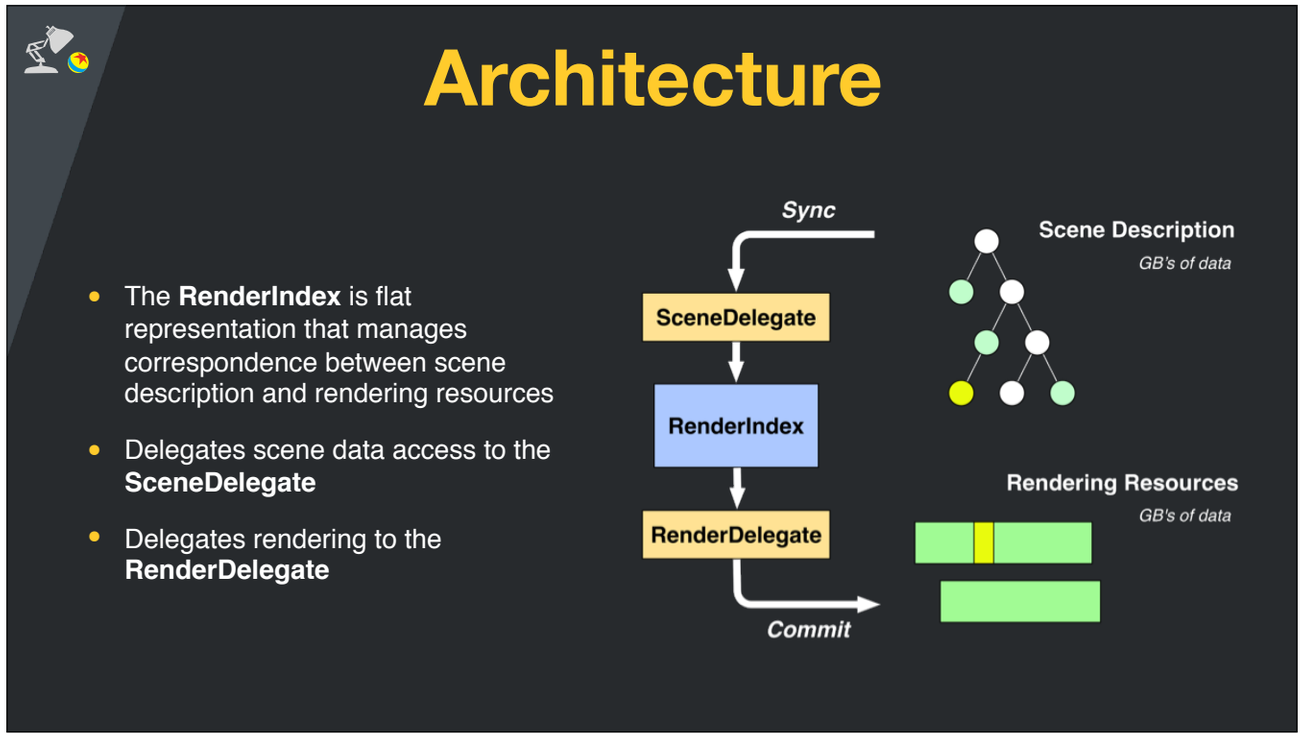


Geometry, instancing, material networks, lights, cameras, volumes, coordinate systems, shading nodes, and many many more.



In Hydra, we convert these descriptions of the scene to Hydra Primitives.

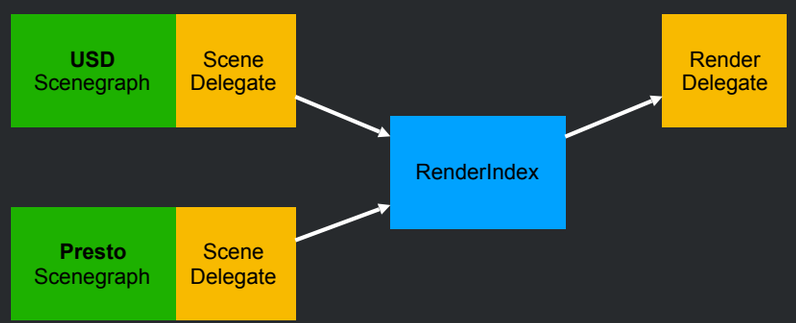
This is how we deal with transporting primitives through Hydra, but how does Hydra work? For that, let’s look at the architecture from a high level.



Three Hydra blocks are necessary to convert generic scene description into generic rendering resources:
- The central piece is the **RenderIndex**, a flat representation that manages correspondence between scene description and rendering resources.
- The RenderIndex will delegate scene data access to the **Scene Delegate**. This way, we can abstract the concept of scene graphs from the core.
- Similarly, the RenderIndex will delegate rendering to the **Render Delegate** just we can abstract the concept of rendering.

This might sound like a lot of machinery but we will talk about performance in just a second. First, let’s look at the possibilities that this brings to the table.

Multiple Scene Graphs

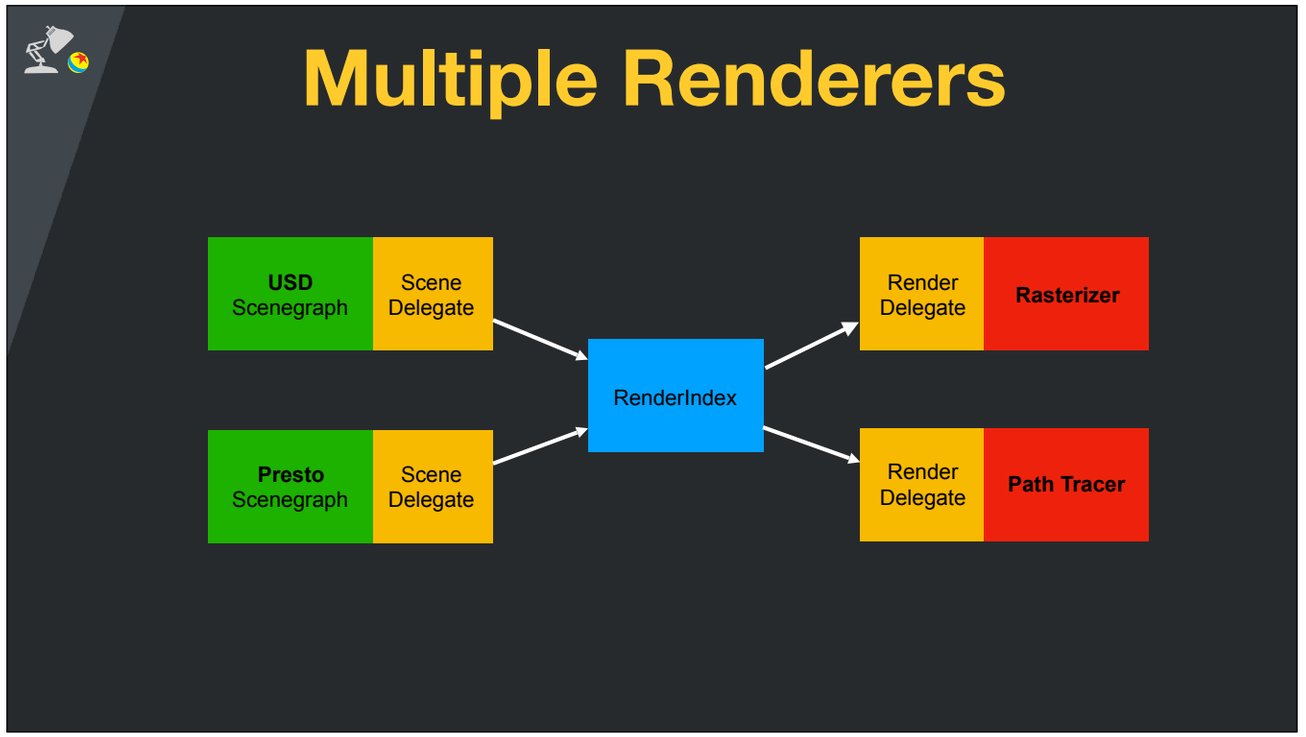


Since we have abstracted the concept of scene graphs from the core of Hydra, we can actually have multiple scene graphs feeding Hydra, for instance a USD scene graph and a Presto scene graph.

Presto is our animation tool.

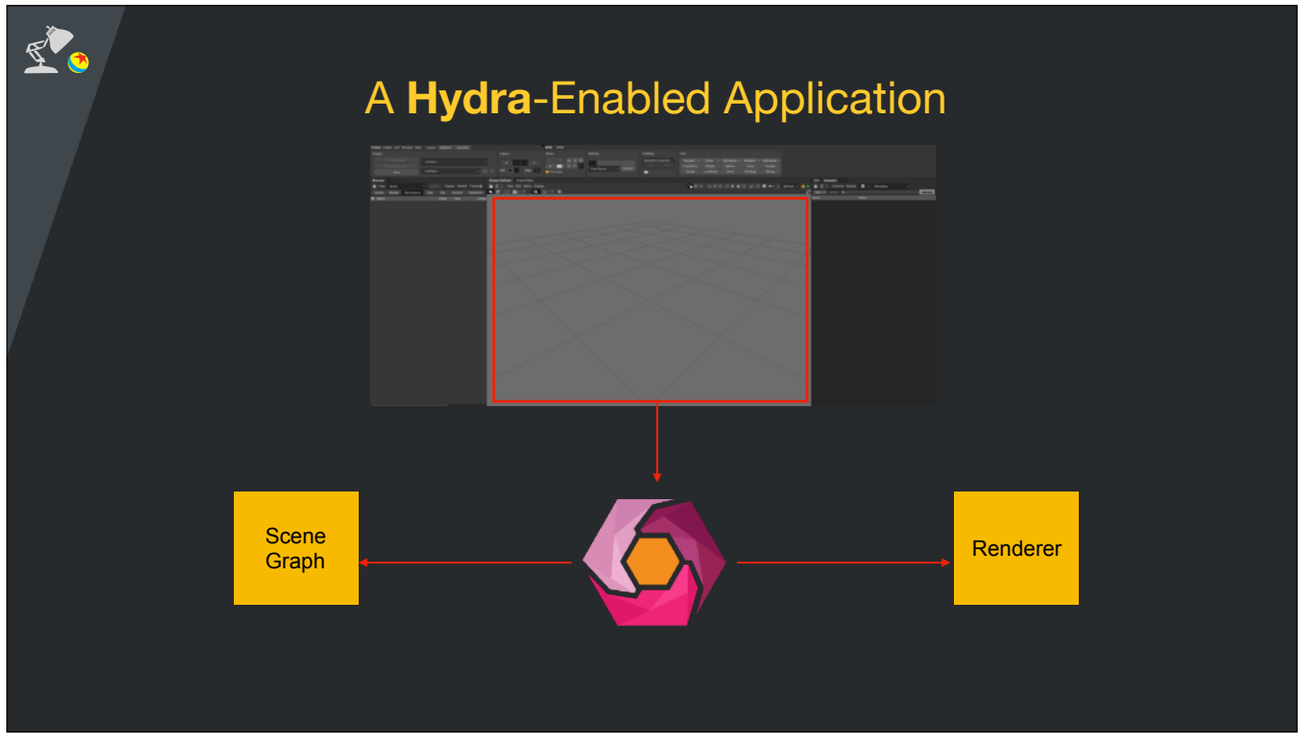
And this is not limited to two scene graphs, you can have as many as you want or need.

This data is all indexed in Hydra and made available to the renderer connected.



Of course, since we have also abstracted the renderer… you can have multiple renderers connected to Hydra.

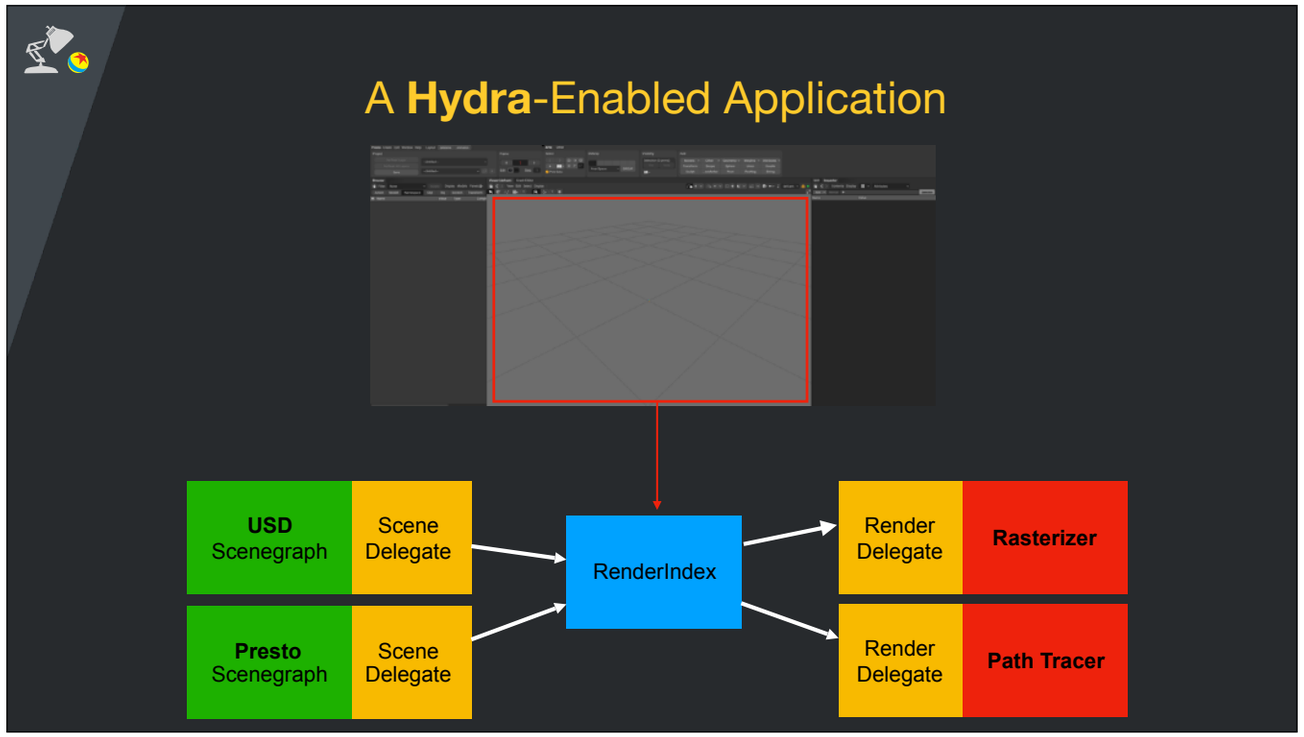
For instance, a rasterizer and a path tracer.



So, going back to our initial drawing!

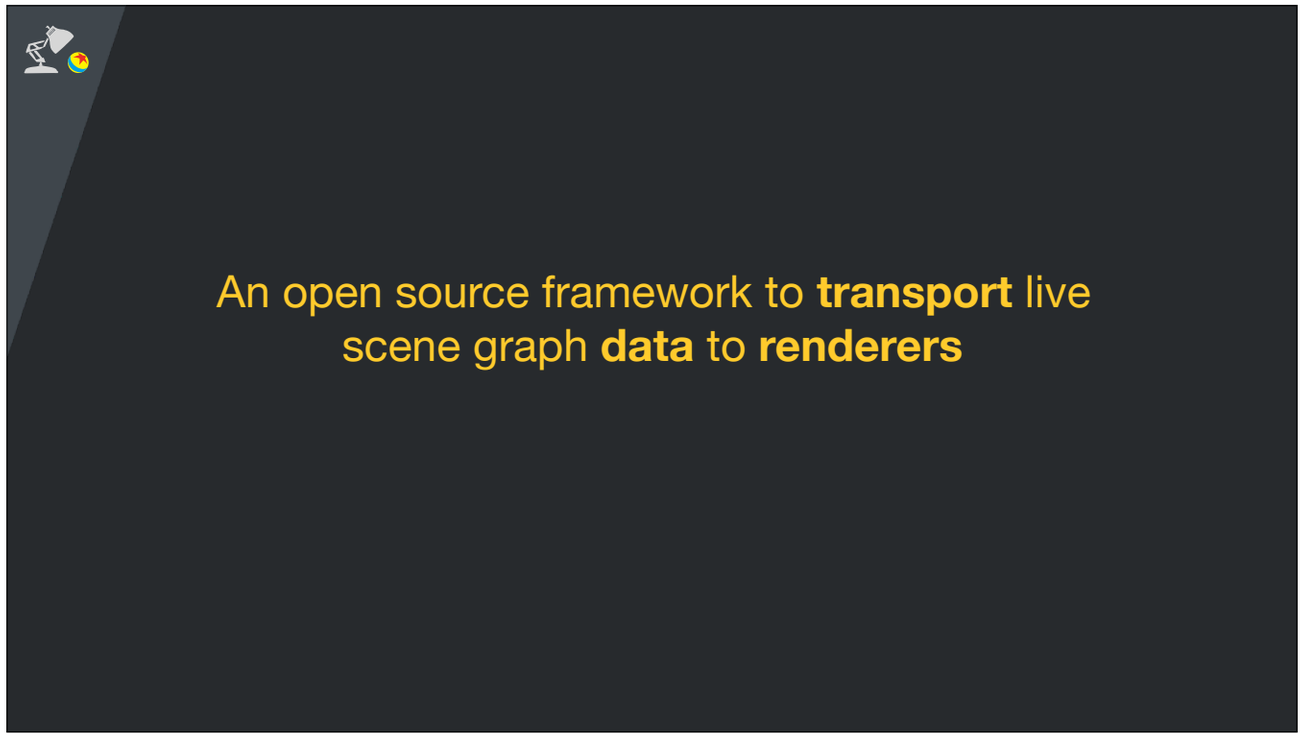
Once you have integrated Hydra in your application, you can now connect a scene graph and a renderer.

But it does not need to be one.



You can actually have multiple sources of data feeding Hydra.

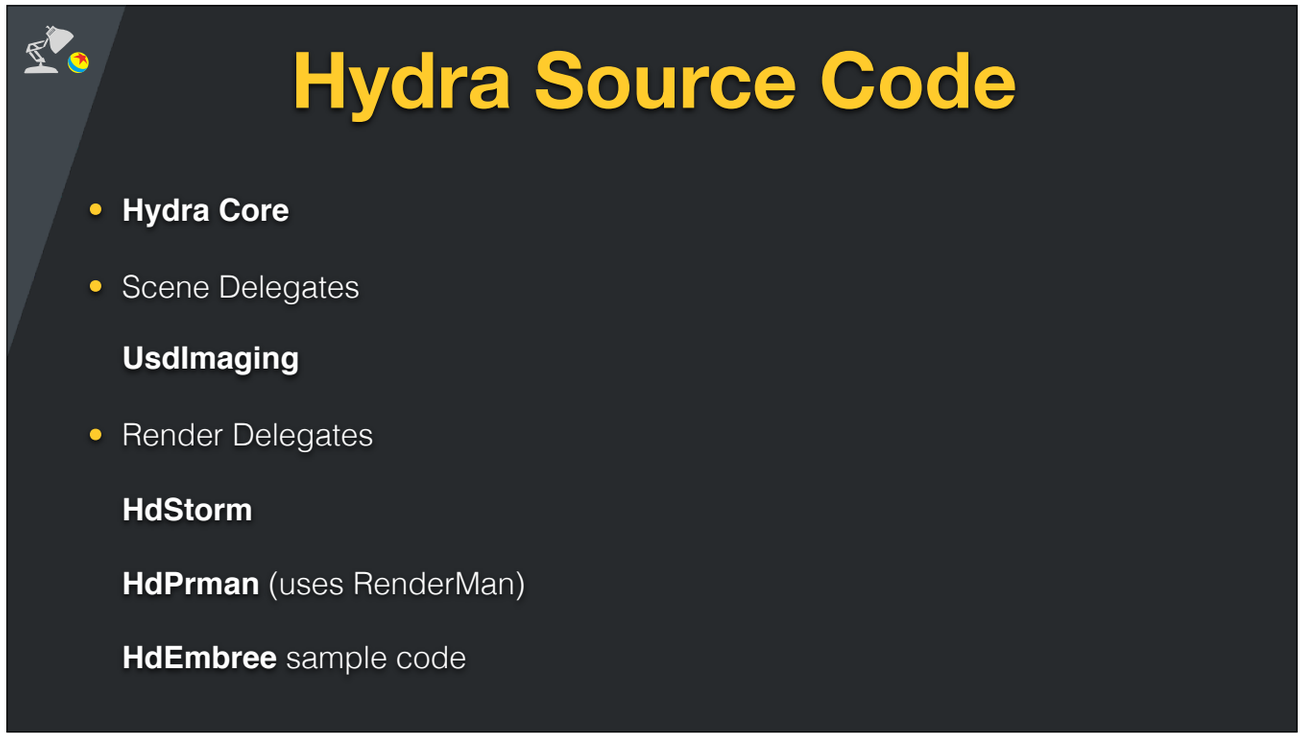
And on top of that, you gain the possibility to connect a different renderer to your application.



So, now going all the way back to the beginning of the presentation.

Using our open source framework you can transports live scene graph data to renderers.

So far, in this presentation, we have talked about the scene graph data and the renderer… But we have not talk about open source! Let’s talk about that.



In the open source, you will find the Hydra core with the RenderIndex and many other components that we will discuss throughout the presentation today.

You will find the USD Scene Delegate, which we call UsdImaging.

Finally, you will find three render delegates:
- HdStorm, which is our real time rasterizer for preview, it used to called Stream.
- HdPrman, which gives you great path tracing. This is just the render delegate, you will still need to get RenderMan for it to work.
- HdEmbree, which is sample code that is useful when you want to build your own.

HdStorm Render Delegate

Efficient mesh batching

Multi-level instanced drawing

OpenSubdiv integration

Highlighting for faces, points, edges

GLSL materials

Compute kernels

Support Udims, UV, Ptex textures

UsdPreviewSurface support

Order Independent Transparency

Support for Windows / Mac / Linux

And more!

HdPrman Render Delegate

Meshes, curves or points

Instancing

Non-trivial material networks

Volumes

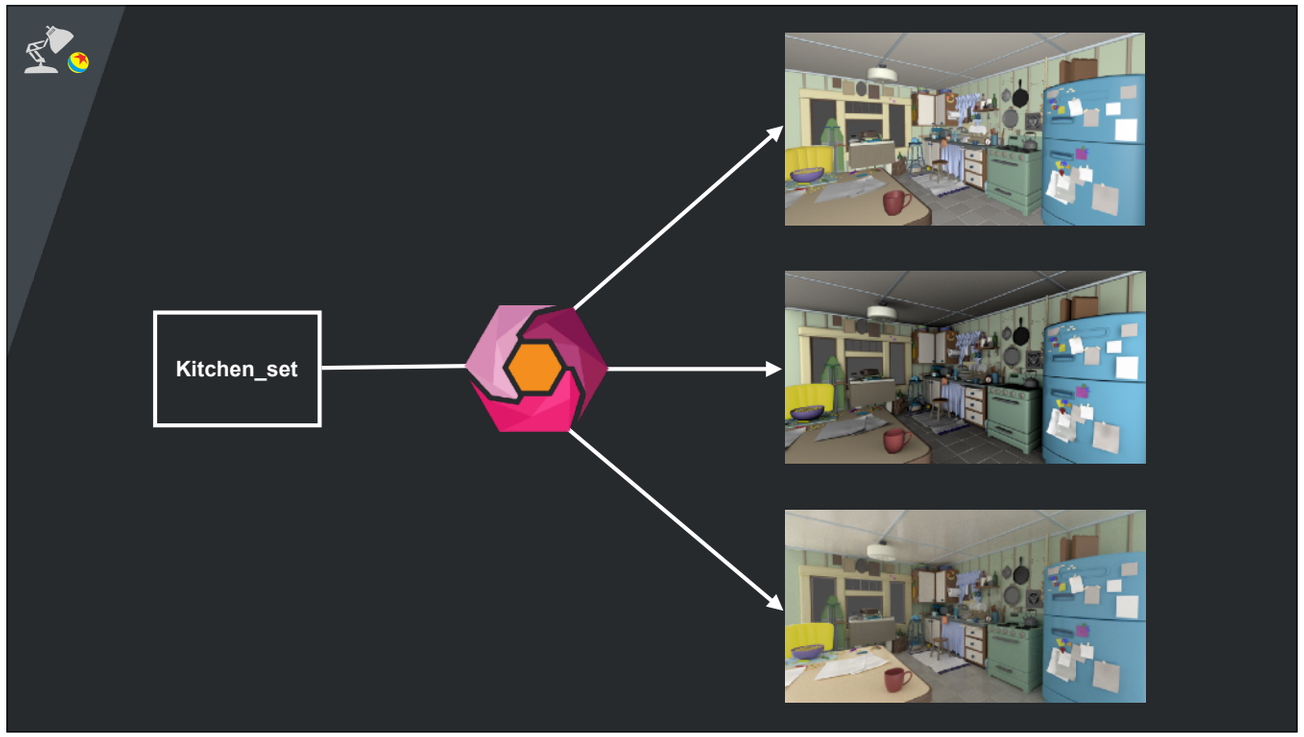
Coordinate systems

Basic support for UsdPreviewSurface

Computations

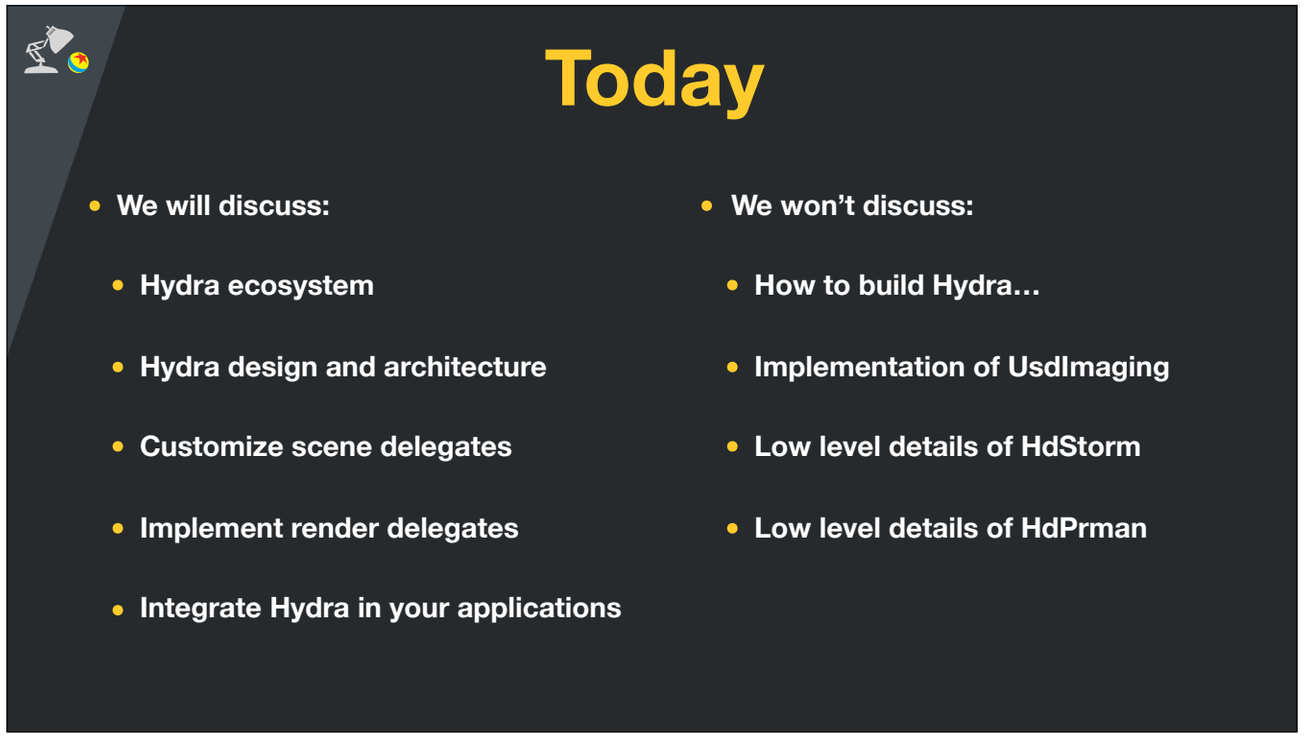
Support for picking and highlighting

And more!



With all this different building blocks that we open source. You could integrate Hydra in your application, download our kitchen set and…. Render with HdStorm, or maybe HdEmbree, or maybe using the HdPrman plugin to Renderman

You will get consistent data transport across renderers.



Finally, I wanted to clarify about we will be discussing today and what we will not be talking about, because Hydra is a big ecosystem.

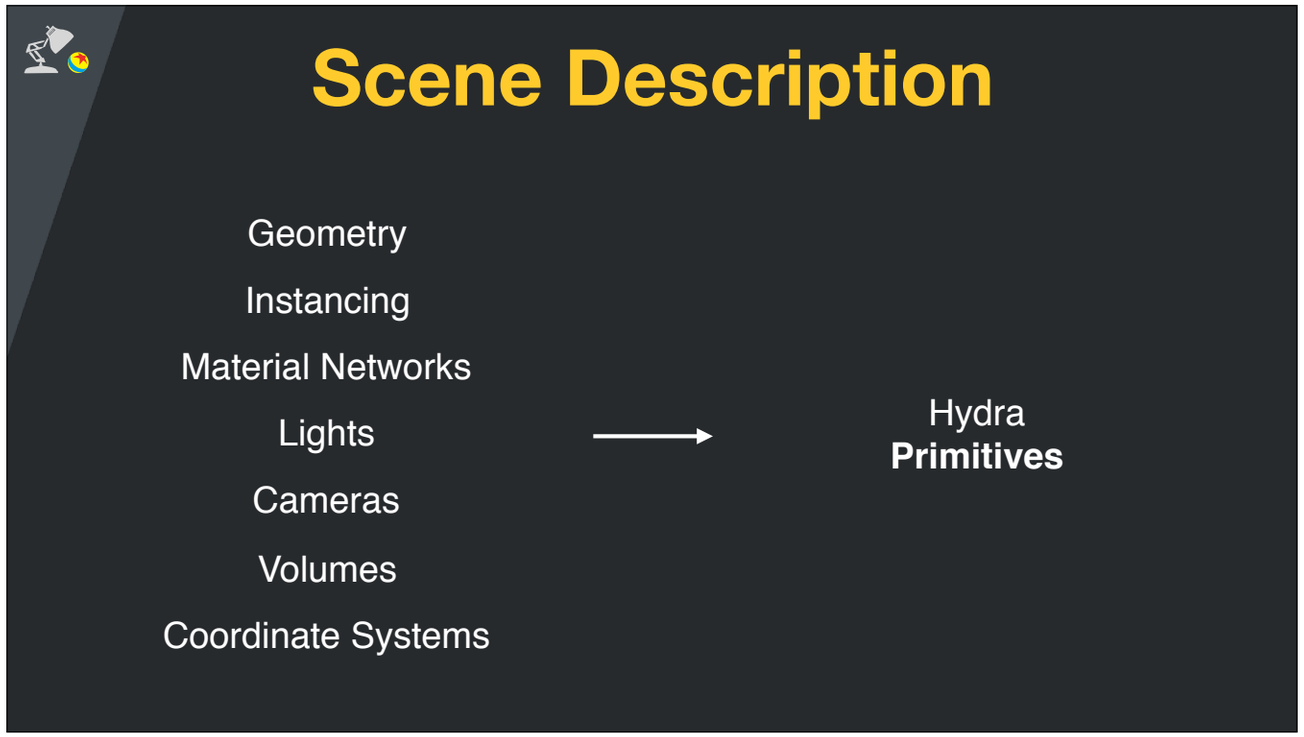
We will be discussing :

We will not be discussing :

Having said that… Please, please, come find us after if you have questions about any of the topics covered or not covered today!

Now let’s talk about Hydra’s Architecture.

And we will start with the Core.



I showed this slide before, and it was an over simplification, so let’s go deeper.

I said that all this scene description becomes Hydra primitives, but… What are these Hydra primitives?

Hydra Primitives

Rprims

Sprims

Bprims

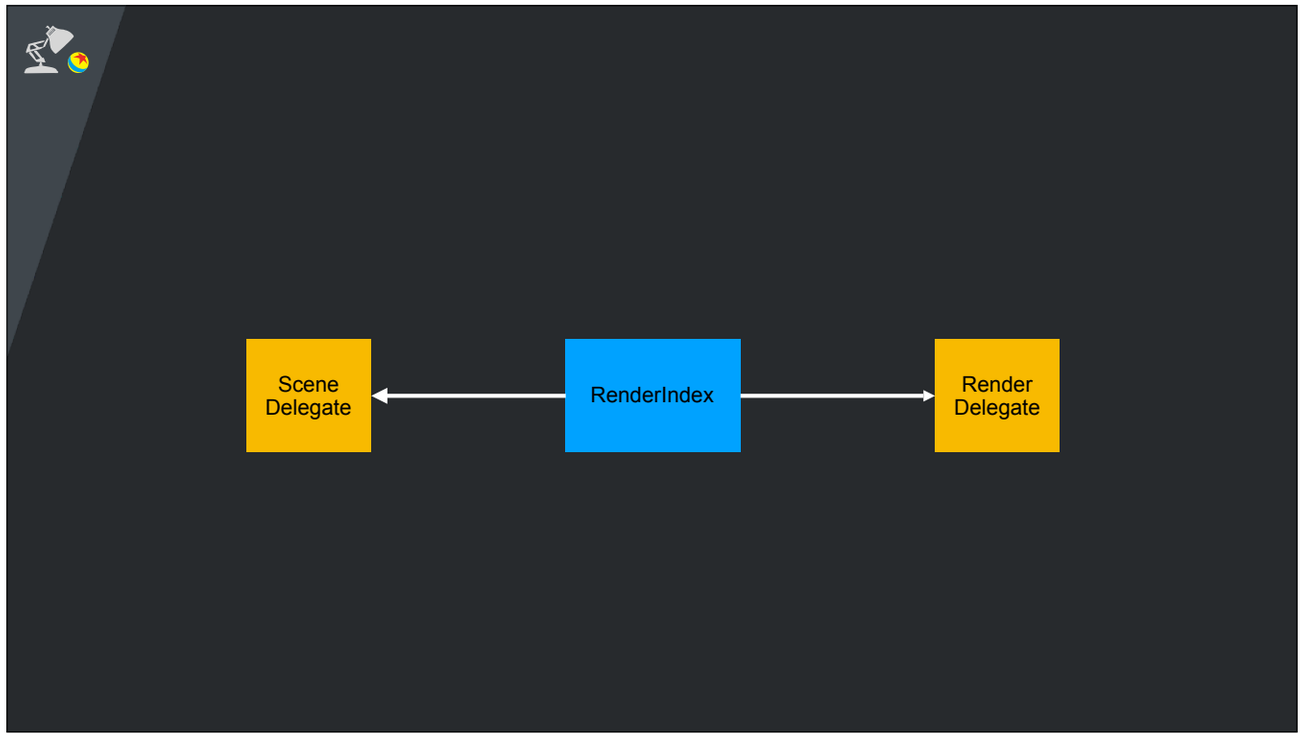
There are three types of primitives :

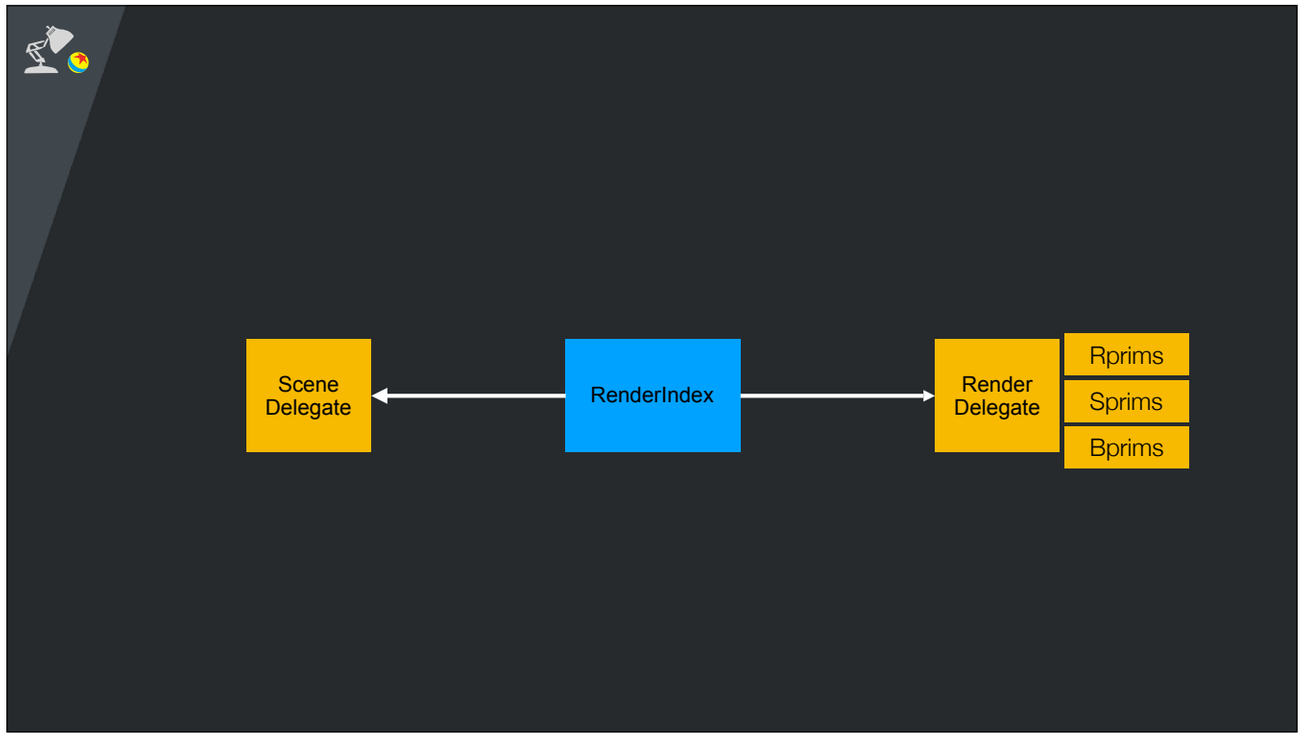
(next slide) — —

- Rprims are useful to represent renderable primitives - R from render - Things like meshes, curves, points…
- Sprims are useful to represent state - S from state - Things like cameras
- Bprims are useful to represent buffers - B from buffer - Things like textures

Hydra Primitives

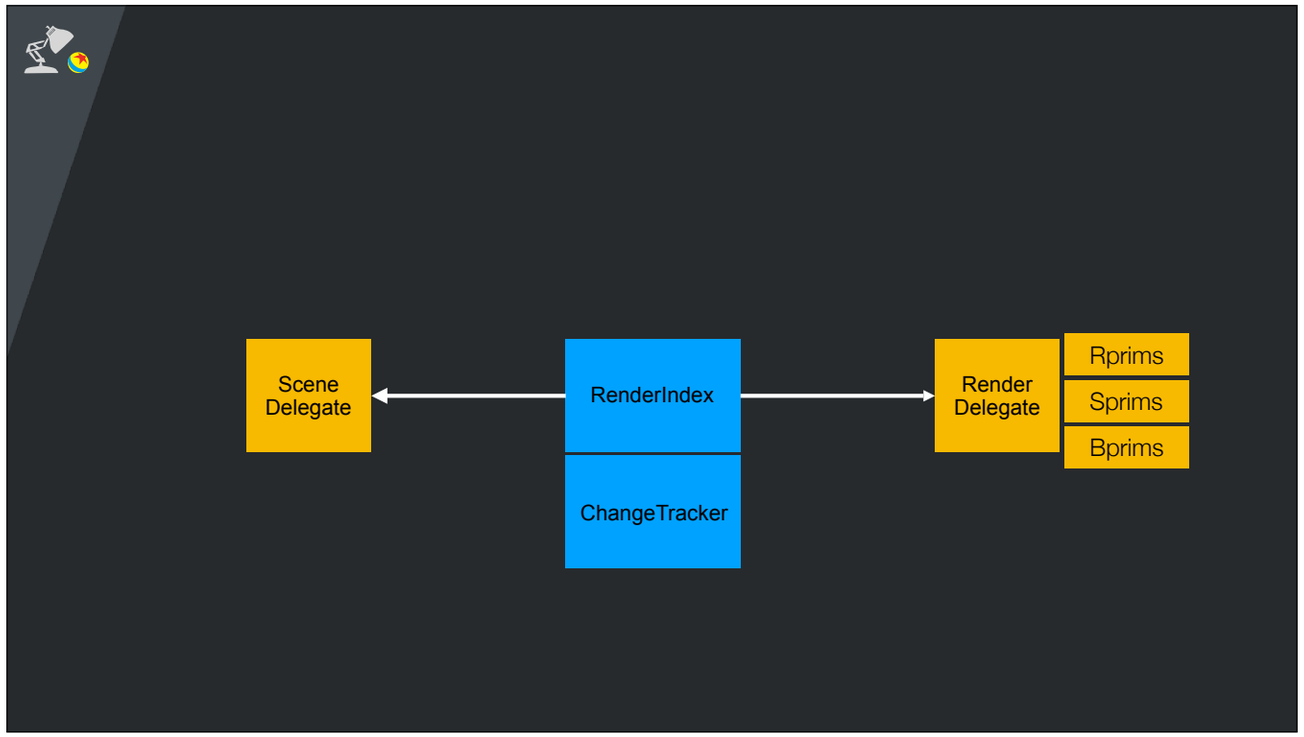
Rprims	Sprims	Bprims
Mesh	Camera	Texture
Basis Curves	Light	Buffer
Points	Material	Field
Volume	Computation	
	Coordinate System	


Scene
Delegate
RenderIndex
Render
Delegate

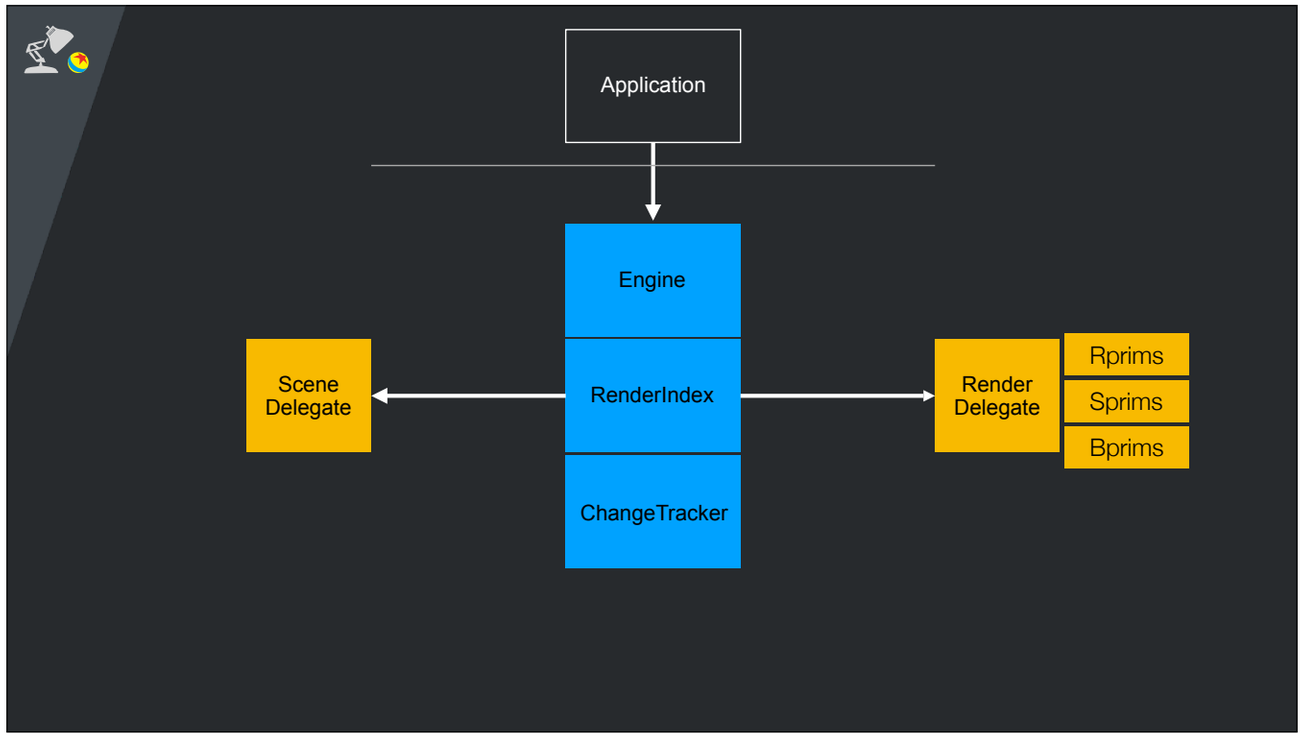


Hydra is a pull system, but not just that, it only pulls data when it is needed.

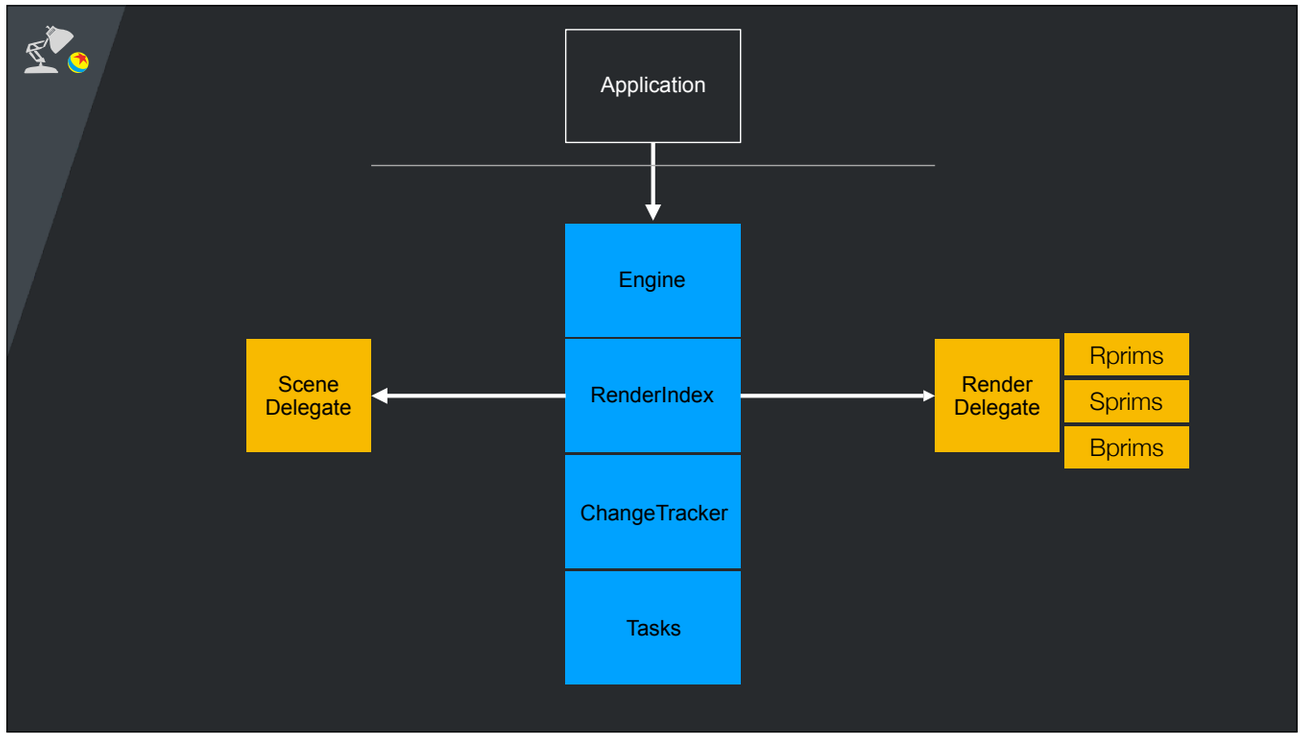
As we said before, the render index is like a list of prims that you can do queries against.



Change tracker allows us to notify the render delegate when its information is not up to date anymore.



Engine is the Hydra ecosystem access point for an application.

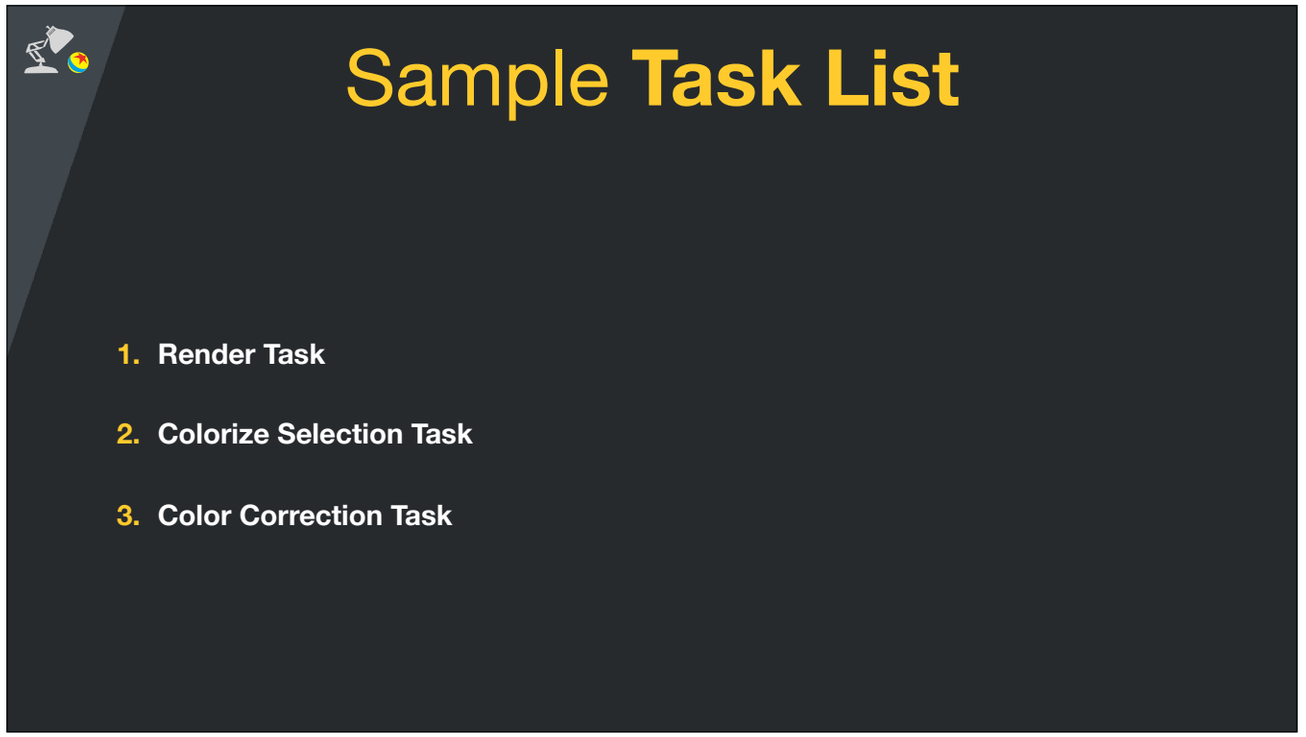


Tasks tell the Hydra engine what to do, for instance rendering, calling to OCIO for color correction, compositing selection highlighting…


Hydra **Execute**

Hydra **Execute**

- Hydra Engine Execute is the main entry point for Hydra
- Requires a list of tasks for Hydra to execute



A classic example of a task is Render but we can do much more!
Let’s take a look at this simple example ：
(read slide)
This is pretty much what we execute with Hydra when you are in Usdview.

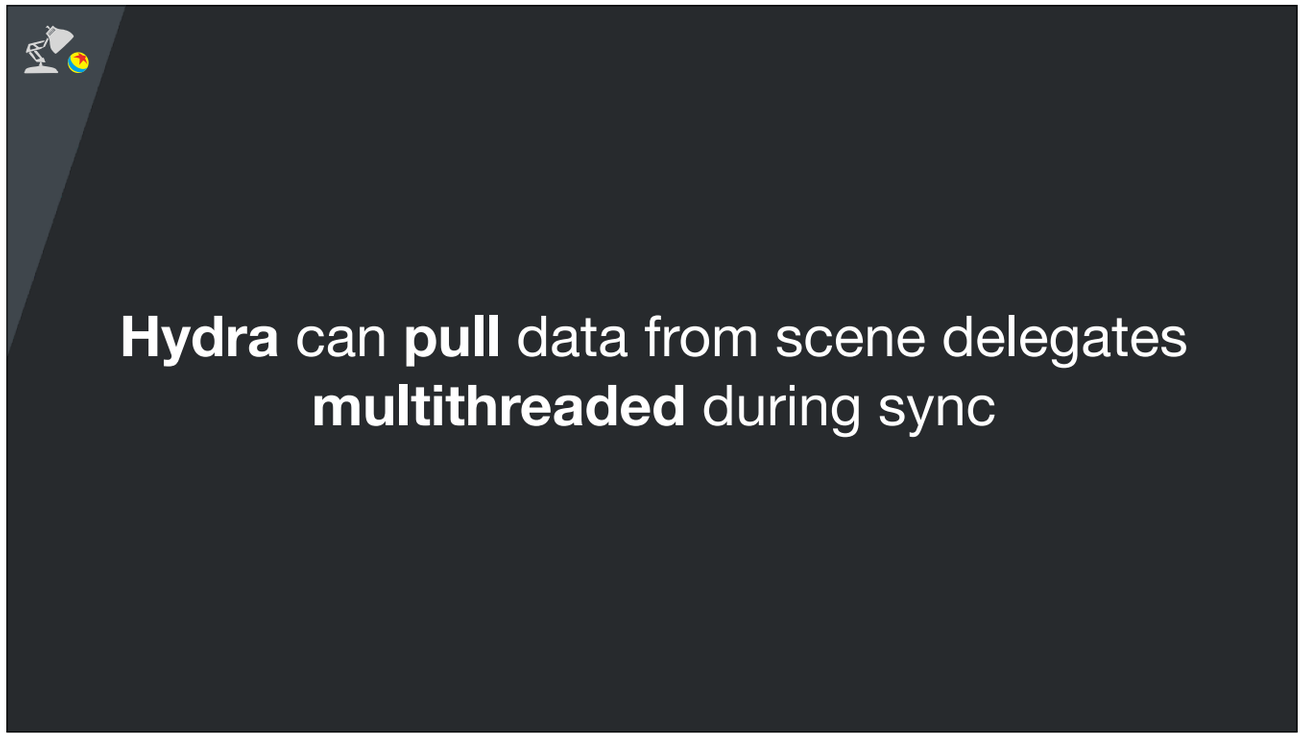
Hydra Engine **Execute**

Phases

1. **Sync RenderIndex**
2. **Prepare all tasks**
3. **Commit resources**
4. **Execute all tasks**

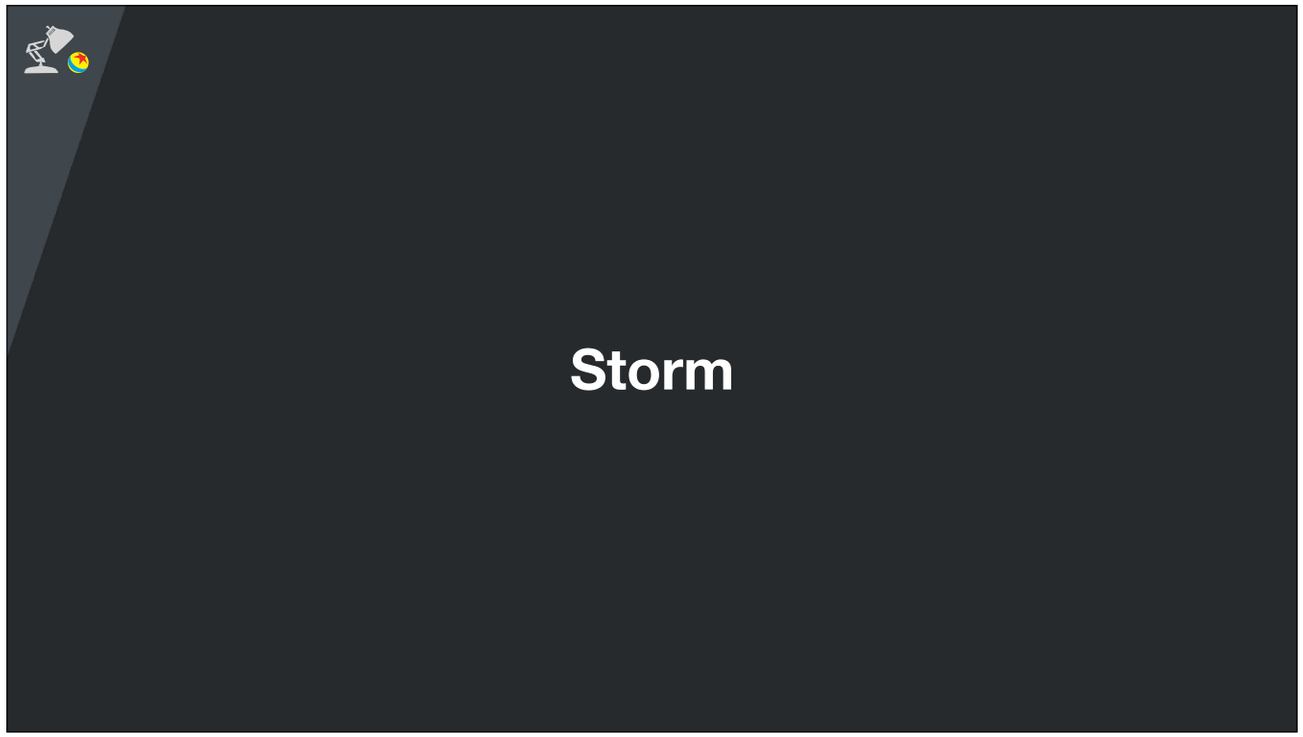
Sync - Pulls data from the scene graph
Prepare - Opportunity to resolve prim dependencies since sync has run for all prims
Commit - Opportunity to submit to the GPU for instance
Execute - Rendering

Performance Remarks



The scene delegate should allow for data to be pulled data in a multithreaded way.

UsdImaging, our USD scene delegate, allows for data to be pulled data in a multithreaded way.



Speaking of multithreading…
Our real-time rasterizer runs on the same thread as the scene delegate, having said that, we heavily use gpu compute prepare buffers in the gpu for graphics consumption.
Our embree sample code uses a new component (which is optional), the render thread.

The render threads allows for separating rendering from the render delegate.

This is specially useful for progressive rendering.

During render delegate initialization, you can initialize the renderThread and pass a callback, you will get call when rendering is needed. The renderThread has a state machine inside.

Embree sample code - It uses a new component (which is optional) that we introduced in 2018, the render thread.

RenderThread Example

```
class MyRenderDelegate : HdRenderDelegate
{
public:
    MyRenderDelegate() {
        _renderThread.SetRenderCallback(std::bind(&MyRenderDelegate::_RenderCallback, this));
        _renderThread.StartThread();
    }

    ~MyRenderDelegate() {
        _renderThread.StopThread();
    }

private:
    void _RenderCallback() {
        // generate pixels.
    }

    HdRenderThread _renderThread;
};
```

Here is an example of render delegate that uses the RenderThread API to separate the actual execution of the renderer from the thread where Hydra and the Render Delegate runs.
As you can see the code is very simple. Create the object, and provide a call to generate pixels. The RenderThread components takes care of state management.

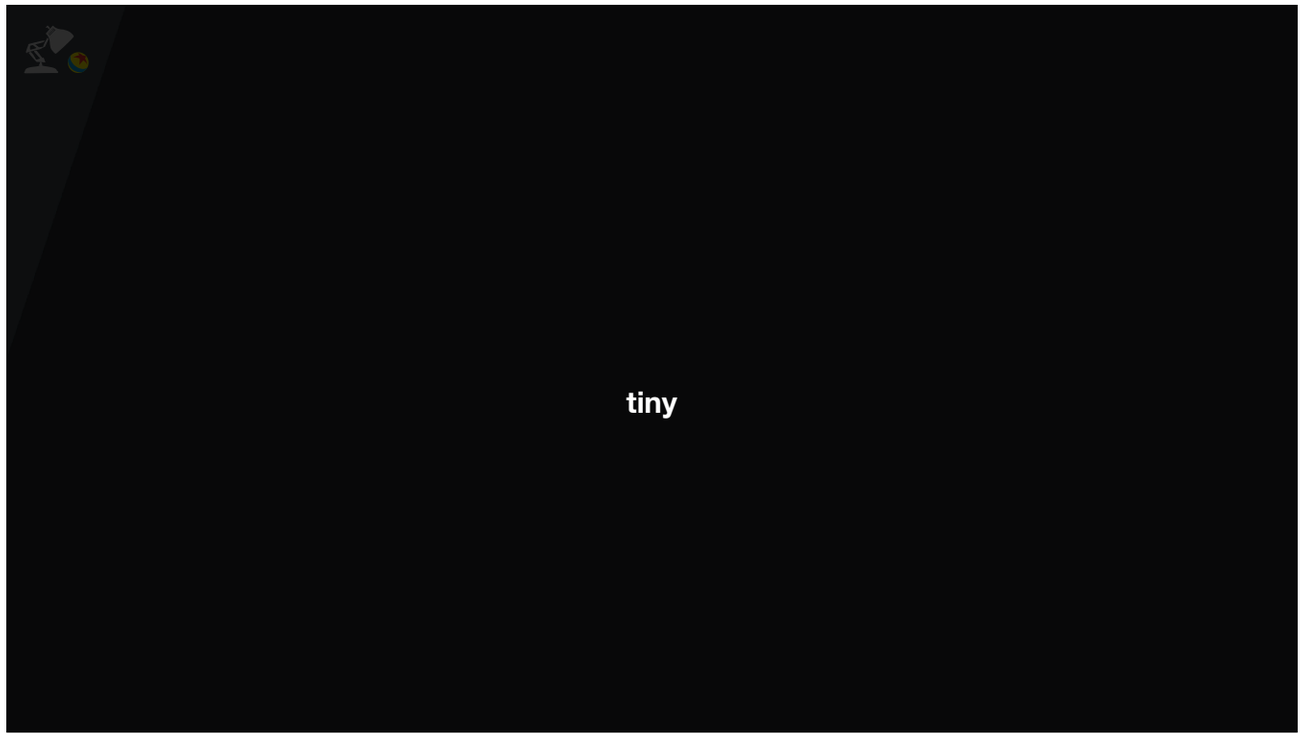
Another important question is… what about memory.

Similar to USD, anywhere Hydra uses a VtArray, we have zero copy behaviors, and you can adapt your data via “foreign data source” if needed.

You do need to remember that if you try to mutate it, you will need to do a copy on write.

A Trip Through Hydra


Introducing

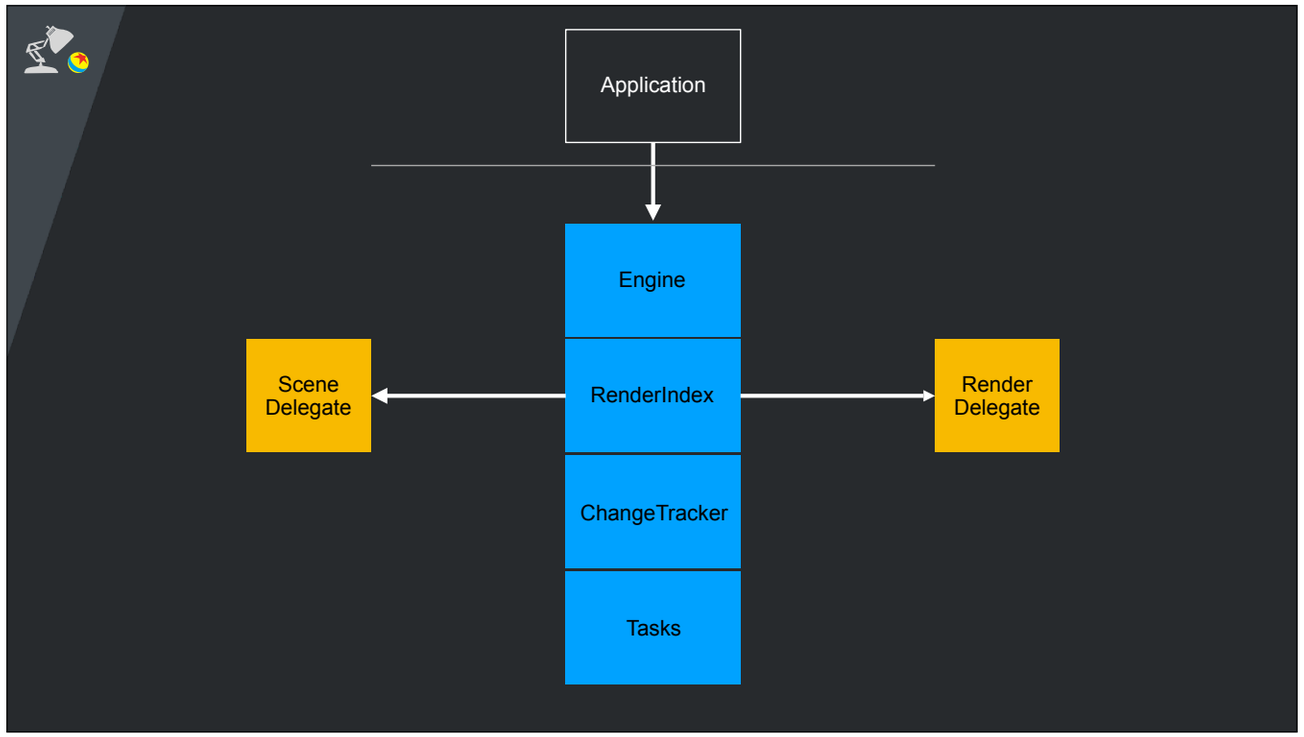

tiny

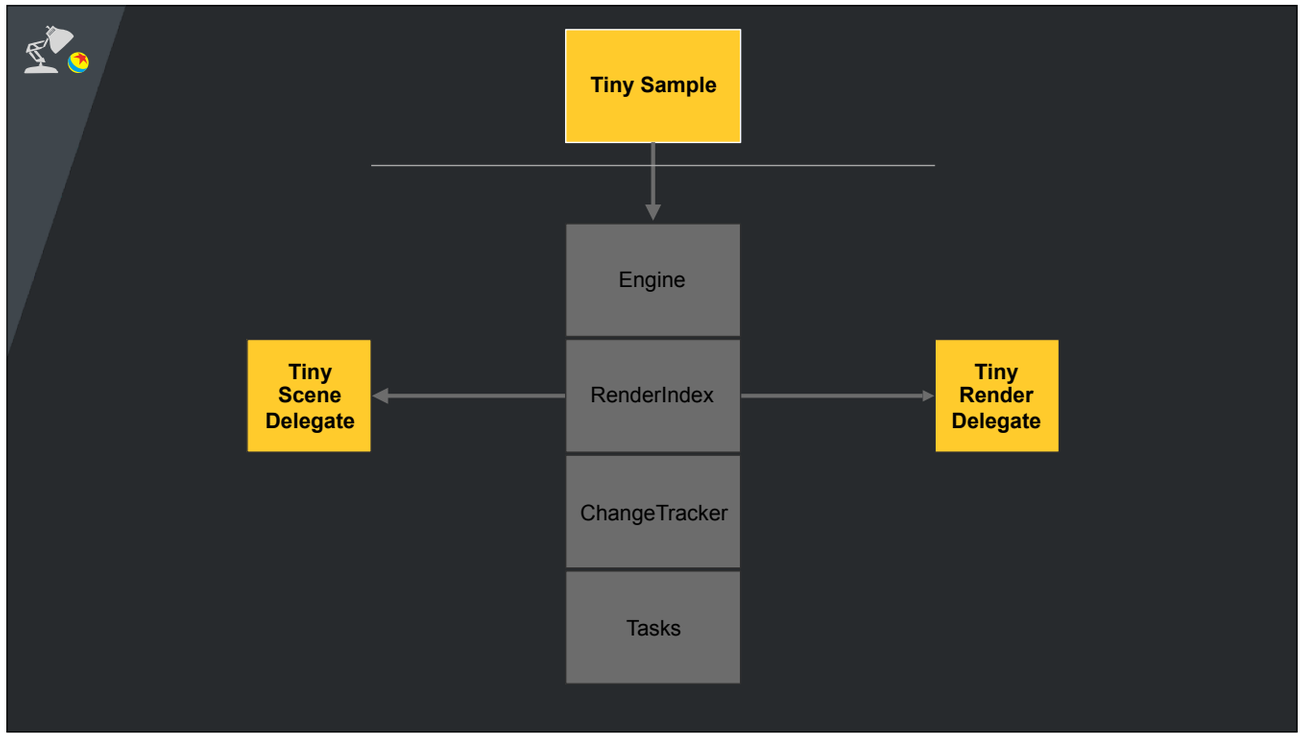
Tiny Sample Code

- A tiny application
  - Tiny scene delegate
  - Tiny render delegate

Tiny Sample Code

- Scene Graph
  - One cube with no time samples
  - One cube with time sampled transforms
- Renderer
  - Output to console when events happen


Application
Engine
Scene Delegate
RenderIndex
Render Delegate
ChangeTracker
Tasks


Tiny Sample
Engine
Tiny Scene Delegate
RenderIndex
Tiny Render Delegate
ChangeTracker
Tasks

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

You can think of collections as a way to determine what you want to render from the render index. It acts almost as a google search query.
The renderpass is the actual call to the renderer to do a render.
The renderpass state sets up the necessary information for a render to happen.
RenderTask is just a user define task that combines this concepts, basically it first sets up the state required for rendering and then, it renders.

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(&renderDelegate);
    TinySceneDelegate sceneDelegate(renderIndex, SdfPath::AbsoluteRootPath());

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDelegate.CreateRenderPass(renderIndex, collection));
    HdRenderPassStateSharedPtr renderPassState(renderDelegate.CreateRenderPassState());
    HdTaskSharedPtr taskRender(new RenderTask(renderPass, renderPassState));
    HdTaskSharedPtr taskColorCorrection(new ColorCorrectionTask());
    HdTaskSharedPtrVector tasks = { taskRender, taskColorCorrection };

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and generate image
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

A Tiny Sample

```
void TinySceneDelegate::Populate()
{
    SdfPath id1("/Cube1");
    _AddCube(id1);
    GetRenderIndex().InsertRprim(HdPrimTypeTokens->mesh, this, id1);

    SdfPath id2("/Cube2");
    _AddCube(id2);
    GetRenderIndex().InsertRprim(HdPrimTypeTokens->mesh, this, id2);
    ...
}
```

SdfPath : In Hydra we use SdfPath to identify prims.

A Tiny Sample

```
void TinySceneDelegate::SetTime(unsigned int time)
{
  SdfPath id("/Cube1");
  GetRenderIndex().GetChangeTracker().MarkRprimDirty(id, HdChangeTracker::DirtyTransform);
  …
}
```

A Tiny Scene Delegate

```
class TinySceneDelegate final: public HdSceneDelegate
{
public:
    // Scene delegate implementation
    HdMeshTopology GetMeshTopology(SdfPath const& id) override;
    GfMatrix4d GetTransform(SdfPath const& id) override;
    VtValue Get(SdfPath const& id, TfToken const& key) override;
    HdPrimvarDescriptorVector GetPrimvarDescriptors(SdfPath const& id,…) override;

    // Scene graph population
    void Populate();

    …
};
```

A Tiny Scene Delegate

```
class TinySceneDelegate final: public HdSceneDelegate
{
public:
    // Scene delegate implementation
    HdMeshTopology GetMeshTopology(SdfPath const& id) override;
    GfMatrix4d GetTransform(SdfPath const& id) override;
    VtValue Get(SdfPath const& id, TfToken const& key) override;
    HdPrimvarDescriptorVector GetPrimvarDescriptors(SdfPath const& id,…) override;

    // Scene graph population
    void Populate();

    …
};
```

A Tiny Scene Delegate

```
class TinySceneDelegate final: public HdSceneDelegate
{
public:
    // Scene delegate implementation
    HdMeshTopology GetMeshTopology(SdfPath const& id) override;
    GfMatrix4d GetTransform(SdfPath const& id) override;
    VtValue Get(SdfPath const& id, TfToken const& key) override;
    HdPrimvarDescriptorVector GetPrimvarDescriptors(SdfPath const& id,…) override;

    // Scene graph population
    void Populate();

    …
};
```

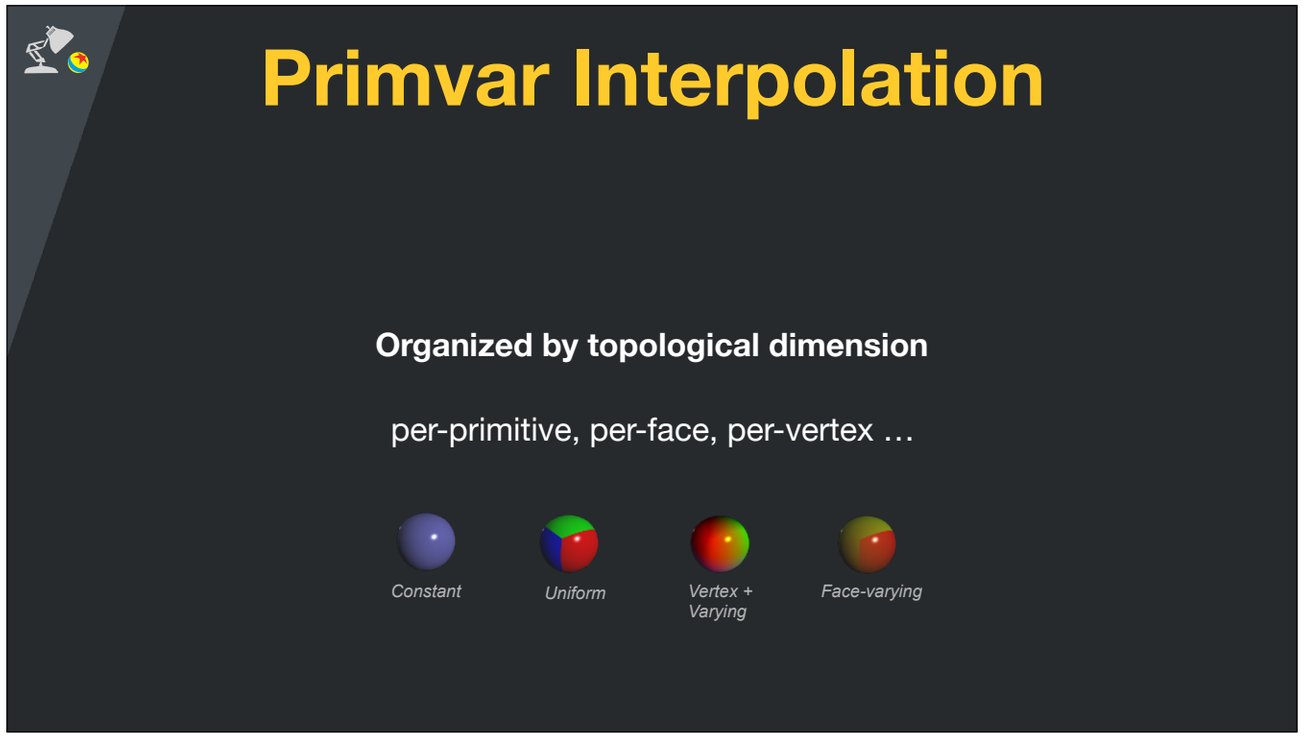
These methods can be called from multithreading code.

What is a Primvar?

Geometric primitive data that is not topology

position, color, uv …

Primvars are simply all of the geometric primitive data that is not topology,
For example: position, color, uv, specular, etc.



Primvar data is organized by its topological dimension,
That is: per-prrimitive, per-face, per-vertex, etc.

RenderMan introduced a specific set of names to describe these aspects and we have adopted them throughout our tools.

Constant - per-mesh or per-curve set
Uniform - per-face or per-curve
Vertex and Varying - per-vertex with basis or linear reconstruction
Face-Varying - for data that might have a different value per face along a shared edge like the seam resulting from a UV unwrap

A Tiny Scene Delegate

```
class TinySceneDelegate final: public HdSceneDelegate
{
public:
    // Scene delegate implementation
    HdMeshTopology GetMeshTopology(SdfPath const& id) override;
    GfMatrix4d GetTransform(SdfPath const& id) override;
    VtValue Get(SdfPath const& id, TfToken const& key) override;
    HdPrimvarDescriptorVector GetPrimvarDescriptors(SdfPath const& id,…) override;

    // Scene graph population
    void Populate();

    …
};
```

A Tiny Render Delegate

```
class TinyRenderDelegate final: public HdRenderDelegate
{
public:
    // Create/Destroy supported types of Hydra primitives (Rprim, Sprim, Bprim)
    HdRprim *CreateRprim(TfToken const& typeId,
        SdfPath const& rprimId, SdfPath const& instancerId) override
    {
        return new TinyRenderDelegate_Mesh(typeId, rprimId, instancerId);
    }

    void DestroyRprim(HdRprim *rPrim) override
    {
        delete rPrim;
    }

    …
};
```

A Tiny Render Delegate

```
class TinyRenderDelegate final: public HdRenderDelegate
{
public:
    // Create/Destroy supported types of Hydra primitives (Rprim, Sprim, Bprim)
    HdRprim *CreateRprim(TfToken const& typeId,
        SdfPath const& rprimId, SdfPath const& instancerId) override
    {
        return new TinyRenderDelegate_Mesh(typeId, rprimId, instancerId);
    }

    void DestroyRprim(HdRprim *rPrim) override
    {
        delete rPrim;
    }

    …
};
```

A Tiny Render Delegate

```
class TinyRenderDelegate final: public HdRenderDelegate
{
public:
    // Create/Destroy supported types of Hydra primitives (Rprim, Sprim, Bprim)
    HdRprim *CreateRprim(TfToken const& typeId,
        SdfPath const& rprimId, SdfPath const& instancerId) override
    {
        return new TinyRenderDelegate_Mesh(typeId, rprimId, instancerId);
    }

    void DestroyRprim(HdRprim *rPrim) override
    {
        delete rPrim;
    }

    …
};
```

A Tiny Render Delegate

```
class TinyRenderDelegate_Mesh final: public HdMesh
{
public:
    void Sync(HdSceneDelegate *delegate, HdDirtyBits *dirtyBits …) override
    {
        SdfPath const& id = GetId();

        if (HdChangeTracker::IsTransformDirty(*dirtyBits, id)) {
            delegate->GetTransform(id);
            cout << "Pulling new transform -> " << id << endl;
        }

        *dirtyBits &= ~HdChangeTracker::AllSceneDirtyBits;
    }

    HdDirtyBits GetInitialDirtyBitsMask() const override
    {
        return HdChangeTracker::AllDirty;
    }

    …
};
```

Sync happens multithreaded for this prim, so these are the calls to the scene delegate that happen multithreaded.

A Tiny Render Delegate

```
class TinyRenderDelegate_Mesh final: public HdMesh
{
public:
    void Sync(HdSceneDelegate *delegate, HdDirtyBits *dirtyBits …) override
    {
        SdfPath const& id = GetId();

        if (HdChangeTracker::IsTransformDirty(*dirtyBits, id)) {
            delegate->GetTransform(id);
            cout << "Pulling new transform -> " << id << endl;
        }

        *dirtyBits &= ~HdChangeTracker::AllSceneDirtyBits;
    }

    HdDirtyBits GetInitialDirtyBitsMask() const override
    {
        return HdChangeTracker::AllDirty;
    }

    …
};
```

Pull data from scene graph **only** when it is needed.

A Tiny Task

```
class ColorCorrectionTask final : public HdTask
{
public:
    void Execute(HdTaskContext* ctx) override
    {
        std::cout << “(2) Color correcting image" << std::endl;
    }

    …
};
```

As we said before, in order to render, you need tasks!
So, we will create a simple task, in this case it is the Color correction task which just prints to console, but it could be more advance.

Task can be generic and work cross renderer, like most of the ones we use in TaskController, but they can even do special behaviors if you are ok to have those tasks only work with one renderer.

A Tiny Task

```
class ColorCorrectionTask final : public HdTask
{
public:
    void Execute(HdTaskContext* ctx) override
    {
        std::cout << “(2) Color correcting image" << std::endl;
    }

    …
};
```

A Tiny Task

```
class ColorCorrectionTask final : public HdTask
{
public:
    void Execute(HdTaskContext* ctx) override
    {
        std::cout << “(2) Color correcting image" << std::endl;
    }

    …
};
```

A Tiny Sample

```cpp
int main()
{
    // Hydra initialization
    HdEngine engine;
    TinyRenderDelegate renderDelegate;
    HdRenderIndex *renderIndex = HdRenderIndex::New(
    TinySceneDelegate sceneDelegate(renderInde

    // Create your task graph
    HdRprimCollection collection(…);
    HdRenderPassSharedPtr renderPass(renderDel
    HdRenderPassStateSharedPtr renderPassState
    HdTaskSharedPtr taskRender(new RenderTask(
    HdTaskSharedPtr taskColorCorrection(new C
    HdTaskSharedPtrVector tasks = { taskRende

    // Populate scene graph and generate image
    sceneDelegate->Populate();
    engine.Execute(renderIndex, &tasks);

    // Change time causes invalidations, and
    sceneDelegate->SetTime(1);
    engine.Execute(renderIndex, &tasks);

    return EXIT_SUCCESS;
}
```

```
> tinySample

Hydra engine execute
    Sync multithreaded
        Pulling new transform -> /Cube1
        Pulling new transform -> /Cube2

    Executing tasks
        (1) Generating image
        (2) Color correcting image

Hydra engine execute
    Sync multithreaded
        Pulling new transform -> /Cube1

    Executing tasks
        (1) Generating image
        (2) Color correcting image
```

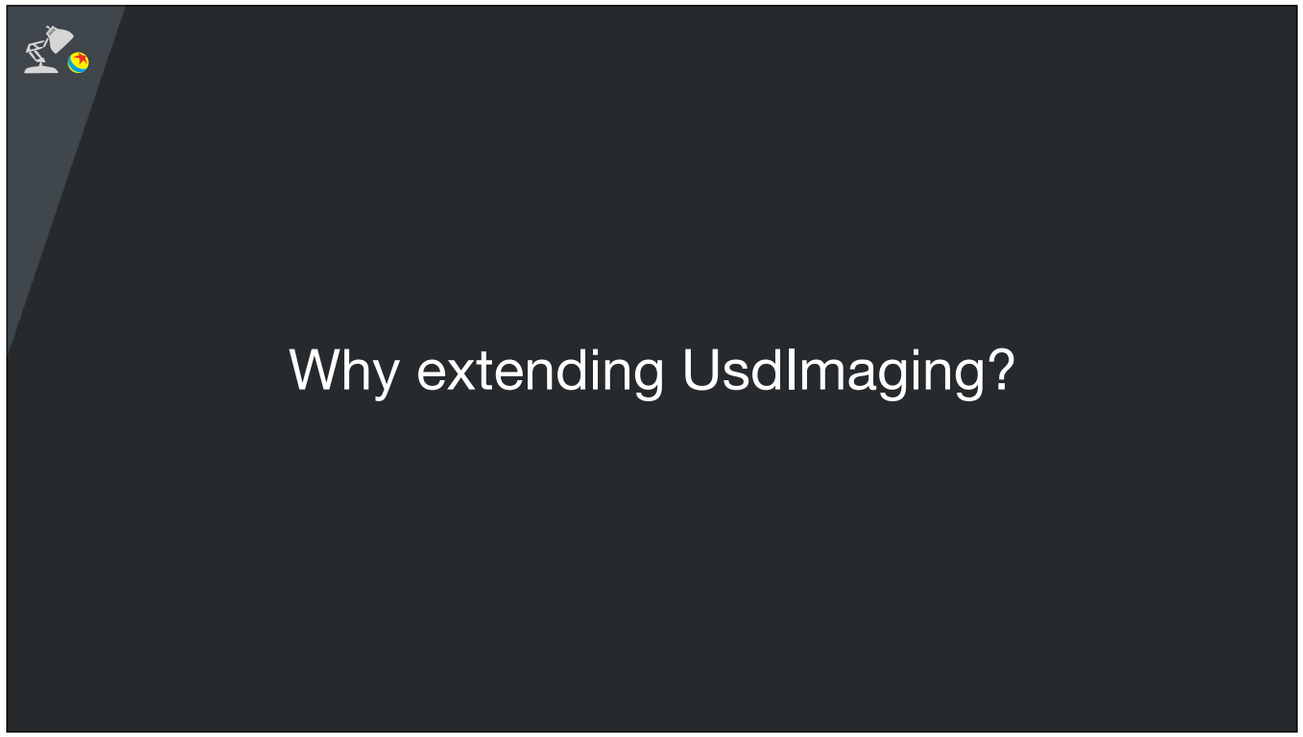
Integration Strategies

Customization & Integration

- Adding your Scene Delegate
- Adding your Render Delegate
- **Extending the USD Scene Delegate**
- **Integrating Hydra into your Application**

Extending the USD Scene Delegate

Even though we mentioned we wouldn’t go too much into detail on how our USD scene delegate works in detail, we do need to discuss a bit of the high level in order to understand when we need to extend it.



Imagine a situation in which your studio requires a very specific prim type that is not currently covered by UsdImaging.

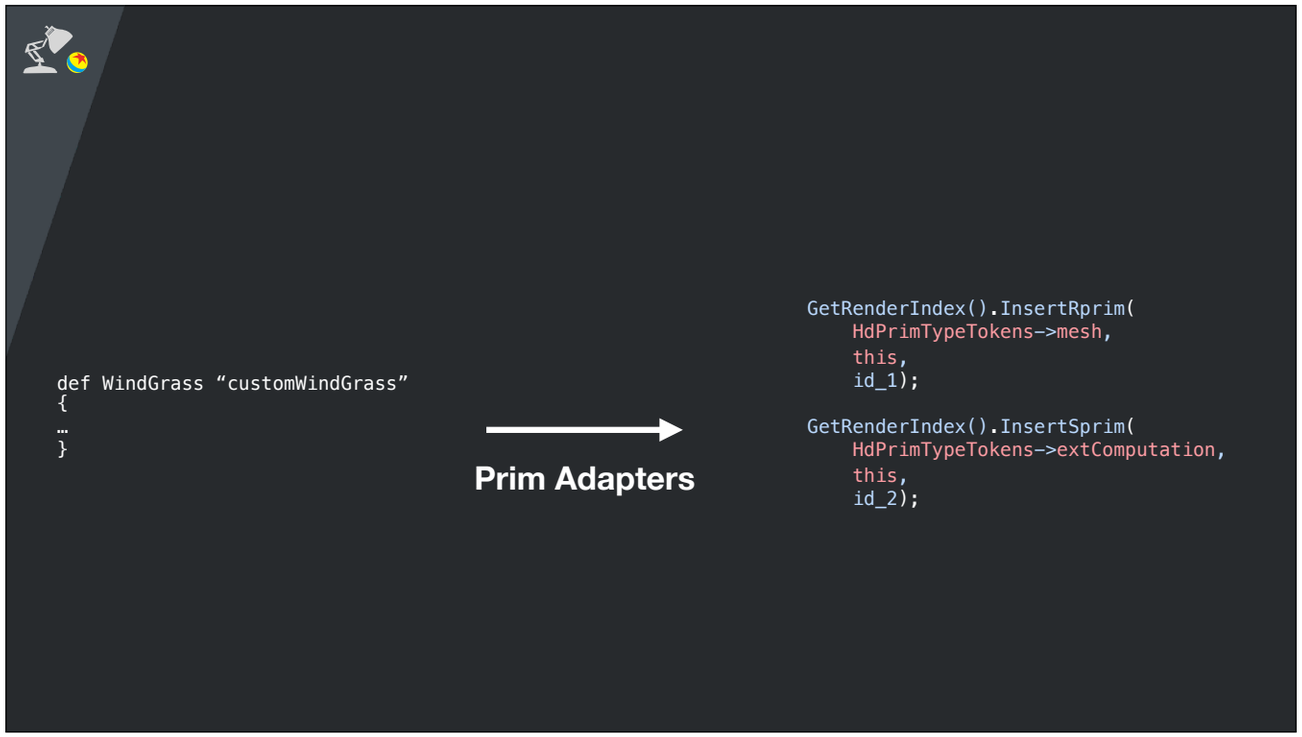
For instance, a very special type curves you have just invented that does not exist in UsdImaging, but that it can be transformed into a Hydra Prim to carry it to the renderer.

Internally, we have use this mechanism for TetMeshes for instance, they are translated to regular Hydra meshes.

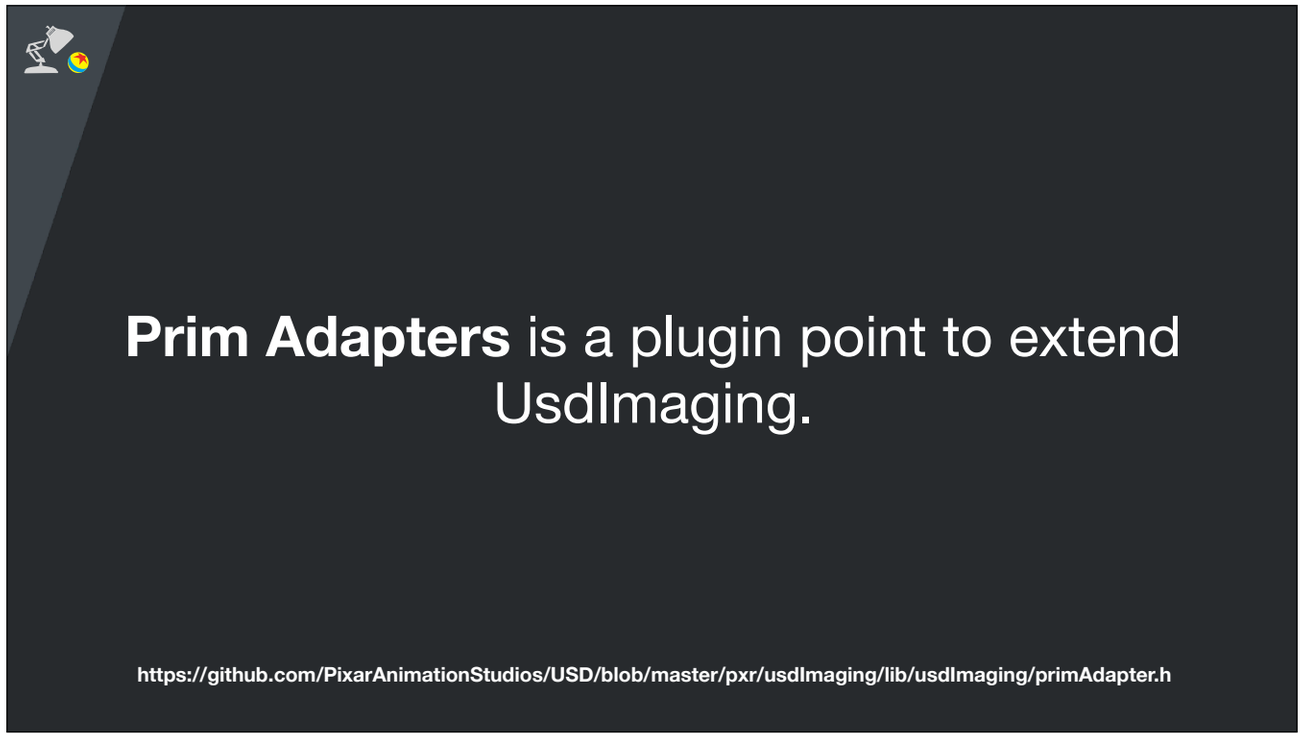
```
def Sphere “sphere”
{
…
}
```

Prim Adapters

```
GetRenderIndex().InsertRprim(
    HdPrimTypeTokens->mesh,
    this,
    id);
```



You are adding additional transformations from USD Scene Description to Hydra prims.



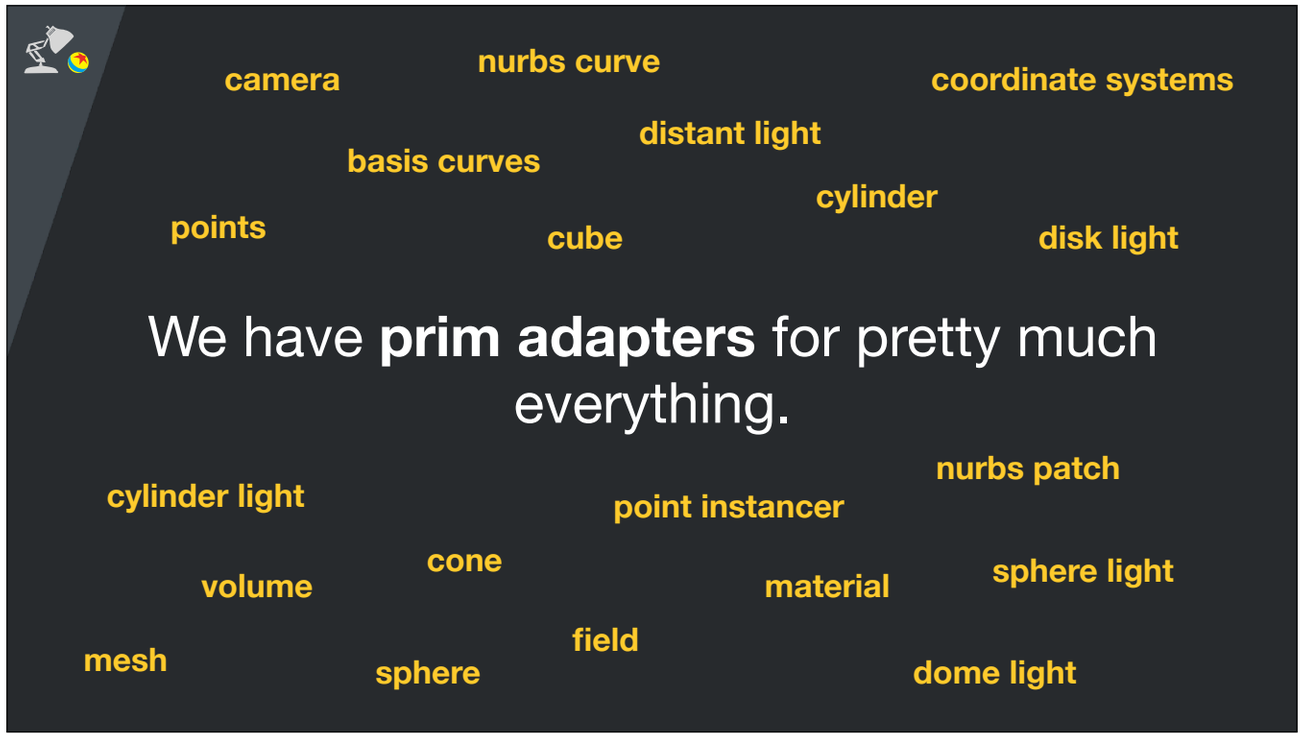
Prim adapters allow for associating a type to code that can populate data for Hydra to interpret.

One simple way to do this would be to just walk the hierarchy in Populate() and then have a big statement that Inserts the Rprims/Sprims into Hydra.

This idea is quite limiting and instead we went a different route in which we delegate the Hydra prim population to prim adapters, which provide a lot of flexibility.

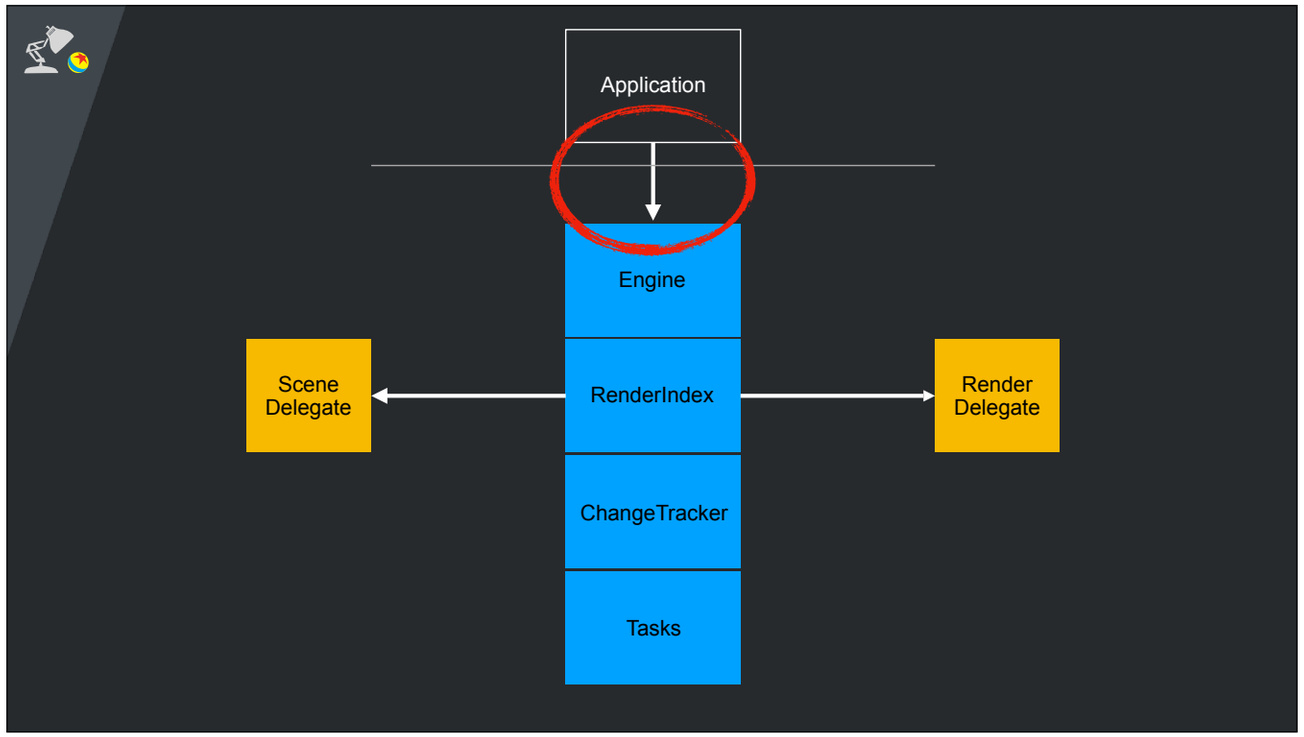
Prim Adapter

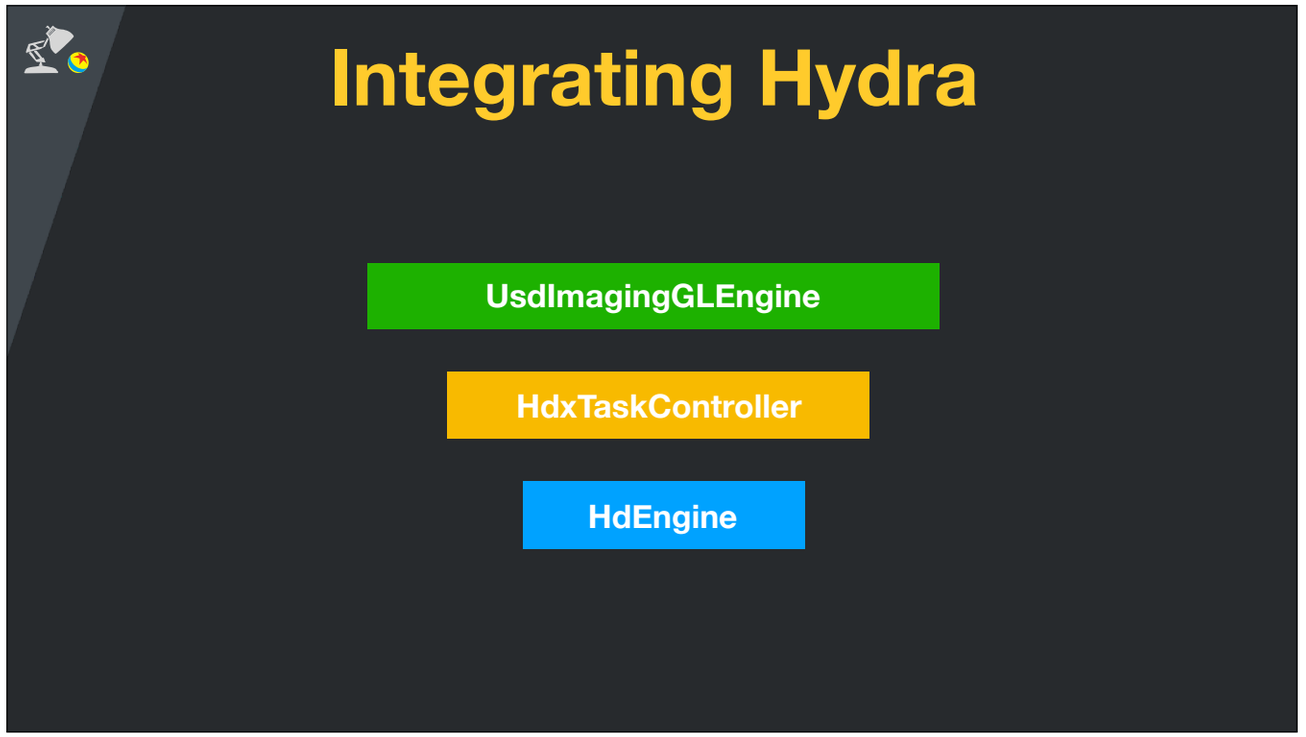
```
class UsdImagingWindGrass : public UsdImagingPrimAdapter
{
public:
  void TrackVariability(UsdPrim const& prim,
                        SdfPath const& cachePath,
                        HdDirtyBits* timeVaryingBits,
                        UsdImagingInstancerContext const* instancerContext = nullptr) const override;

   void UpdateForTime(UsdPrim const& prim,
                      SdfPath const& cachePath,
                      UsdTimeCode time,
                      HdDirtyBits requestedBits,
                      UsdImagingInstancerContext const* instancerContext = nullptr) const override;

   HdDirtyBits ProcessPropertyChange(UsdPrim const& prim,
                                     SdfPath const& cachePath,
                                     TfToken const& propertyName) override;

   void MarkDirty(UsdPrim const& prim,
                  SdfPath const& cachePath,
                  HdDirtyBits dirty,
                  UsdImagingIndexProxy* index) override;
…
```



Matt will be talking about more use cases in production for imaging prim adapters during the next section.


Integrating Hydra
into your Application


Application
Engine
Scene Delegate
RenderIndex
Render Delegate
ChangeTracker
Tasks



Hydra can be embedded to your application in multiple ways :

- For some applications you might find it useful to directly instance HdEngine and be able to configure your scene delegates and your render delegate, this is a great option if you don't want to carry any OpenGL dependencies.
- A different option is to use HdxTaskController which provides a render graph that you can give to HdEngine and it will do things like rendering, colorization, picking...
- Yet another option (more higher level) is to use UsdImagingGLEngine which uses both HdxTaskController and HdEngine, and it facilitates loading a usd stage.

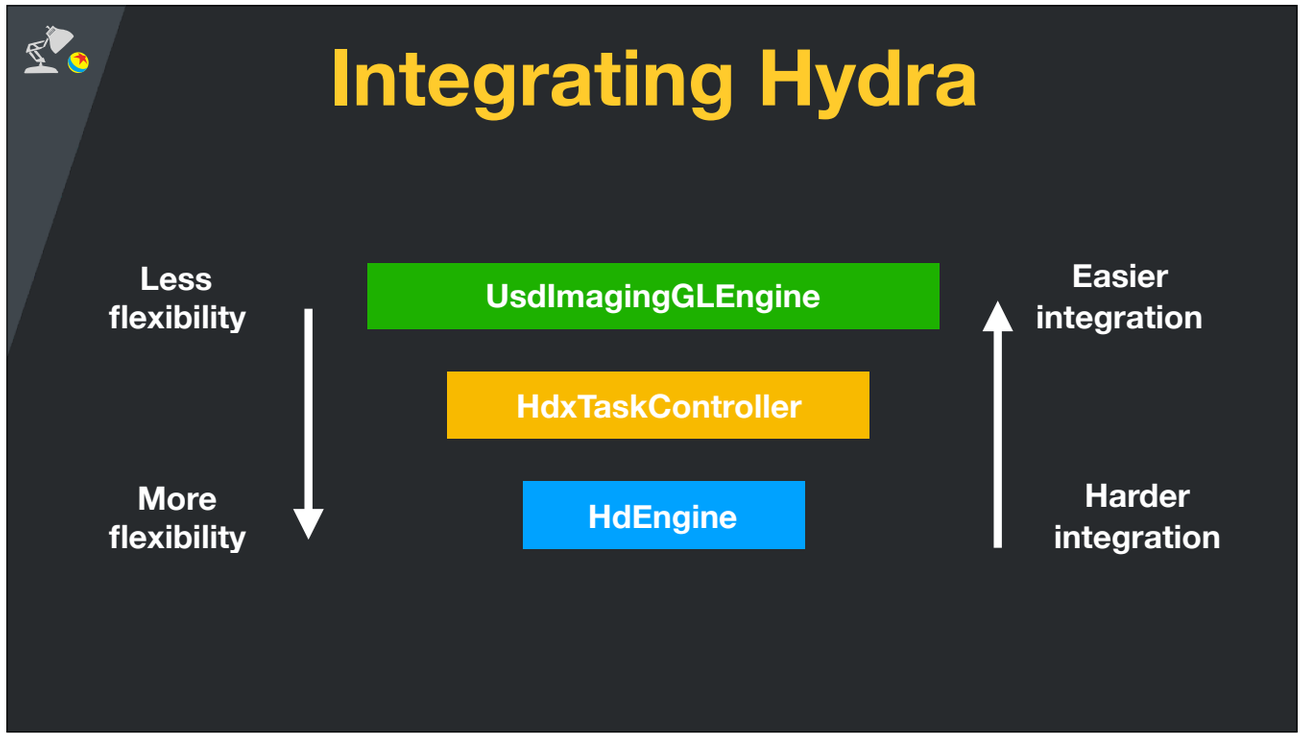
Integrating Hydra

- **UsdImagingEngine**
  Uses HdEngine and HdxTaskController, and it also uses a Usd delegate to provide easy rendering of usd stages

- **HdxTaskController**
  Provides a task graph with rendering, picking and more, it can be used with HdEngine

- **HdEngine**
  Low level Hydra integration

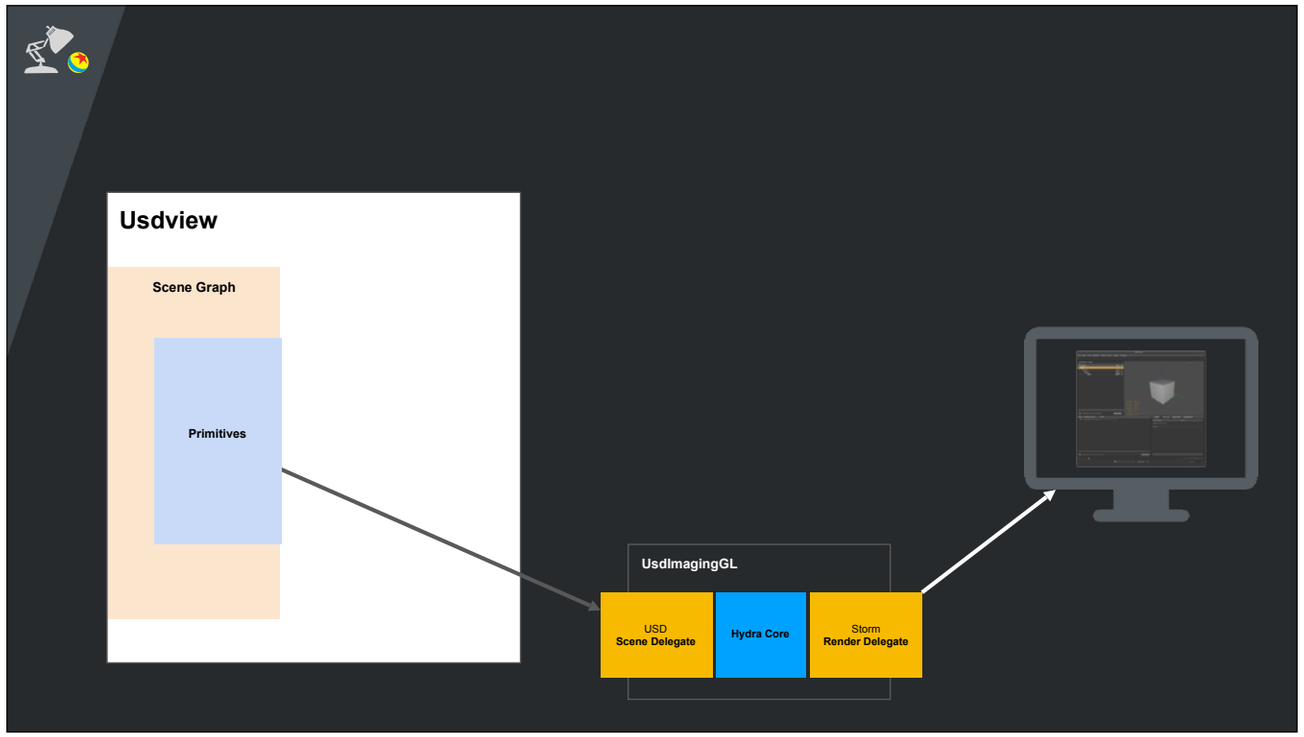
Usdview uses UsdImagingGLEngine
Katana uses taskController
Maya uses its own delegates
Presto its own delegates



Let’s go back to the previous diagram.

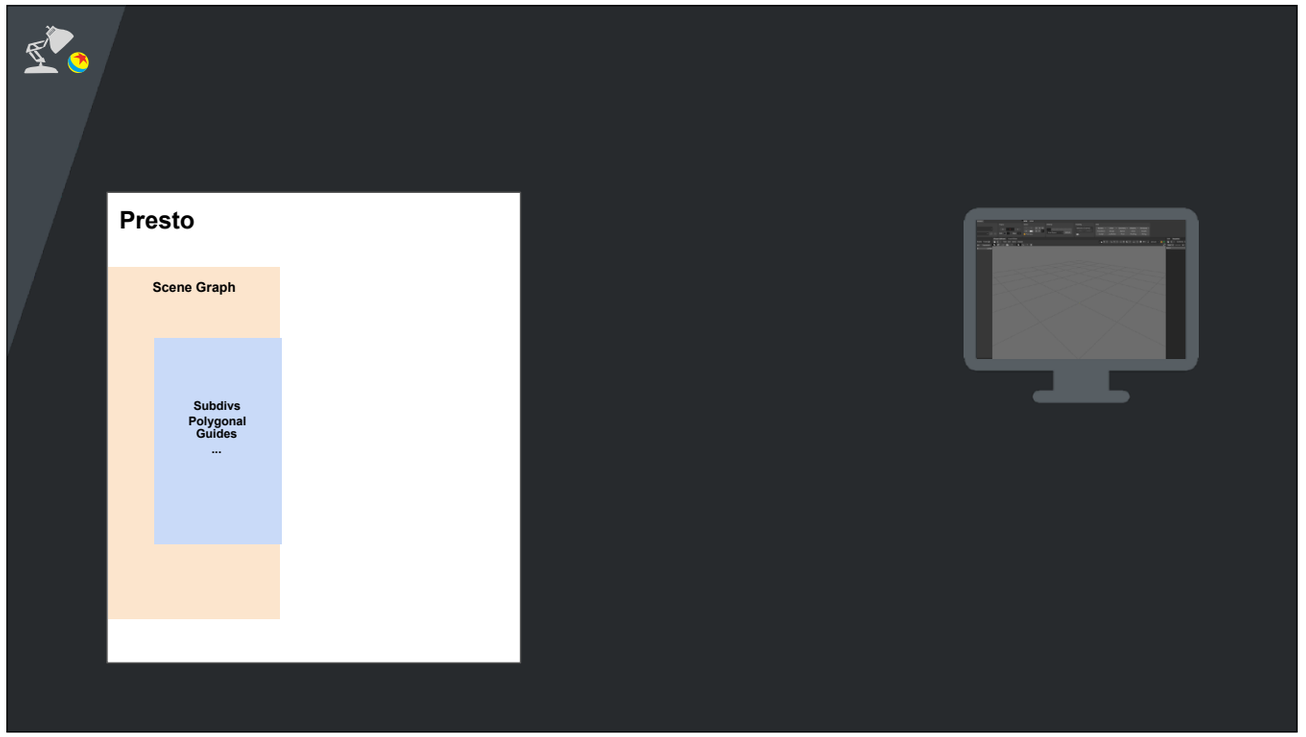
In terms of control, the higher level you go the less low level control you will have.

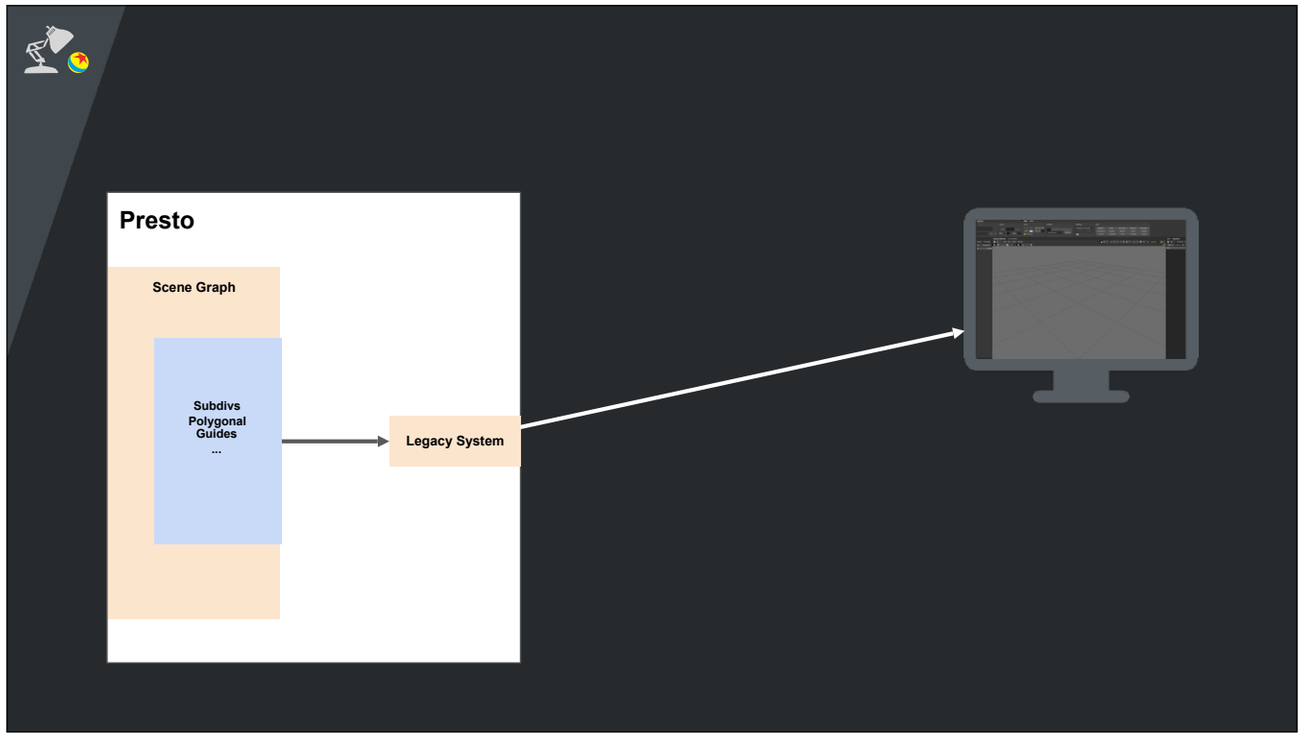
Usdview


Usdview
Scene Graph
Primitives
UsdImagingGL
USD
Scene Delegate
Hydra Core
Storm
Render Delegate

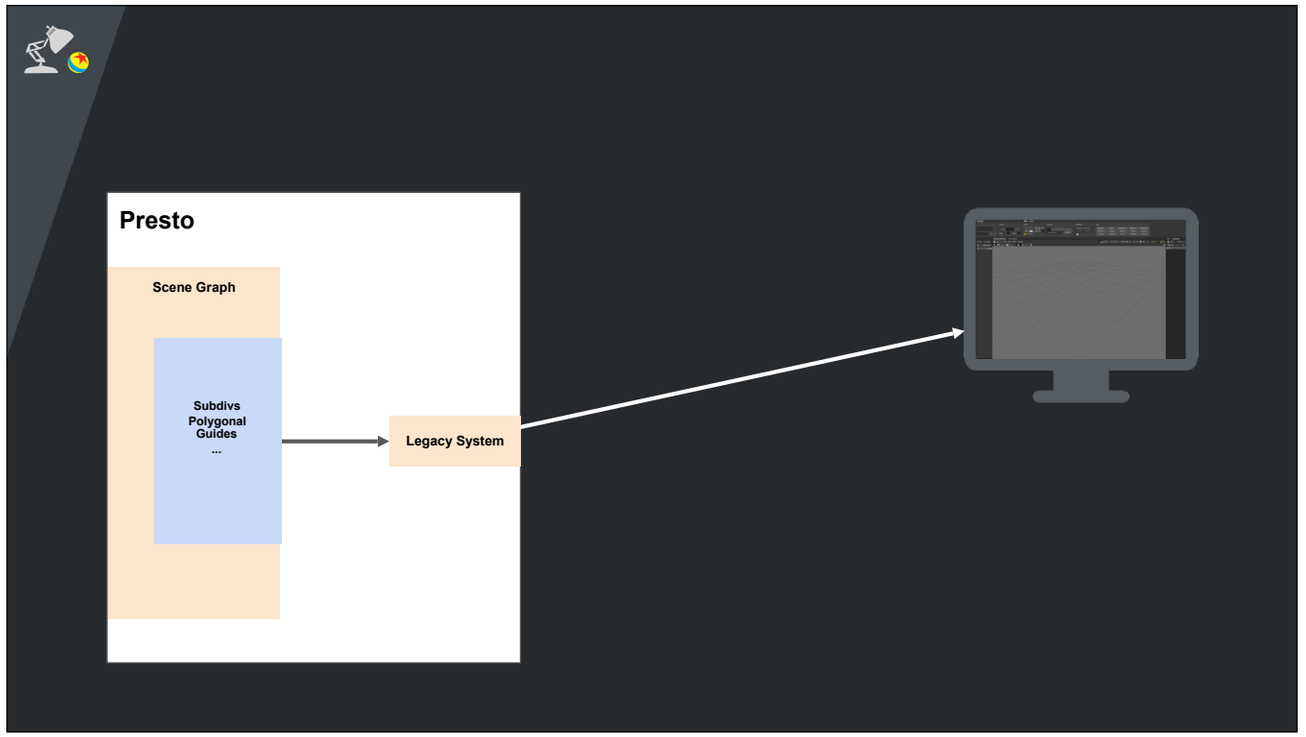
This can also apply to other software like Maya.


Presto
Legacy Rendering System

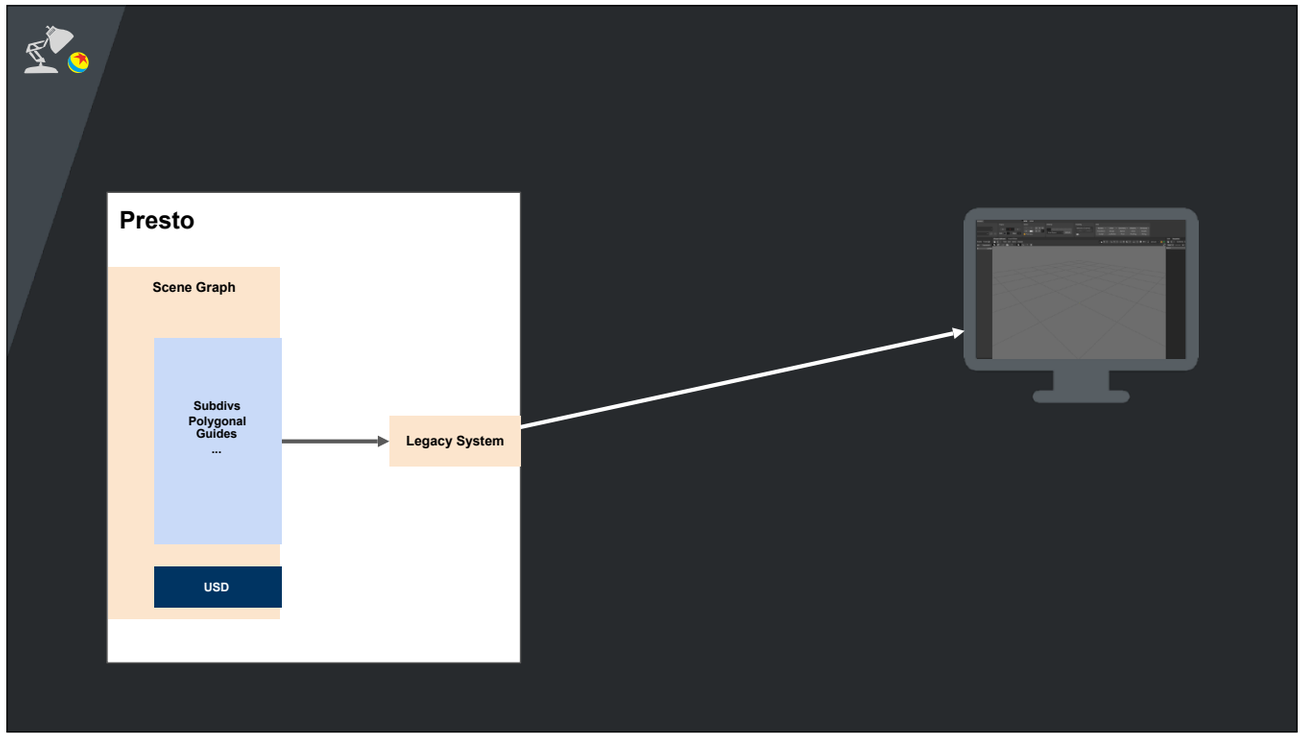

Presto
Scene Graph
Subdivs
Polygonal
Guides
...

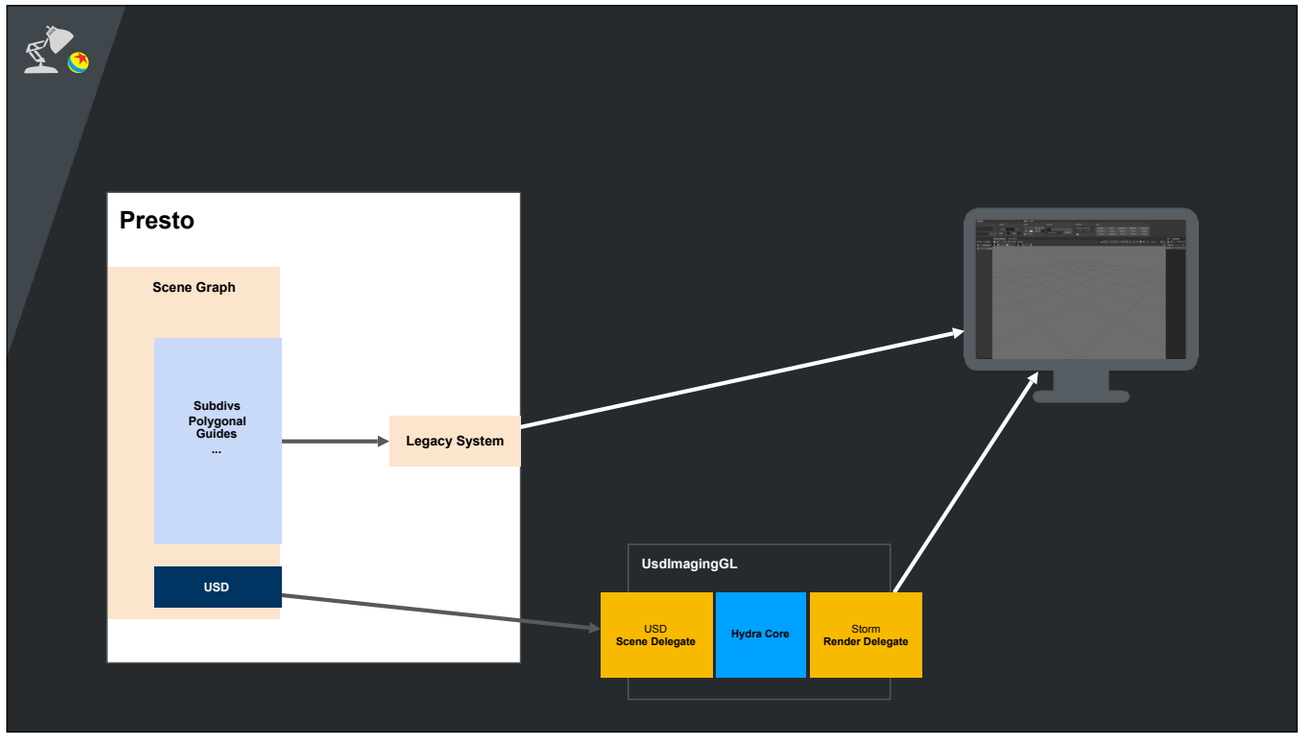

Presto
Scene Graph
Subdivs
Polygonal
Guides
...
Legacy System


Presto
Basic Hydra Integration

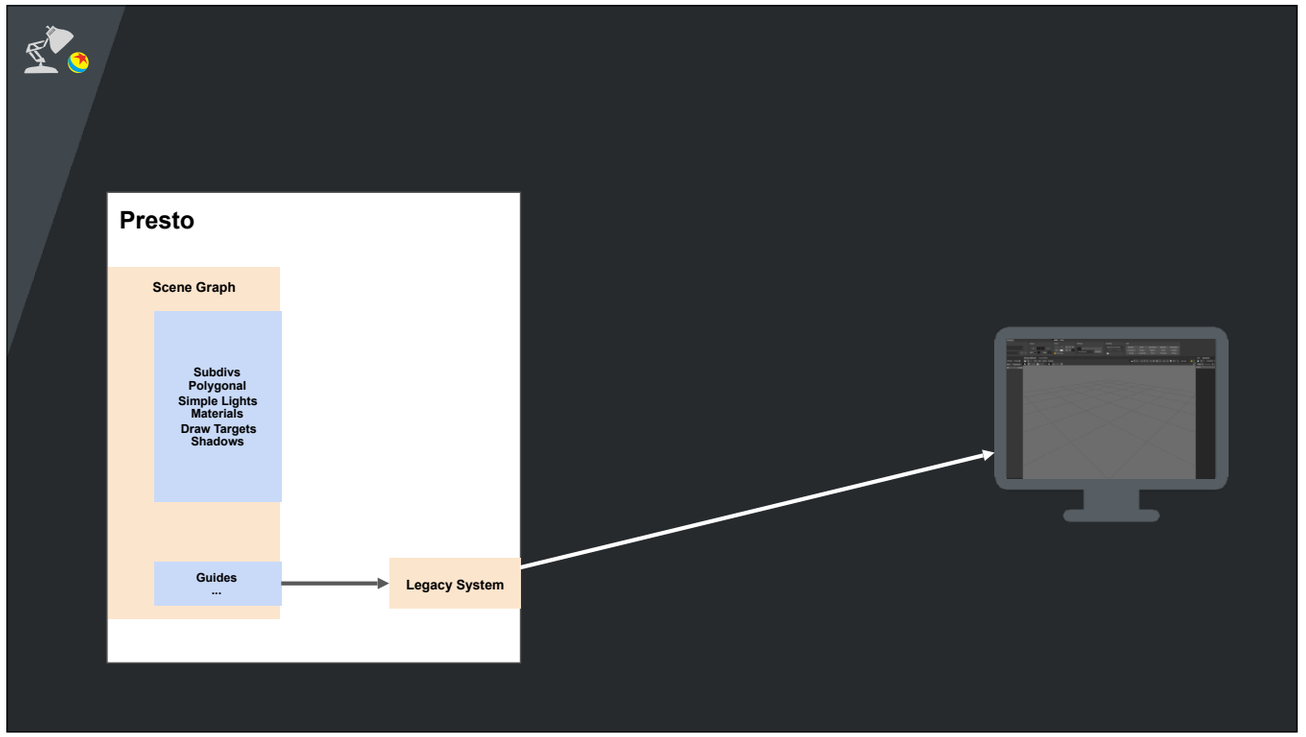


You don’t have to integrate Hydra all over your application, and we didn’t either, you can start small, let’s say integrating UsdImagingGL.

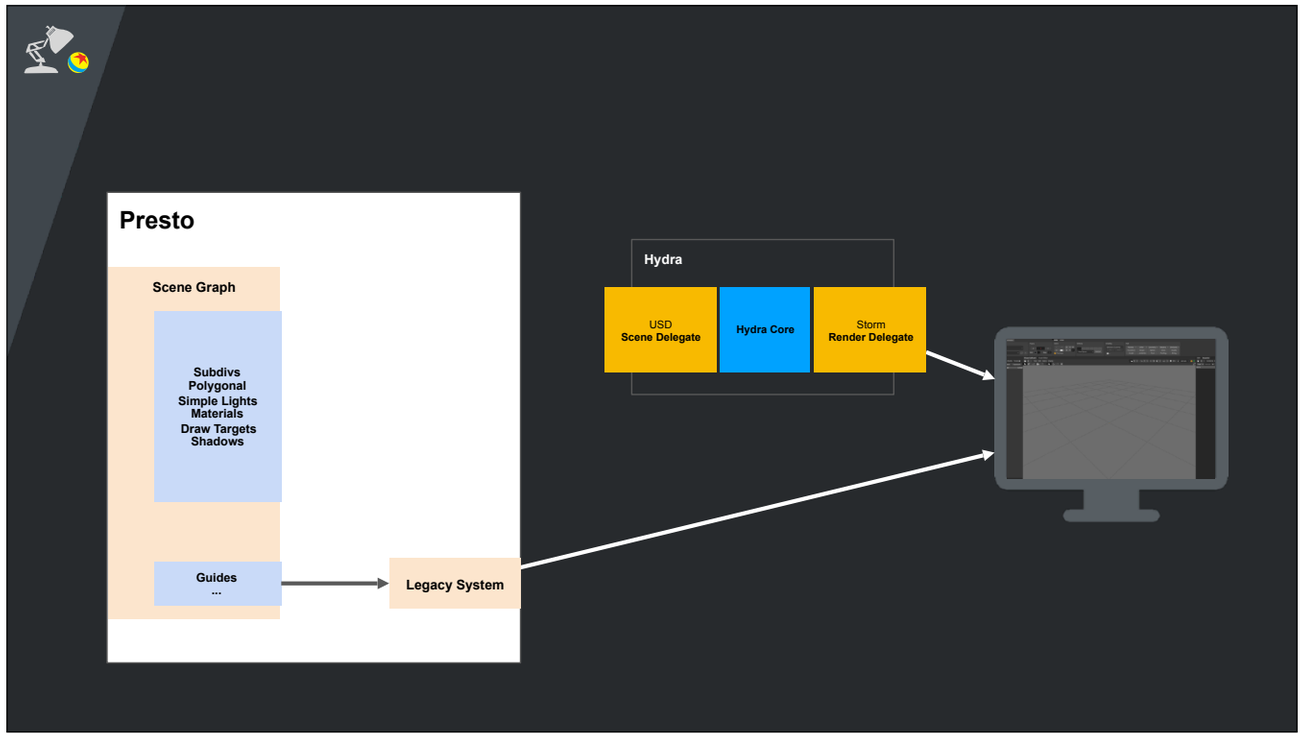

Presto
Scene Graph
Subdivs
Polygonal
Guides
...
USD
Legacy System

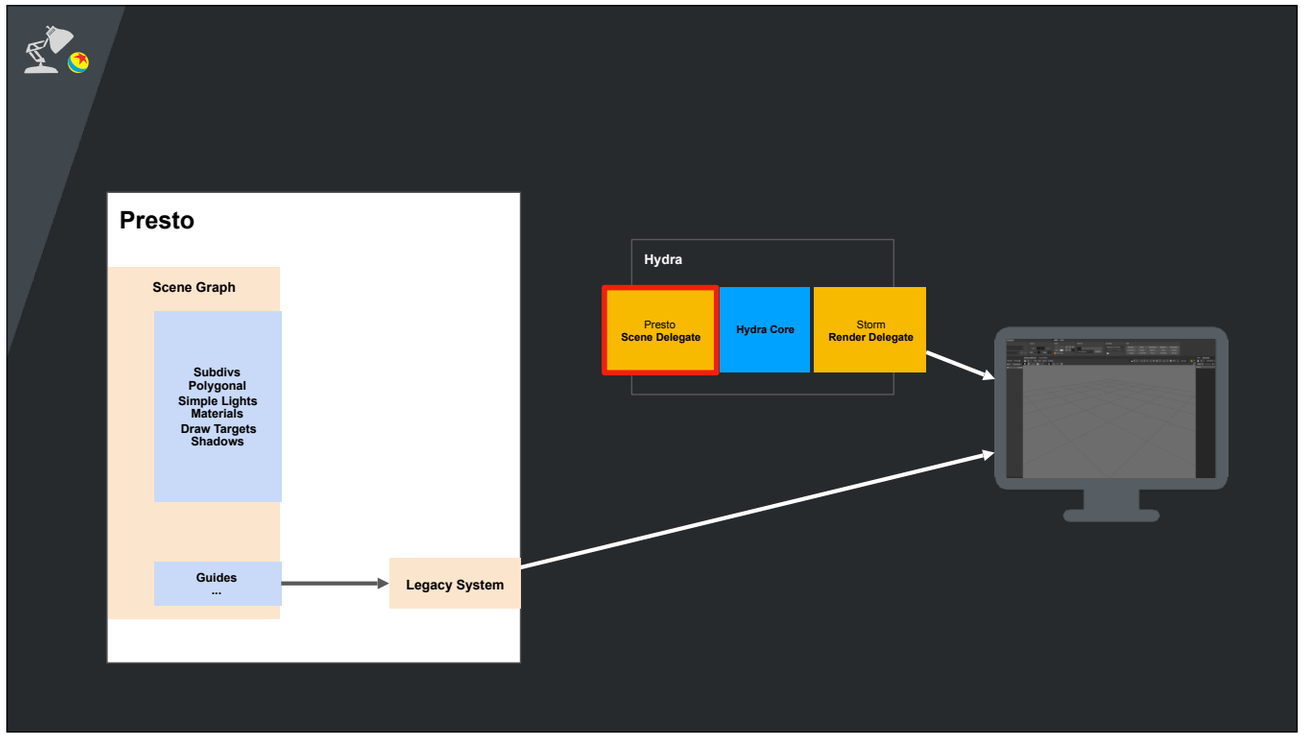

Presto
Scene Graph
Subdivs
Polygonal
Guides
...
USD
Legacy System
UsdImagingGL
USD
Scene Delegate
Hydra Core
Storm
Render Delegate

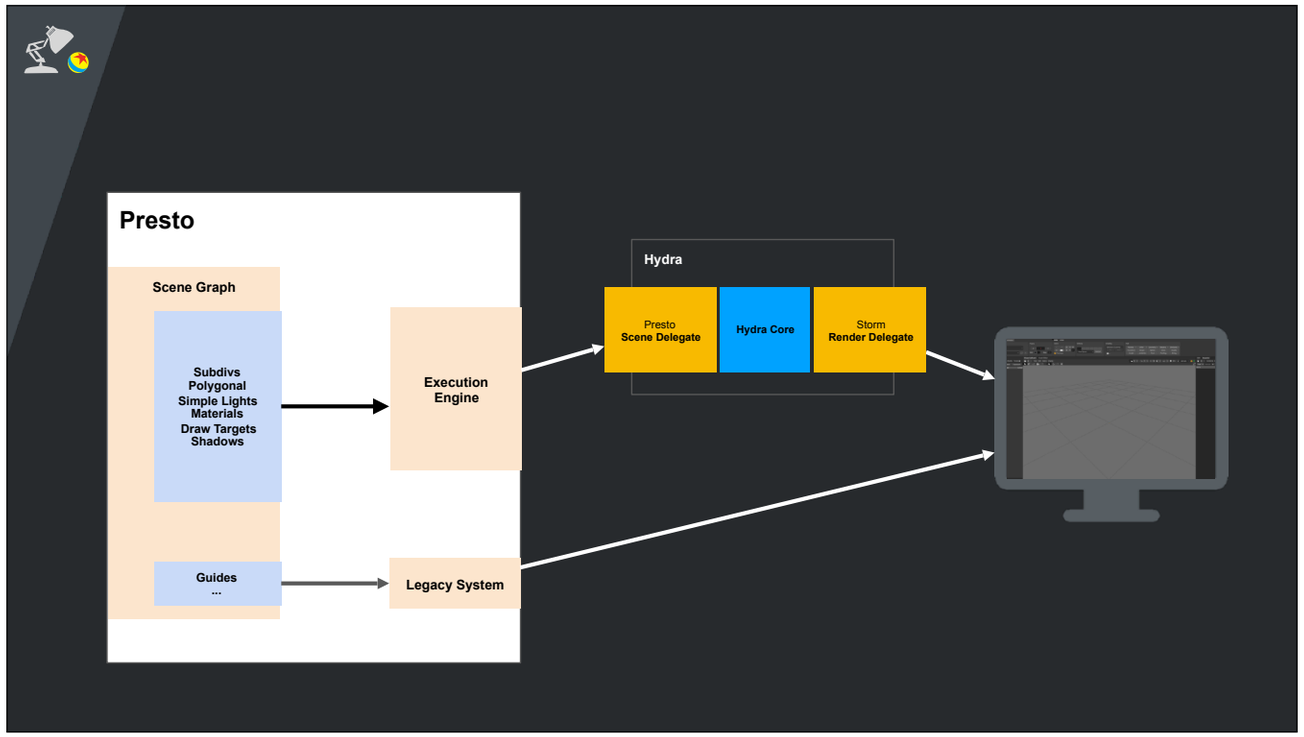

Presto
Full Hydra Integration

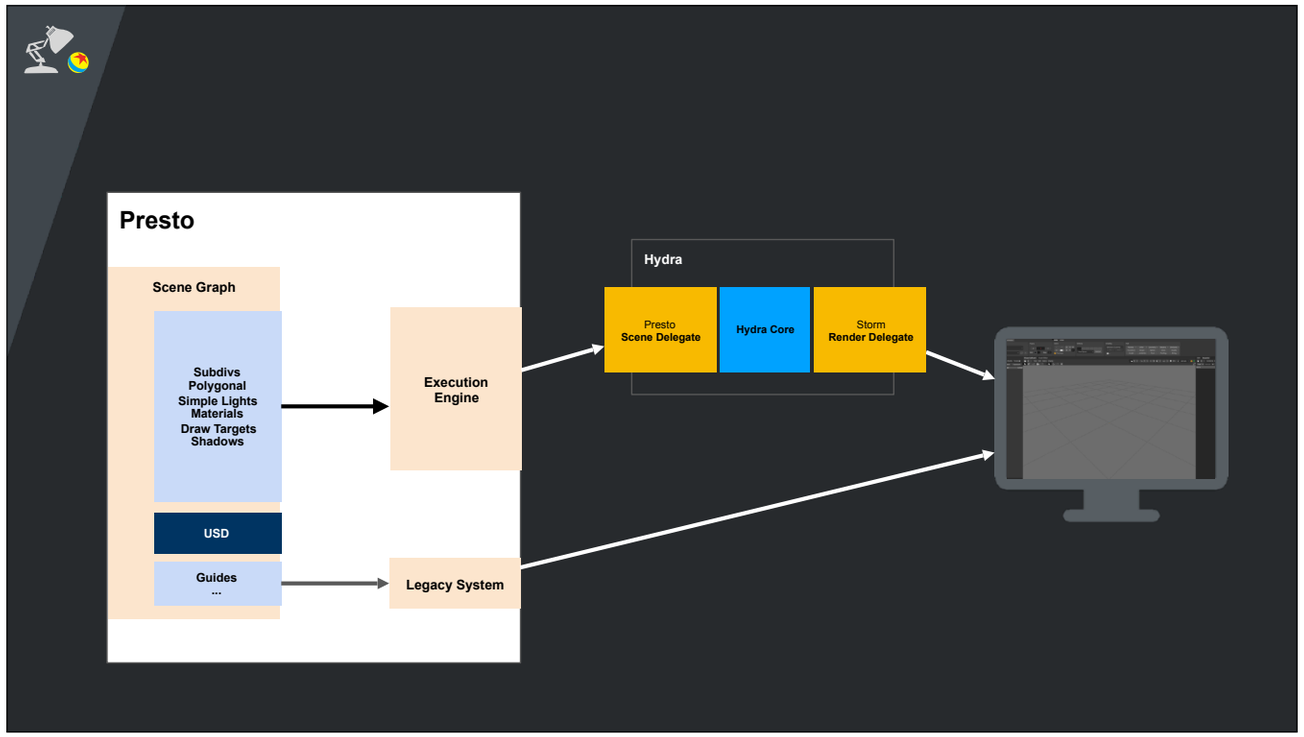


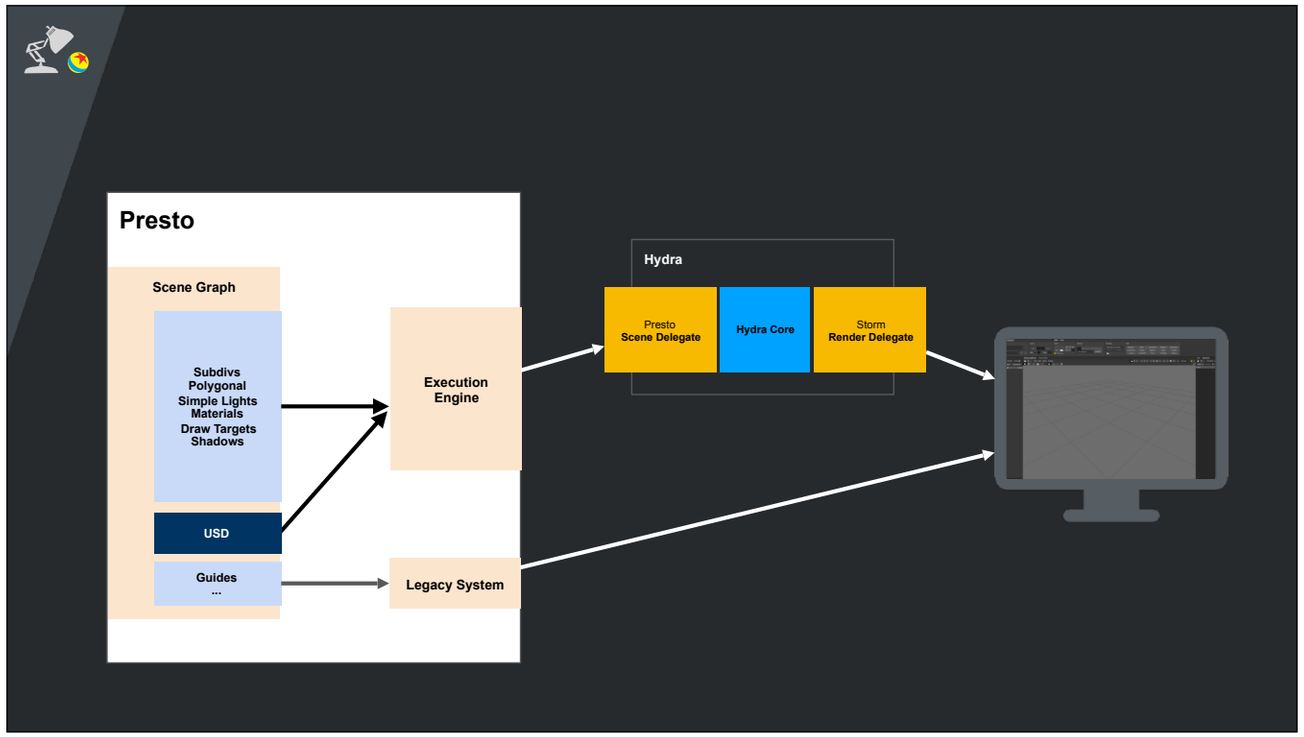
I should say that we didn’t want to do Guides because they are OpenGL callbacks and it was a lot to tackle.

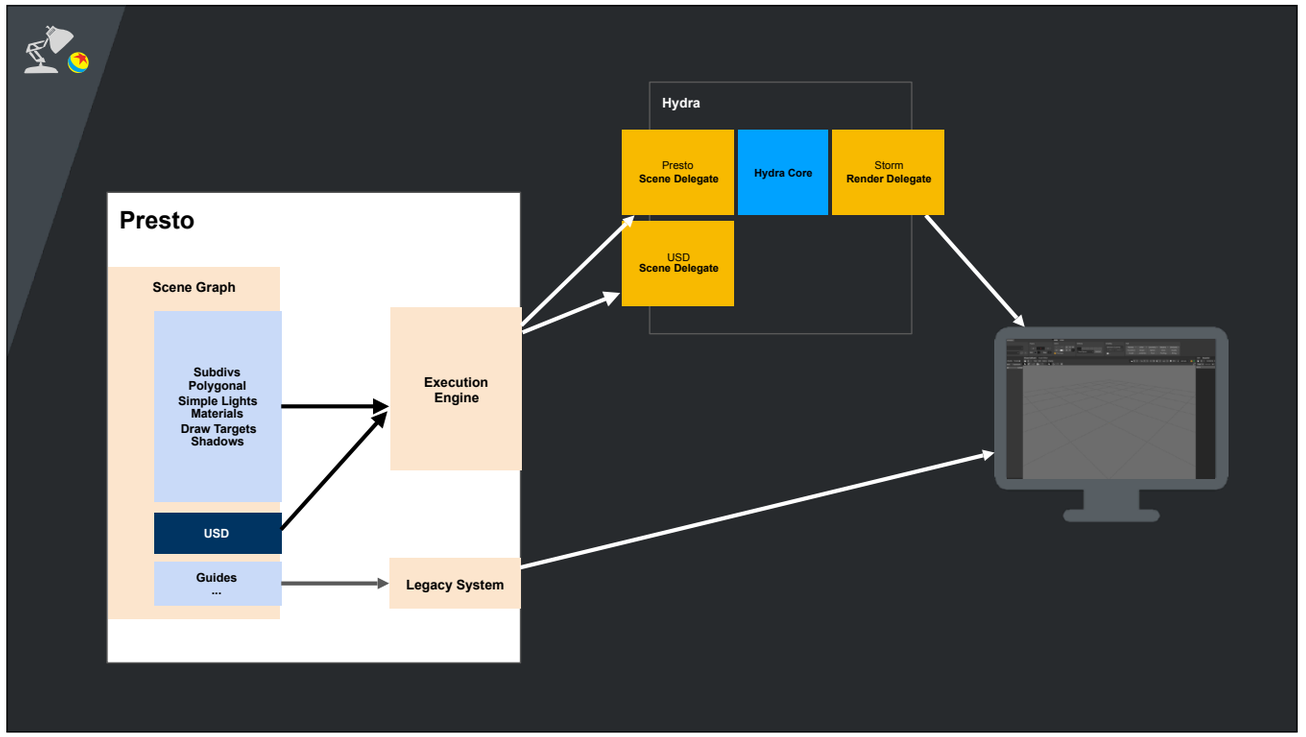

Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
Guides
...
Legacy System
Hydra
USD
Scene Delegate
Hydra Core
Storm
Render Delegate

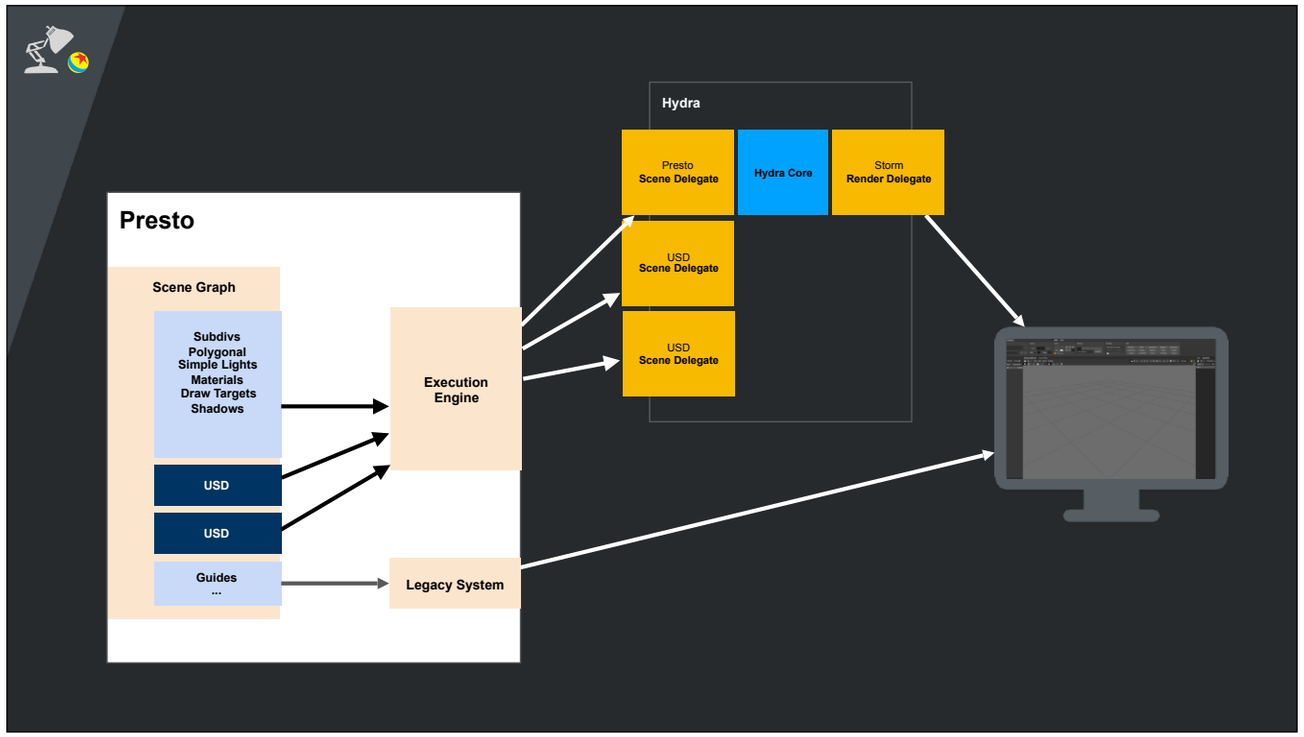

Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
Guides
...
Legacy System
Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate


Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
Guides
...
Execution Engine
Legacy System
Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate

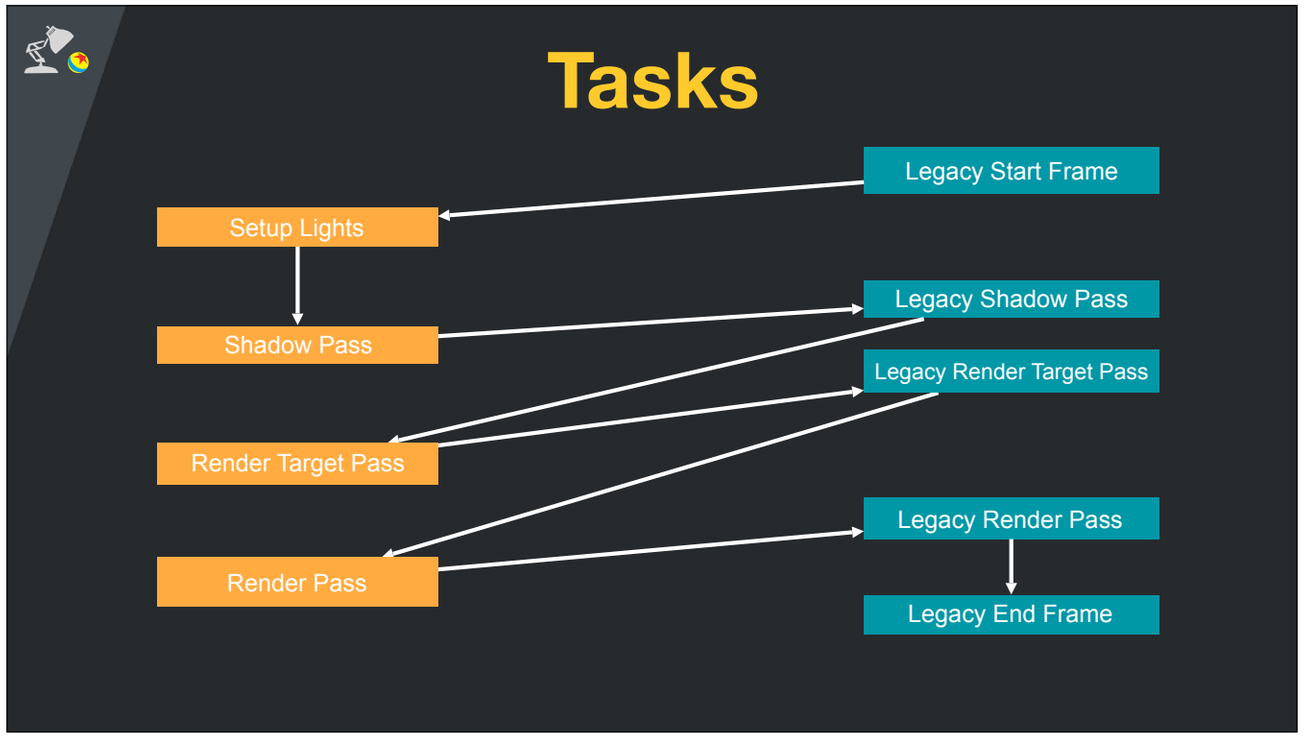

Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
USD
Guides
...
Execution Engine
Legacy System
Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate

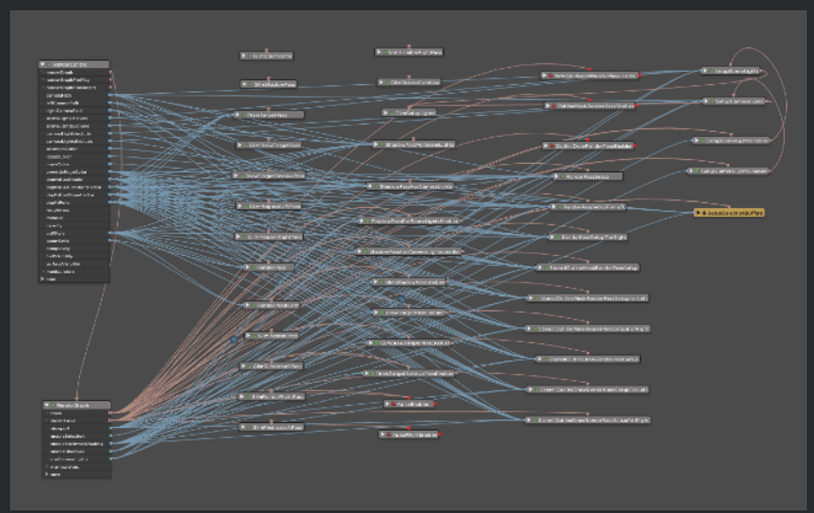

Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
USD
Guides
...
Execution Engine
Legacy System
Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate


Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate
USD
Scene Delegate
Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
USD
Guides
...
Execution
Engine
Legacy System


Hydra
Presto
Scene Delegate
Hydra Core
Storm
Render Delegate
USD
Scene Delegate
USD
Scene Delegate
Presto
Scene Graph
Subdivs
Polygonal
Simple Lights
Materials
Draw Targets
Shadows
USD
USD
Guides
...
Execution
Engine
Legacy System

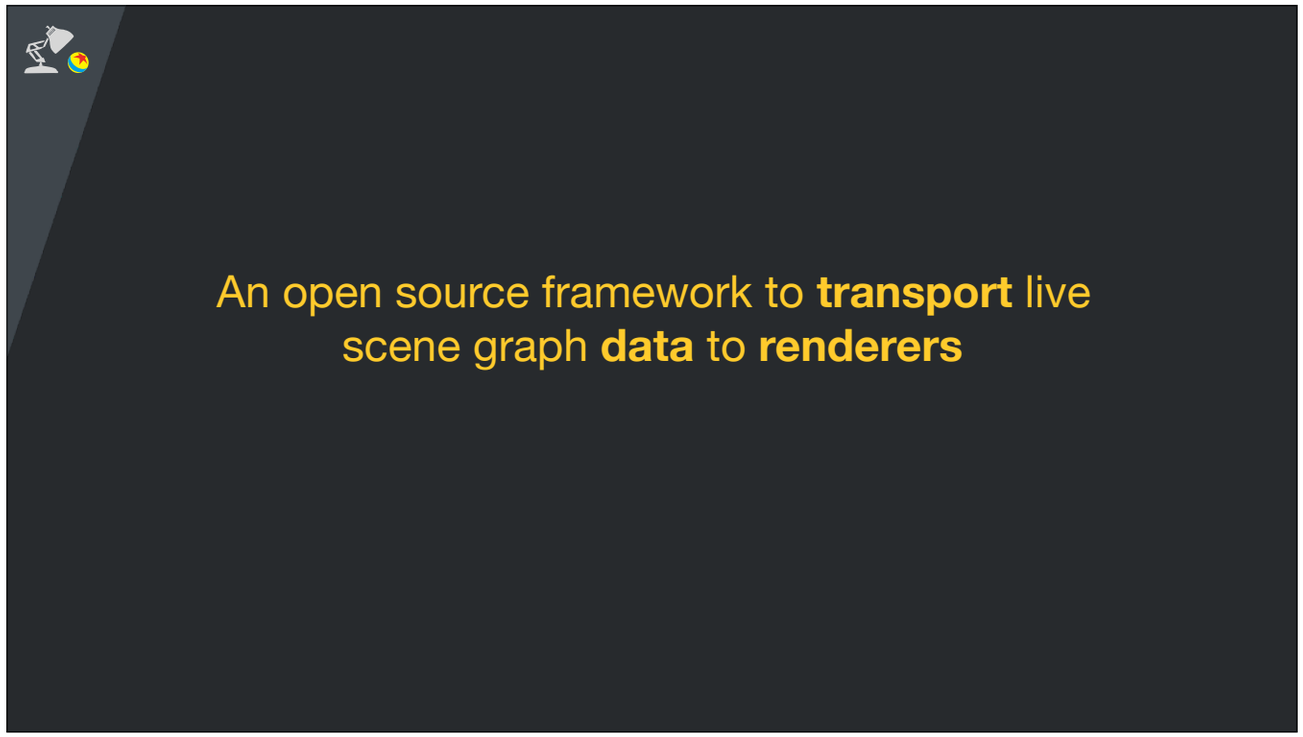

Presto
Hybrid Rendering


Tasks
Legacy Start Frame
Setup Lights
Legacy Shadow Pass
Shadow Pass
Legacy Render Target Pass
Render Target Pass
Legacy Render Pass
Render Pass
Legacy End Frame

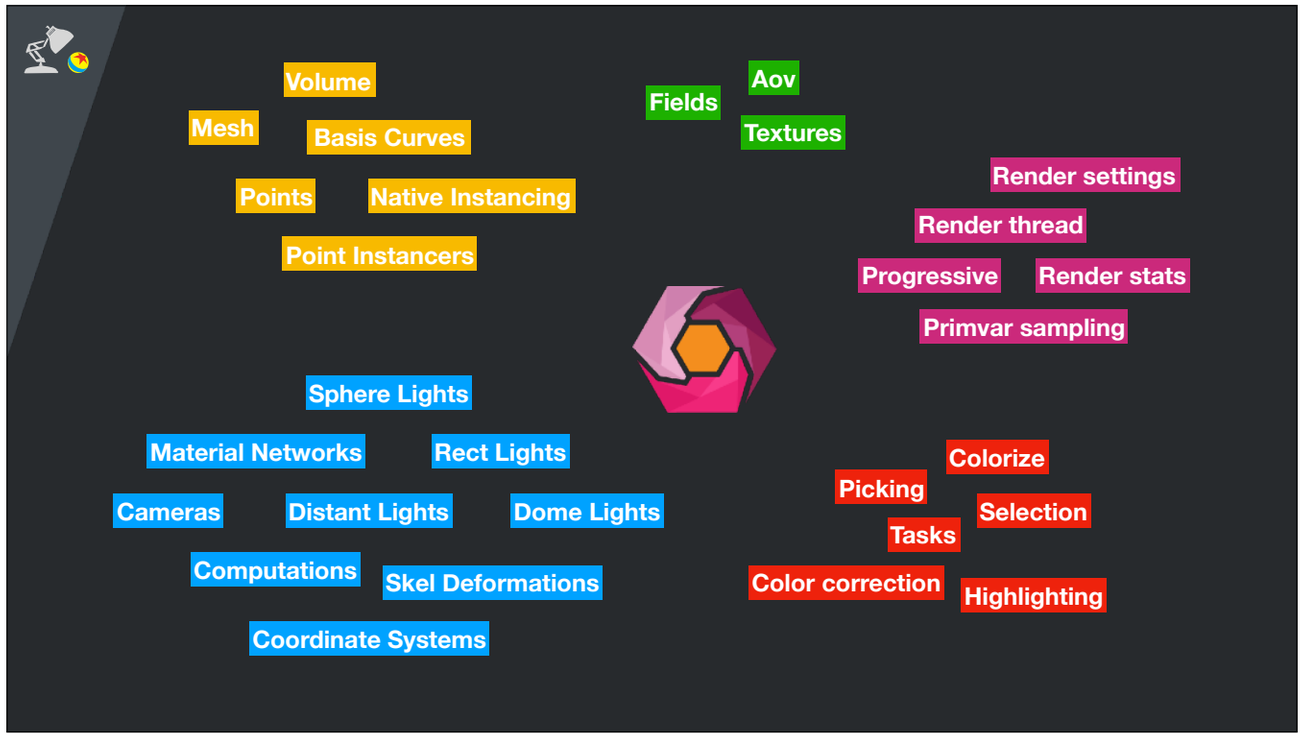


Summary

To summarize what we have seen today…



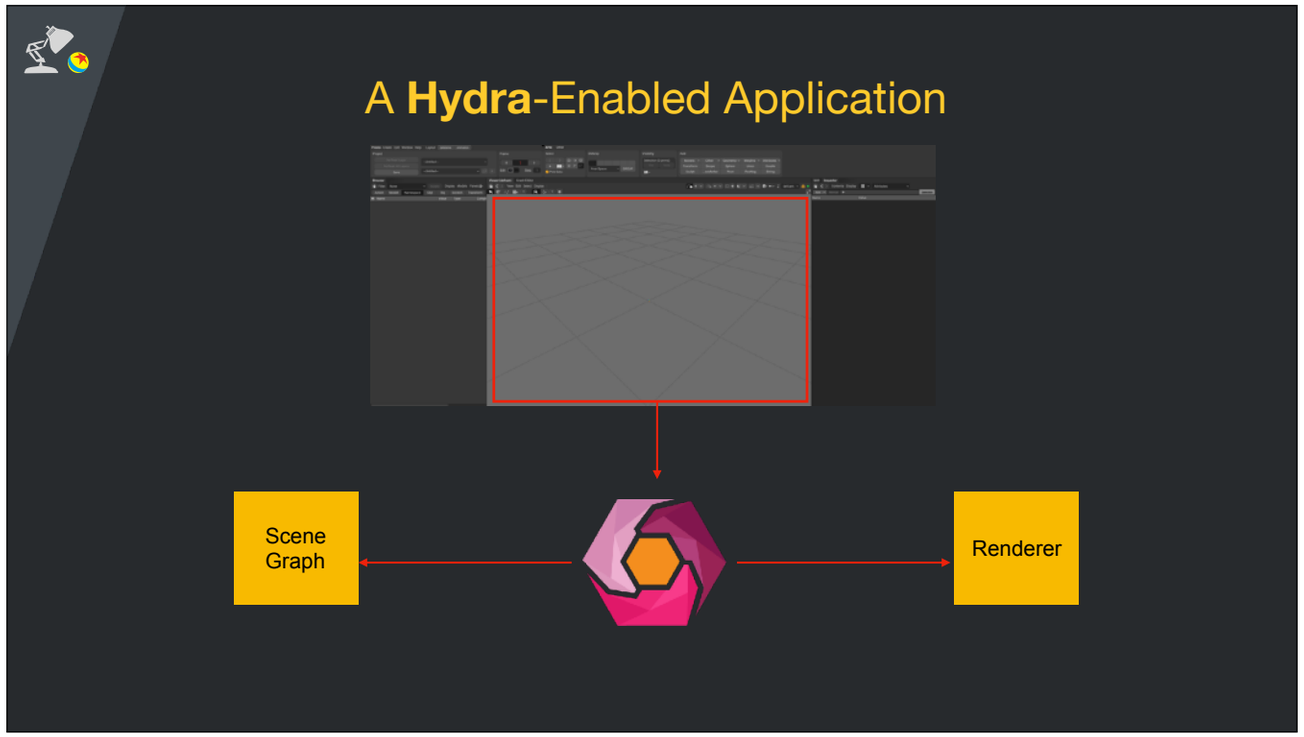
Hydra is an open source framework to transport live scene graph data to renderers.



Hydra can transport Rprims like volumes or meshes, Bprims like textures, Sprims like lights.

It provides building blocks to support multithreaded rendering, progressive renderers, sample primvars, our render settings.

And it comes with a set of tasks ready to use that can provide colorization, highlighting and more.



When you have integrated Hydra in your application, you can use USD in your application, and you gain access to our rasterizer Storm and our path tracer RenderMan.

But not just that, you also have access to multiple plugins and examples that are being build by the open source community.

Once again, if you have any questions about what we covered today or didn’t cover, please feel free to reach out to us. We are actively working on all these pieces and any feedback is welcome!

Alright! We are now gonna take another 5 minutes break, we will be here if you have any questions.

Thank you!