

The image features the Bungie logo centered on a dark, starry background. The logo consists of the word "BUNGIE" in a white, sans-serif font. A white arc is positioned above the letters "N" and "G", starting from the top of the "N" and ending at the top of the "G".

BUNGIE®

DESTINY®



# TFX

## Destiny Shader Pipeline

Natalya Tatarchuk

Chris Tchou

BUNGIE

DESTINY 

Hello! Welcome to the talk about Destiny Shader Pipeline.



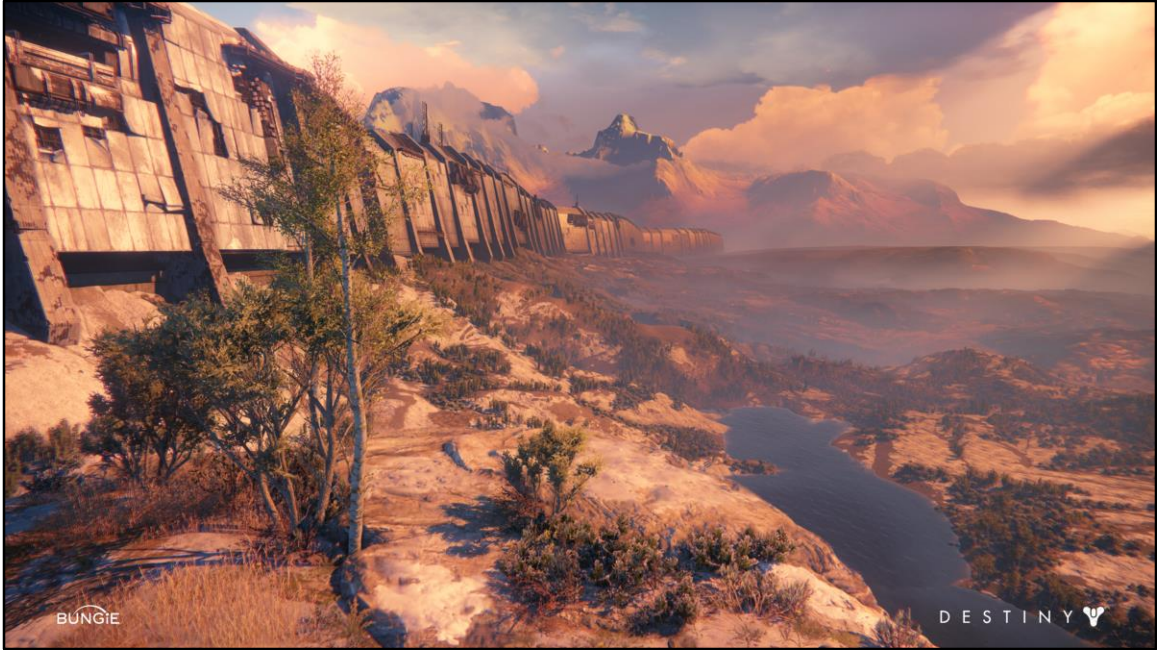
In this talk we will be describing the shader pipeline system we developed for creation of Destiny games.

## For Some Context...

[GDC 2015]



We have covered some of the details of the Destiny renderer architecture in the GDC 2015's talk (<http://www.gdcvault.com/play/1021926/Destiny-s-Multithreaded-Rendering>) which we recommend you take a look at if you want more context. But don't worry, anything relevant for this talk, we will be sure to cover here directly!



Destiny is a vast and diverse world.

Starting in the wilderness of Russia,



We may move to a giant evil spaceship orbiting Saturn,



Delve into deep dark dungeons





Climb Snow-swept mountains



Explore the sandy dunes of Mars



Or the desolate surface of the moon



A Mysterious space station



Wild frontiers



Alien worlds



And Crumbling European cities, just to name a few.



Destiny worlds are complex, alive and beautiful.





Destiny players explore large destinations, with diverse environments, lush vegetation.



All of this required a renderer with high-quality lighting, dynamic time of day, real-time shadows, high-resolution rendering, and a host of other rendering systems.



We wanted our renderer and shader pipeline to be data-driven, to respond to changes quickly and automatically.



We wanted to support a high level of dynamic content.

For example, dynamic time of day – here's Mars during the bright sun midday.



And from another vantage point, this is what Mars looks like in the middle of the night



To reach as many players as possible, we ship games on a multitude of platforms



And to handle dynamic situations in game with reliable frame rate, we multithreaded everything in our engine including all of the rendering.



We also needed a great variety of different materials and effects.





And everything needed to be fast. In each frame in our game we:



- Submit several dozens of render passes



- Process thousands of drawcalls



- Execute heavy CPU and GPU workloads all while ensuring we have low latency responsiveness in the game



Destiny also provided players with an incredible variety of armor, guns



And player gear



While we define shaders or shader elements in code, most of the shader content is now authored by artists directly (including shader parameters).



We had less than ten graphics engineers on Destiny, while we had several hundred artists.





We wanted to create a shader pipeline system that allowed us to unleash the creativity of the artists - create visuals programmers didn't think of (as you can see in these exotic gear examples here)



We also had to handle custom materials such as subsurface scattering on skin...



and anisotropic material model for hair



We needed our shader system to support deep customization – with little changes to content we wanted to create drastically varied looks – such as this floating purple balls of death



That could turn into Floating red balls of death with a toggle of a shader variant



The incredible shader variety was particularly important – and in fact – required for effects and custom visuals (such as the Taken character here and its sword).



giant robots with guns



Undead bony guys with swords





Flying drones with guns



Flying space wizards



This friendly guy



And these nice fellas..



















And just to give you an idea of the scale per frame – this Venus Destination uses over 3500 unique shader techniques at runtime for just this frame (3102 - 3518 techniques) (this is after collapsing non-unique materials) (on Xbox One)



And in a different area of our Cosmodrome destination we use over 9000 techniques used for the frame submission  
(Cosmo Ambient Activity: 9107 techniques) (on Xbox One)



We had a huge variety of shaders authored by artists – in the Taken King there were 18,000 artist authored shaders (content) (Although a number of them shared vast portions of their subgraphs from a few key templates).

## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline

BUNGIE

DESTINY 

From the technical perspective, we setup the following goals for the TFX shader system:

- We want the shader pipeline architecture to be agnostic of the specific material representation (for example, PBR or alternatives), and rendering submission pipeline, like deferred or forward. It should let us express either approach without having to change the underlying system

## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline
- Encapsulate majority of the submission state per draw call

BUNGIE

DESTINY 

- Defines much of submission state for any given draw / dispatch call with good encapsulation mechanisms



## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline
- Encapsulate majority of the submission state per draw call
- Cross platform *write once / use many* shader authoring

BUNGE

DESTINY 

- Supports multiplatform “write-once/use many” authoring paradigm – we only want to write the shader code once

## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline
- Encapsulate majority of the submission state per draw call
- Cross platform *write.once / use many* shader authoring
- Easily extensible

BUNGE

DESTINY 

- Easily extensible to support growing material library – we ship DLCs and we want to be able to grow as needed without any pain, for example, when we added snow materials in the Rise of Iron

## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline
- Encapsulate majority of the submission state per draw call
- Cross platform *write once / use many* shader authoring
- Easily extensible
- **Optimize redundant state changes**

BUNGIE

DESTINY 

- Make the system performant = optimize automatically to reduce redundant state changes which yields better CPU and GPU performance and lower memory footprint

## TFX Shader Technical System Goals

- Agnostic of material representation and/or rendering pipeline
- Encapsulate majority of the submission state per draw call
- Cross platform *write once / use many* shader authoring
- Easily extensible
- Optimize redundant state changes
- Artist-friendly automatically-built front-end

BUNGE

DESTINY 

- Automatically build artist-friendly front end from the shader code with minimal markup. Allow artists to edit shaders using the visual node graph paradigm. Additionally, we wanted to support easy creation of new shader building blocks by the tech artists

## Shader System Design Principles

- Data-driven validated GPU / CPU state management

BUNGE

DESTINY 

When we designed TFX (along with our Destiny renderer), we relied on several key design principles that helped us make choices throughout development:

- Design for data-driven validated GPU / CPU state management – ensure that the GPU state is encapsulated from the source code / content data provided and that we always validate the state the shader expects versus what the runtime layers provide. This became particularly important once we started working on platforms with flexible low-level API that could easily result in GPU crashes

## Shader System Design Principles

- **Data-driven validated** GPU / CPU state management
- **Flexibility** of state management

BUNGE

DESTINY 

- Support flexible state management to allow handling of variety of situations we needed to ship. We did not want to have a system that needed to be re-designed any time a new edge case was discovered.

## Shader System Design Principles

- **Data-driven validated** GPU / CPU state management
- **Flexibility** of state management
- **Performant** submit based on optimized **frequency** of submission

BUNGE

DESTINY 

- Ensure that the system is performant on CPU by optimizing frequency of GPU state submission. The less we have to send data to the GPU, the less it costs us per frame.

## Shader System Design Principles

- **Data-driven validated** GPU / CPU state management
- **Flexibility** of state management
- **Performant** submit based on optimized **frequency** of submission
- **Locality** of state

BUNGE

DESTINY 

- However, especially in the previous generations of consoles, we did not have unlimited state registers to bind state, so we had to be very careful to ensure that we maintain locality of state to reduce state collisions for runtime submission
- Well-encapsulated interfaces hide complexity behind simpler interfaces (including platform differences)





Let's take a deeper look at the TFX shader system

## Quick TFX Terminology

- **TFX Source** – code in TFX language

BUNGIE

DESTINY 

First a quick set of definitions, because it can get a bit confusing sometimes:

Now: 'TFX Source' refers to code in the TFX language – which is a custom language that we treat as content.

## Quick TFX Terminology

- **TFX Source** – code in TFX language
- **Shader** – code that drives the GPU (i.e. pixel shader program)

BUNGE

DESTINY 

Shader refers to code that drives the GPU (for example: a pixel shader) –

this can refer to either a compiled native hardware format, or a source format like HLSL.

## Quick TFX Terminology

- **TFX Source** – code in TFX language
- **Shader** – code that drives the GPU (i.e. pixel shader program)
- **Node Graph** – the content creator interface file

BUNGE

DESTINY 

We have the Node Graph, which refers to the content creator interface file, where options and parameter values are chosen.

Confusingly, this is also sometimes called a 'shader' or a 'material', but we'll try to keep it straight.

## Quick TFX Terminology

- **TFX Source** – code in TFX language
- **Shader** – code that drives the GPU (i.e. pixel shader program)
- **Node Graph** – the content creator interface file
- **Technique** – shader programs and corresponding GPU state

BUNGE

DESTINY 

The Technique is the combination of a number of shaders and their corresponding GPU state.

This is basically the encapsulated state necessary to render any dispatch or drawcall.

## Quick TFX Terminology

- **TFX Source** – code in TFX language
- **Shader** – code that drives the GPU (i.e. pixel shader program)
- **Node Graph** – the content creator interface file
- **Technique** – shader programs and corresponding GPU state
- **HLSL** – cross-platform high level shader language (HLSL-like)

BUNGIE

DESTINY 

And, finally, when we say “HLSL” we just mean any one of our platform’s shading languages (it could HLSL, PSSL, CG.. or similar)

## TFX Shader System Key Elements

- Authoring shaders
  - TFX Source and Node Graph

BUNGE

DESTINY 

Breaking down the TFX shader pipeline, we will go over these key elements:

- First, how do we author shaders? (This refers to authoring both the TFX shader source, and the Node Graphs.)

## TFX Shader System Key Elements

- Authoring shaders
- Managing shader data sources
  - Static (content) and dynamic (engine) shader inputs

BUNGIE

DESTINY 

Where do we get the data to feed to the GPU?

Managing shader data sources is an important part of TFX. It handles both static data sources from content files (like textures, constants, or animation curves), as well as dynamic sources from the runtime engine (for example, sampling from render targets as textures, or binding data from game objects)



## TFX Shader System Key Elements

- Authoring shaders
- Managing shader data sources
- Managing and validating GPU state
  - Encapsulation of GPU state
  - Managing frequency of submission for GPU state
  - Validating expected GPU state

BUNGIE

DESTINY 

Another key element is how we manage and validate GPU state for the shaders. How do we encapsulate GPU state coherently, and manage its frequency of submission to avoid redundant state setting.

And how do we validate that we have all the correct state specified and aren't corrupting GPU state for any other elements of the frame?

## TFX Shader System Key Elements

- Authoring shaders
- Managing shader data sources
- Managing and validating GPU state
- Managing shader permutations
  - Components
  - Templates
  - Variants

BUNGIE

DESTINY 

We also needed to determine a good way to manage shader permutations.

This involves using components to encapsulate shader options, as well as drive nodes in the node graph, and the connections between them.

Then we have a templating system, allowing easy sharing of functionality and values between Node Graphs.

And we also have the variant layer system, which allows us to store many permutations of a shader in a single Node Graph file.

## TFX Shader System Key Elements

- Authoring shaders
- Managing shader data sources
- Managing and validating GPU state
- Managing shader permutations
- Integration with the core renderer and engine architecture

BUNGE

DESTINY 

And finally, there are numerous details on how we integrated the shading system into the core renderer and engine architecture.

## TFX Shader System Key Elements

- Authoring shaders
- Managing shader data sources
- Managing and validating GPU state
- Managing shader permutations
- Integration with the core renderer and engine architecture

BUNGE

DESTINY 

We will cover these key elements of the TFX system in this presentation in-depth.

Hopefully, we can give you a good understanding of what drove our system's design, and how it worked for our game, to give you some ideas of how you can adopt these principles for the design of your game systems.

## TFX Pipeline Overview



Define Nodes,  
Parameters,  
HLSL  
fragments

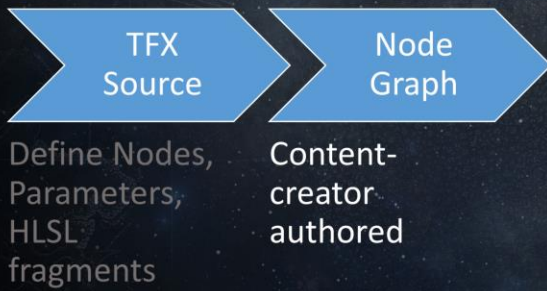
BUNGIE

DESTINY 

**So this is the basic TFX pipeline.**

We start with the TFX Source files;  
These allow a graphics programmer or technical artist to define nodes, parameters and fragments of HLSL code, using the TFX language.

## TFX Pipeline Overview

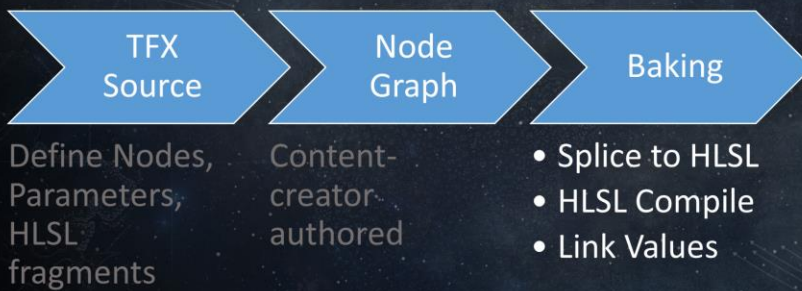


BUNGIE

DESTINY 

The TFX source files are then used to automatically build the node graph interface, allowing content creators to construct node graphs, defining their shaders.

## TFX Pipeline Overview



BUNGIE

DESTINY 

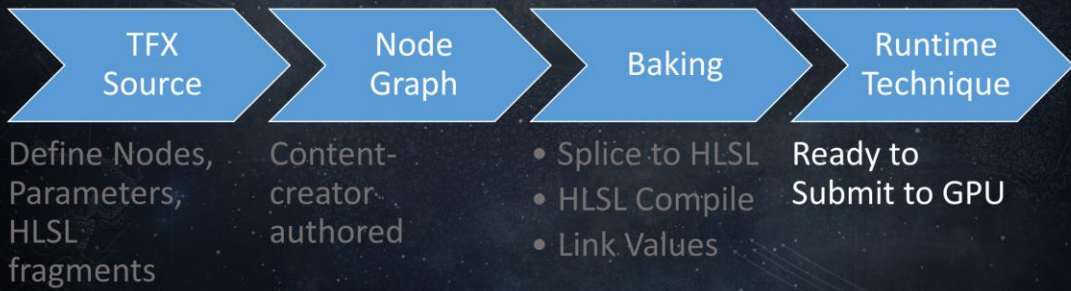
At bake time, we compile our shaders in three steps:

First we generate the HLSL by splicing together the HLSL fragments defined by the TFX source.

This is then passed off to the platform's shader compiler,

And the result of that is linked to the parameter values, defined in either the node graph or the TFX source.

## TFX Pipeline Overview

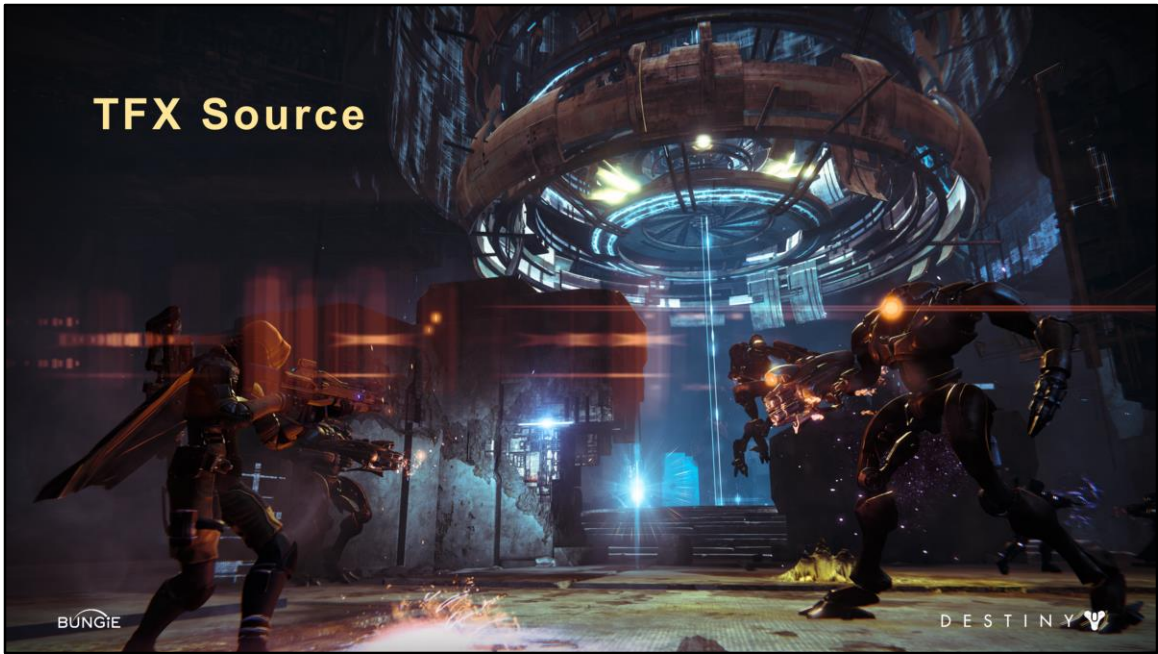


BUNGIE

DESTINY 

The final result of all this is a runtime technique that is ready to submit to the GPU at runtime.





How exactly does the TFX Source work? It is written in the TFX language,

## TFX Language

- Thin layer on top of HLSL

BUNGE


DESTINY 

Which is built as a thin layer on top of the underlying HLSL language.

## TFX Language

- Thin layer on top of HLSL
- A shader language pre-processor

BUNGIE

DESTINY 

TFX acts very much like a fancy pre-processor; it doesn't really understand the code, but it knows enough to manipulate it.

## TFX Language

- Thin layer on top of HLSL
- A shader language pre-processor
- Copy/paste HLSL **fragment splicing**



BUNGE

DESTINY

Remember, we use copy/paste splicing to generate the HLSL code to compile.

And this is not JUST laziness on our part..

## Advantages of Splicing

- All underlying language features are available
- Easy to convert existing shaders to TFX
- Easy to support new platforms or languages

BUNGIE

DESTINY 

There are some advantages to splicing.

Every language feature of HLSL is available; which makes it super simple to convert existing shaders to TFX or understand how they work.

It also makes it easier to add new platforms or languages, as we don't have to write code to generate those languages.

And on the next slide, we can see what these HLSL fragments that we are splicing look like.

## TFX Language Features

- Organize GPU state with **scopes**
- Wrap functionality in **components**
- Node graph **UI controls**
- Complex **dynamic expressions** of game state
- Flexible **metadata**
- And more!

BUNGIE

DESTINY 

On top of this basis, TFX also adds a bunch of other features:

- It gives you the ability to organize state with Scopes,
  - You can use components to wrap up functionality, similar to classes in C++
  - It builds the node graph interface automatically
  - It also lets us build complex expressions of game state, and evaluate them efficiently at runtime.
  - We also have a flexible metadata system that lets us add functionality easily.
- And of course much more

Let's take a quick look at a simple TFX shader:

## Simple TFX Source

```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

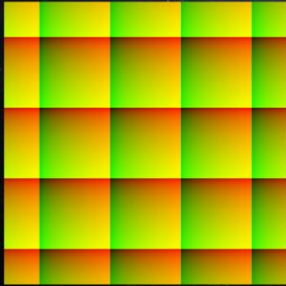
technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

DESTINY 

So, this is about as simple of a shader as you can get -- no parameters, 100% procedural.

## Simple TFX Source



```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

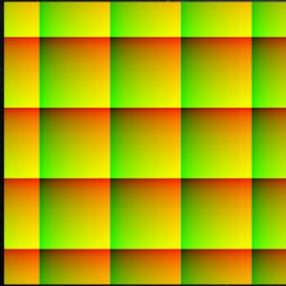
DESTINY

All this shader really does is just take the texture coordinates from the interpolators, scale them and write them out to the render target.

On the left you have what it would look like.



## Simple TFX Source



```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

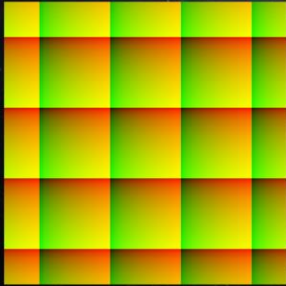
BUNGIE

DESTINY

At the top we have “import” commands to pull in definitions from other files.

I’ve separated the vertex shader and interpolators into their own files, to make this slide more readable.

## Simple TFX Source



```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

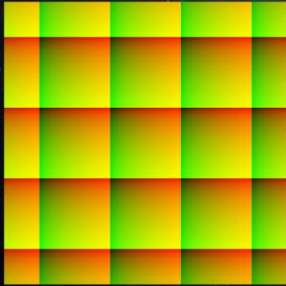
technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

DESTINY

Then this section here between #hls1 and #end is an HLSL FRAGMENT  
This is a chunk of HLSL shader code that will get copy/pasted into the generated HLSL  
when we go to do splicing.

## Simple TFX Source



```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

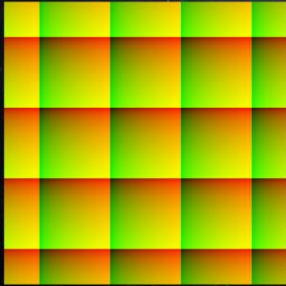
DESTINY

Note that the code in the fragment is actually cross platform.

We do the standard thing of using preprocessor defines to standardize the syntax – for example TFX\_TARGET0 here, gets defined as either SV\_TARGET0, or COLOR0 or S\_TARGET\_OUTPUT0 depending on which platform you are baking out.

If your language is sufficiently different from others, you could also make use of preprocessor conditionals, like #if #else, to do more drastic changes to the code on each platform.

## Simple TFX Source



```
import "main_vs.tfx"
import "interpolators.tfx"

#hlsl
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

technique my_technique
{
    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

DESTINY

The technique declaration at the bottom, defines how to **build** the technique during the baking process.

Basically it is saying: if someone requests 'my\_technique' then we need to compile both a vertex shader and a pixel shader for all platforms, using the 'main\_vs' and 'main\_ps' entry points.

## compile\_shader(...)

1. Splice HLSL fragments
2. Compile generated HLSL code
3. Link parameter values

All compiled shaders and corresponding parameter values  
→ runtime technique → GPU

BUNGE

DESTINY

Each of these `compile_shader` commands invokes the three operations: of splice, compile, and link.

All of the compiled shader programs, plus the corresponding GPU state descriptions, are combined to make the runtime technique that we can then submit to the GPU.

So let's take a look at what our example shader looks like after it is spliced together:

```

// entry_point: main_ps   shader: ps_5_0

// global_d3d11.tfx
#define TFX_POSITION      SV_Position
#define TFX_TEXCOORD0    TEXCOORD0
#define TFX_TARGET0      SV_Target0

// interpolators.tfx
struct s_ps_in {
    float4 pos : TFX_POSITION;
    float2 texcoord : TFX_TEXCOORD0;
};

// example.tfx
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}

```

## Spliced HLSL Output

DESTINY 

TFX has spliced the global definitions for the given platform at the top, as well as the HLSL fragments coming from various files.

This code would then be fed into the shader compiler in order to build the pixel shader.

So, now let's make a bit more interesting, and add some parameters to this.



Parameters and the GPU state they control is where much of the magic lies...

## Parameters are not Copy/Paste

TFX *does* understand parameter declarations

- Necessary to manage GPU state

During splice:

generate HLSL to declare GPU state for parameters

BUNGE

DESTINY 

Whereas TFX is very much hands-off with the HLSL fragments, it gets it's hands dirty dealing with the parameters.

Because one of our goals is to manage and organize GPU state, TFX has to understand the parameters completely.

During splicing, TFX generates the HLSL code to declare the GPU state for each parameter.



## Data-Driven Shader Parameters

- No rebuilds from any declaration or definition changes
  - Neither editor nor engine
- Automatically pick up the changes
  - UI
  - Baking
  - Runtime



BUNGE

DESTINY

We want to author parameters and not have to recompile code for the engine (preprocessor or runtime) (this also include the editor code)

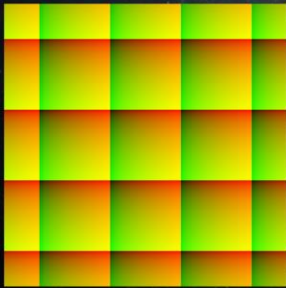
We want to ensure that all the changes required to change shader parameters are in one place (in the shader source code only) keeping them local to the location of authoring.

We want our bake, runtime and Node Graph UI automatically picks up the changes

The UI should be automatically building UX elements for shader components

The runtime engine manages GPU state automatically from the declared parameters in a data driven mannerWe want them to be tightly bound to the TFX state

## Adding Parameters



```
// artist parameters for scale/offset transform
float2 scale      @default(float2(1.0f, 1.0f));
float2 center     @default(float2(0.5f, 0.5f));
float2 offset     @default(float2(0.0f, 0.0f));

#hlsl
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale,
        offset + center - center * scale);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

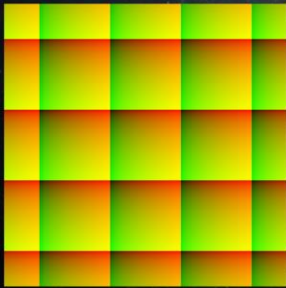
BUNGIE

DESTINY

We've added three parameters, 'scale', 'center', and 'offset', to control the transform we apply before writing to the render target.

Because these parameters are not assigned to anything, they will automatically be picked up as tweakable values in our shader node graph interface.

## Adding Parameters



```
// artist parameters for scale/offset transform
float2 scale @default(float2(1.0f, 1.0f));
float2 center @default(float2(0.5f, 0.5f));
float2 offset @default(float2(0.0f, 0.0f));

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale,
        offset + center - center * scale);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

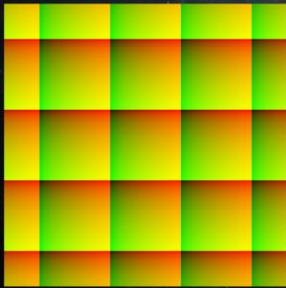
    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

We also have a mechanism to specify default values for the parameters – these are the values used if no one specifies a value in the node graph.

## Adding Parameters



```
// artist parameters for scale/offset transform
float2 scale @default(float2(1.0f, 1.0f));
float2 center @default(float2(0.5f, 0.5f));
float2 offset @default(float2(0.0f, 0.0f));

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale,
        offset + center - center * scale);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

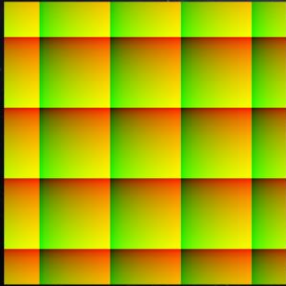
BUNGIE

DESTINY

The defaults here make use of the TFX metadata system – anything starting with the @ symbol is a metadata tag.

You can add whatever metadata you wish, 'default' is just the one it looks for to find default values for parameters.

## Adding Parameters



```
// artist parameters for scale/offset transform
float2 scale @default(float2(1.0f, 1.0f));
float2 center @default(float2(0.5f, 0.5f));
float2 offset @default(float2(0.0f, 0.0f));

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale,
        offset + center - center * scale);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

The meta-data system is incredibly useful in a large number of situations where you might otherwise have to go back and write some new language features to handle special cases. We'll touch base on other use cases for metadata in our system later on in this talk.

## Adding Parameters



```
// artist parameters for scale/offset transform
```

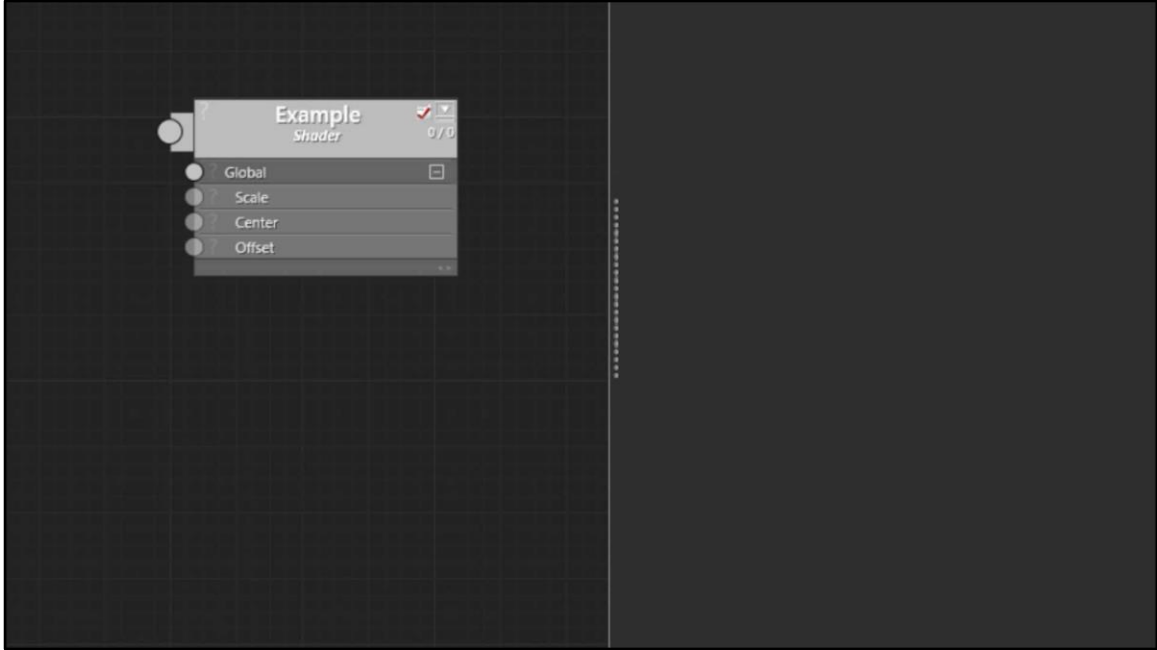
```
float2 scale    @default(float2(1.0f, 1.0f));  
float2 center   @default(float2(0.5f, 0.5f));  
float2 offset   @default(float2(0.0f, 0.0f));
```

```
#hls1  
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0  
{  
    float4 xform= float4(  
        scale,  
        offset + center - center * scale);  
  
    float2 transformed_texcoord=  
        frac(ps_in.texcoord * xform.xy + xform.zw);  
  
    return float4(transformed_texcoord, 0.0f, 1.0f);  
}  
#end
```

BUNGIE

DESTINY

So these parameters are now exposed to the artists for editing directly in our visual node graph editor..



Each of the three parameters is shown on the shader node,  
and by selecting the node, you can edit the parameter values.

**(video 1)**

## What Next?

Want more than a scale/offset transform?  
rotation, warps...

Older systems:  
Choose one!  
Activate code with preprocessor #define

BUNGIE

DESTINY 

What next?

Your content creators probably won't be happy with just a scale/offset transform.

They might want a rotation too.. or some kind of warp...



## What Next?

Want more than a scale/offset transform?  
rotation, warps...

Older systems:  
Choose one!  
Activate code with preprocessor #define

What about combinations of transforms?  
**preprocessor #defines don't scale well**

BUNGIE

DESTINY

On older systems, this kind of choice was often represented using preprocessor defines that activated different sections of code.

But this doesn't scale well to combinations of transforms.

This is one of the main reasons (but definitely not the only reason) behind a TFX feature called....



Which brings us to TFX Components!

## What Is a TFX Component?

Components are kind of like classes in C++

Member variables → Member parameters

Member functions → Member HLSL fragments

Can be instantiated

What is a Component?

You can think of components kind of like classes in C++.

- But Instead of member variables it has member parameters
- And instead of member functions it has member HLSL fragments
- And you can instantiate the whole thing, just like a class.

## Why Components?

- Components **define Shader Graph Nodes**
  - But not every component is a node...
- **Wrap common functionality**
  - Instantiation is better than copy/pasting code
- **Hide complexity**
  - Complex implementations
  - Platform differences

BUNGE

DESTINY 

Components serve a range of purposes:

- Mainly, they are a way of providing flexible shader **'options'**; for example, every node graph **node** is defined by a component.
- But components also can be used to wrap common functionality, so we can use it in many locations.
- Components also provide an interface, behind which you can hide complexity, or platform differences.

## Component Declaration

```
// interface: c_transform
// variant:   scale_offset

component c_transform:scale_offset
{
    // member parameters
    float2 scale   @default(float2(1.0f, 1.0f));
    float2 center  @default(float2(0.0f, 0.0f));
    float2 offset  @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

BUNGIE

DESTINY

This an example of a component declaration.

The name of the component is somewhat special –

## Component Declaration

- Two-part name  
`interface:variant`
- Same interface →  
interchangeable

```
// interface: c_transform
// variant: scale_offset
component c_transform:scale_offset
{
    // member parameters
    float2 scale @default(float2(1.0f, 1.0f));
    float2 center @default(float2(0.0f, 0.0f));
    float2 offset @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

BUNGIE

DESTINY

... it is broken into two parts:

interface colon variant

the idea here is that components with the same interface are **basically** interchangeable

So this code is declaring a component using the 'c\_transform' interface, with a variant called 'scale\_offset'.

## Component Declaration

Member parameters

```
// interface: c_transform
// variant:   scale_offset

component c_transform:scale_offset
{
    // member parameters
    float2 scale   @default(float2(1.0f, 1.0f));
    float2 center  @default(float2(0.0f, 0.0f));
    float2 offset  @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

BUNGIE

DESTINY

Next, we've moved the three parameters we used before into the component,

## Component Declaration

```
// interface: c_transform
// variant:   scale_offset

component c_transform:scale_offset
{
    // member parameters
    float2 scale   @default(float2(1.0f, 1.0f));
    float2 center  @default(float2(0.0f, 0.0f));
    float2 offset  @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

Member HLSL  
function



BUNGIE

DESTINY

And pulled out the transform code and moved it into the 'Apply' member HLSL function, declared in this fragment.



## Component Declaration

The `c_transform` interface requires:

`float2 apply(float2)`

This is implicit

- No declaration
- No enforcement

```
// interface: c_transform
// variant:   scale_offset

component c_transform:scale_offset
{
    // member parameters
    float2 scale   @default(float2(1.0f, 1.0f));
    float2 center  @default(float2(0.0f, 0.0f));
    float2 offset  @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

In this case we have decided that all components using the `c_transform` interface should have an `'apply'` function like this.

NOTE however, that this interface requirement is implicit – we didn't find any reason to actually declare or enforce the interface requirements.

## Component Declaration

The `c_transform` interface requires:

`float2 apply(float2)`

This is implicit

- No declaration
- No enforcement

```
// interface: c_transform
// variant: scale_offset

component c_transform:scale_offset
{
    // member parameters
    float2 scale @default(float2(1.0f, 1.0f));
    float2 center @default(float2(0.0f, 0.0f));
    float2 offset @default(float2(0.0f, 0.0f));

    // member HLSL function 'apply'
    #function(apply)
    float2 apply(float2 texcoord)
    {
        return texcoord * scale +
            offset + center * (1.0f - scale);
    }
    #end
}
```

You *could* make it an explicit interface, which may be worthwhile if you expect to have complex interfaces and want comprehensive errors to catch the non-compliant components.

But, in our case, the interfaces were generally fairly simple, so it wasn't worth the added complexity.

So this whole component declaration only *declares* the component, we need to *instantiate* it if we want to use it.

If we go back to our main shader code...

## Using Components

Instantiate  
Component  
instance:variant

```
import "main_vs.tfx"
import "interpolators.tfx"
import "transform_components.tfx"

// instance of c_transform:scale_offset
c_transform:scale_offset g_transform;

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        g_transform.apply(ps_in.texcoord);

    return float4(
        frac(transformed_texcoord),
        0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

.. go back to our main shader code...

We can instantiate it by name like so,

## Using Components

Use in HLSL

```
import "main_vs.tfx"
import "interpolators.tfx"
import "transform_components.tfx"

// instance of c_transform:scale_offset
c_transform:scale_offset g_transform;

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
    g_transform.apply(ps_in.texcoord);

    return float4(
        frac(transformed_texcoord),
        0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

And then use it's member values and functions, like this.

Instantiating a component by **name** like this will inline all of the members of that component into the parent node in the node graph.

## Using Components



```
import "main_vs.tfx"
import "interpolators.tfx"
import "transform_components.tfx"

// instance of c_transform:scale_offset
c_transform:scale_offset g_transform;

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        g_transform.apply(ps_in.texcoord);

    return float4(
        frac(transformed_texcoord),
        0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

Which means the node graph looks identical to what we had before – the only difference is under the hood, those parameters are coming from the component.

The *other* way to instantiate a component is ...

## Using Components

Instantiate Component interface:\*

Defines a node connection point

Connect to any matching component

```
import "main_vs.tfx"
import "interpolators.tfx"
import "transform_components.tfx"

// instance of c_transform:none component
c_transform:* g_transform @default(none);

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        g_transform.apply(ps_in.texcoord);

    return float4(
        frac(transformed_texcoord),
        0.0f, 1.0f);
}
#end
```

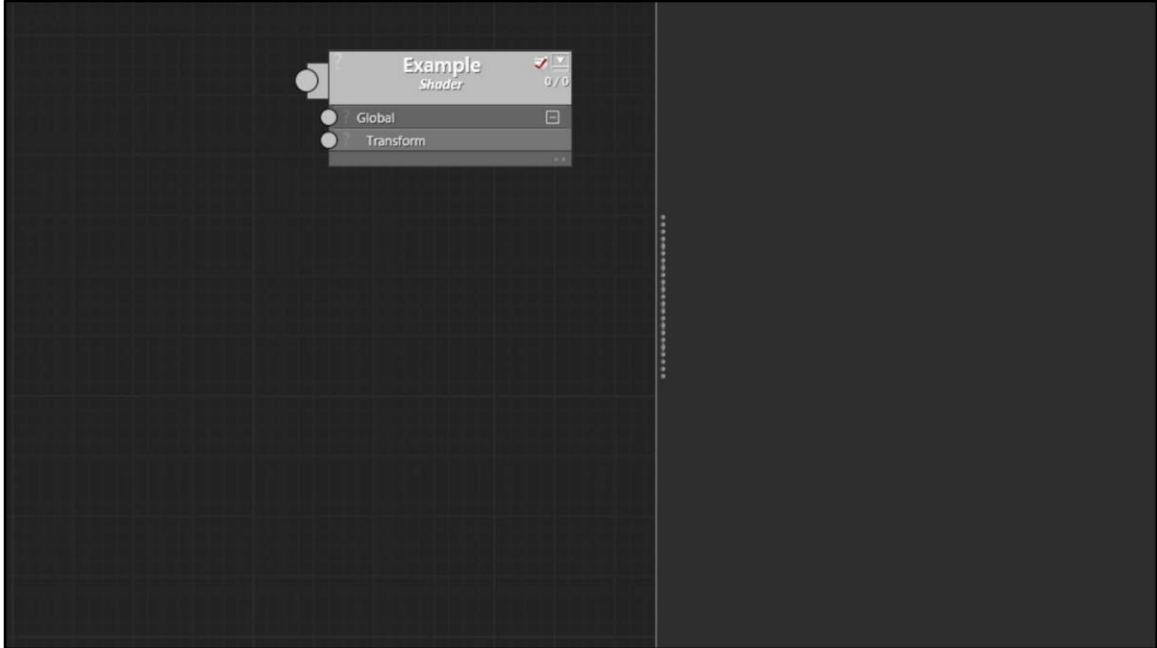
BUNGIE

DESTINY

... to use the interface:\* syntax.

This defines the instance to be **ANY** component that has the matching interface.

In the node graph, **this** instance will show up as a **node connection point**, that can be connected to **any** matching component.



This is now what your shader would look like in the node graph editor:

We have a single connection point for the 'transform', which we can connect to our `scale_offset` component, which then contains the three parameters we defined.

**Now** we have options -- we could, *instead*, connect the transform to a *different* component, say, a spherical warp.

One interesting thing to note is that the lines in our node graph actually represent the component instantiation connections – NOT values or function calls. but, interfaces are a superset of those, so our node graph can be used that way if you want.

**(Video2)**

## Connecting Components Together

Component interface defines how it connects  
(interface names must match)

Node Graph editor only allows matching connections  
Can filter node lists to compatible choices  
Highlight compatible connection points

BUNGIE

DESTINY 

So to summarize: The component interfaces define how they are plugged together; only matching interface names will connect.

So you define your components with their interfaces, and now the Node Graph editor can understand what are valid connections; it can do highlighting and filter the node lists so you only see the compatible nodes, which makes it a lot easier to work with.



## Connecting Components

```
component c_transform:combine
{
    // artist parameters - two transforms we want to combine
    // c_transform:* means any component using the
    //           c_transform interface
    // in the node graph, these show up as connection points
    // that can connect to c_transform nodes

    c_transform:* A    @default(none);
    c_transform:* B    @default(none);

    // member HLSL function 'apply'
    #function(apply)
        float2 apply(float2 texcoord)
        {
            return B.apply(A.apply(texcoord));
        }
    #end
}
```

BUNGIE

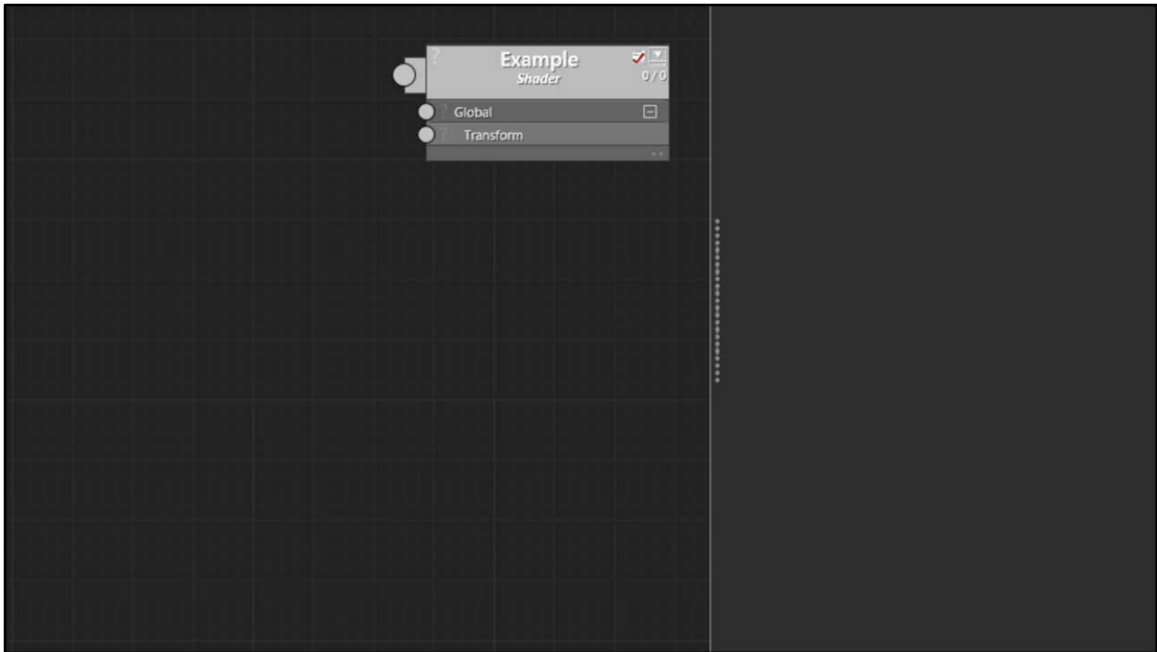
DESTINY

You can also use components from within *other* components.

This is a transform that combines two transforms, applying first A then B.

It declares two sub-components that can be connected to c\_transform nodes.

And this now lets you start building trees of nodes...



We can drag out the connection point, the editor shows a list of all matching transforms, and here is our combine node.

It has the two sub-components we expect, which we can now connect to other transforms.

Nothing crazy really, but the cool thing to me is that these nodes are defined entirely in *content*, which means, say, if an artist needs some new functionality, we can go in *LIVE*, at their machine, and start making new nodes for them; no need to kick new builds or send them new executables.

**(video3)**

## Connecting Components

```
component c_transform:combine
{
    // artist parameters - two transforms we want to combine
    // c_transform:* means any component using the
    //           c_transform interface
    // in the node graph, these show up as connection points
    // that can connect to c_transform nodes

    c_transform:* A    @default(none);
    c_transform:* B    @default(none);

    // member HLSL function 'apply'
    #function(apply)
        float2 apply(float2 texcoord)
        {
            return B.apply(A.apply(texcoord));
        }
    #end
}
```

BUNGIE

DESTINY

Going back to the component definition, we are calling component member functions, just as we would call functions on a class instance in C++.

But, this is not valid HLSL! HLSL doesn't support function names with . In them. How does this work?

## Name Translation

Component members *aren't* copy/pasted during splicing

Find and translate names to something unique

- Allow multiple instances of components

- Avoids collisions

- Resolves subcomponent members

BUNGIE

DESTINY 

For members of a component, we don't just copy/paste when we are splicing – we find and translate all of the member names to something unique.

This lets us instantiate the same component multiple times without collisions, as well as resolve the names of functions on sub-components appropriately.

## Name Translation

Component members *aren't* copy/pasted during splicing

Find and translate names to something unique

Allow multiple instances of components

Avoids collisions

Resolves subcomponent members

```
float2 scale;           → float2 _transform_scale;  
A.apply(texcoord);     → _transform_A_apply(texcoord);
```

BUNGIE

DESTINY 

So it will translate your parameters and member functions something like this.

We try to keep the names readable, they are prefixed with the names of their parents in the hierarchy,  
and if we run into any name collisions, or the names get too long, we start appending hashes to keep them unique.

## Component Override

```
import "main_vs.tfx"
...
c_transform:* g_transform @default(none) @ui_ignore;
#hlsl
..
#end

technique default
{
  ...
  override
  {
    g_transform= c_transform:rotate;
  }
  ...
}
```

BUNGIE

DESTINY

We also found it useful to allow our *techniques* to override components.

For example, here we are overriding that transform to be a rotation.

We have a little metadata tagged to the component to tell the node graph interface to ignore it, because we're going to stomp on whatever you select anyways.

Now this *is* a little bit of a contrived example, overriding the transform, generally we would use it for swapping base functionality, turning on debug modes in debug techniques, or things like that.

## Component Inheritance

```
// interface: c_transform
// variant: scale_offset

component c_transform:common
{
    // member parameters
    float2 scale @default(float2(1.0f, 1.0f));
    float2 center @default(float2(0.0f, 0.0f));
    float2 offset @default(float2(0.0f, 0.0f));

    ...
}

component c_transform:scale_offset
{
    inherit_from c_transform:common;

    ...
}
```

BUNGIE

DESTINY

We also added a **very** simple inheritance mechanism –  
And by simple, I mean it basically is just copy/pastes in all of the text from the component you inherit from.

If there are name collisions for members, whichever came first will win.

This is pretty simple, but it saves us in a ton of places where we would otherwise have to copy/paste that code around manually, duplicating work and increasing the cost of changing that code in the future.

## Destiny Component Stats

Standard opaque:

- 920 components
- 975 parameters



BUNGIE

DESTINY

We made extensive use of components in Destiny.

Our standard opaque shader defines almost 1000 components and parameters.



## Destiny Component Stats

Standard opaque:

- 920 components
- 975 parameters

Standard transparent:

- 1099 components
- 1411 parameters



BUNGIE

DESTINY

Our transparent shader uses slightly more than that,

## Destiny Component Stats

### Standard opaque:

- 920 components
- 975 parameters

### Standard transparent:

- 1099 components
- 1411 parameters

### Particle system:

- 1307 components
- 1775 parameters



BUNGIE


DESTINY

And our particle system uses a good bit more, around 1300 components and almost 1800 parameters.

## Node Graph-Facing Components

- texcoord 2D transforms
- RGBA values
- 3D transforms
- Higher-level shader options:
  - alpha\_test on/off
  - emissive on/off
  - Ambient occlusion override options
  - ...
- Noise functions
- Transparent lighting options
- Secondary vertex animation options
- Volumetrics options
- Subsurface options
- And much much more...

BUNGE

DESTINY 

A lot of these components are all of the node graph nodes, like texture transforms, generic RGBA values, as well as higher level options, like enabling alpha test, emissive, and things like that.

## Functional Components

- texture sampler states
- generalized texture fetch
- render target encode & decode (gbuffer, lighting, etc.)
- standard vertex types & interpolator declarations
- common material model functionality
- color space conversion components

BUNGIE

DESTINY 

But a large number of them are also functional components that generally aren't nodes – they're just functionality we've wrapped up so it can be reused in many locations.

These are things like texture sampler states, render target encode & decode for our passes, and our core material model implementation..



Now that we know how to create parameters and components, we want to talk about an important element of the TFX shader system – namely the mechanisms we have for optimizing parameter computation and for connecting parameters together

## Optimizing Parameter Authoring and Submission

- Artists want to have **semantically-intuitive parameters**

BUNGE

DESTINY 

Artist want to have semantically intuitive parameters

- Degrees instead of radians
- Position and angle instead of transform matrices
- Etc..

## Optimizing Parameter Authoring and Submission

- Artists want to have **semantically-intuitive parameters**
- May not be most CPU or GPU **performant representations**

BUNGIE

DESTINY 

These may not match most performant representations: GPU or CPU

## Optimizing Parameter Authoring and Submission

- Artists want to have **semantically-intuitive parameters**
- May not be most CPU or GPU **performant representations**
- **Keep the intuitive interface**
- Yet **optimize parameters' computation** under the hood

BUNGE

DESTINY 

Keep the intuitive interface but optimize parameters under the hood: Pull out shared operations to happen before shader or drawcall



## Optimizing Parameter Authoring and Submission

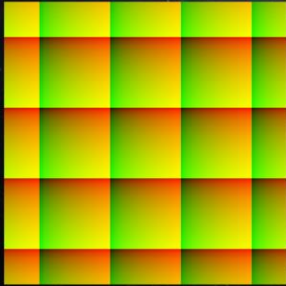
- Artists want to have **semantically-intuitive parameters**
- May not be most CPU or GPU **performant representations**
- **Keep the intuitive interface**
- Yet **optimize parameters' computation** under the hood
  
- TFX provides **automatic mechanisms** for this

BUNGE

DESTINY 

We built a system in TFX that provides a mechanism for managing that automatically

## Optimizing Parameters



```
// artist parameters for scale/offset transform
float2 scale      @default(float2(1.0f, 1.0f));
float2 center     @default(float2(0.5f, 0.5f));
float2 offset     @default(float2(0.0f, 0.0f));

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale,
        offset + center - center * scale);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

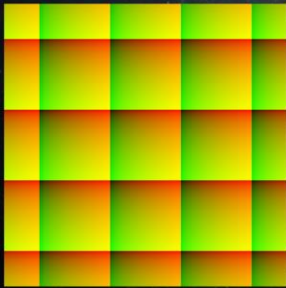
BUNGIE

DESTINY

Take a look at the operation highlighted here – it is doing math on purely uniform values – that is: every pixel will be computing the same value over and over again.

Ideally we want to do this computation once per draw call, instead of once per pixel.

## Optimizing Parameters



```
// artist parameters for scale/offset transform
```

```
float2 scale    @default(float2(1.0f, 1.0f));  
float2 center   @default(float2(0.5f, 0.5f));  
float2 offset   @default(float2(0.0f, 0.0f));
```

```
#hls1  
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0  
{  
    float4 xform= float4(  
        scale,  
        offset + center - center * scale);  
  
    float2 transformed_texcoord=  
        frac(ps_in.texcoord * xform.xy + xform.zw);  
  
    return float4(transformed_texcoord, 0.0f, 1.0f);  
}  
#end
```

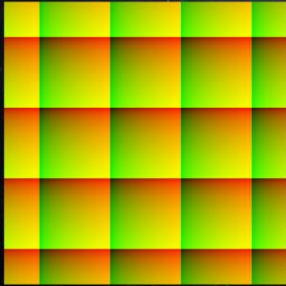
BUNGIE

DESTINY

But we also want to keep the sensible and intuitive artist parameters we had exposed before;

the artists wouldn't like having to edit the raw float4 xform directly

## Optimizing Parameters



```
// artist parameters for scale/offset transform
```

```
float2 scale    @default(float2(1.0f, 1.0f));  
float2 center   @default(float2(0.5f, 0.5f));  
float2 offset   @default(float2(0.0f, 0.0f));
```

```
float4 xform= float4(  
    scale,  
    offset + center - center * scale);
```

```
#hls1  
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0  
{  
    float2 transformed_texcoord=  
        frac(ps_in.texcoord * xform.xy + xform.zw);  
  
    return float4(transformed_texcoord, 0.0f, 1.0f);  
}  
#end
```

BUNGIE

DESTINY

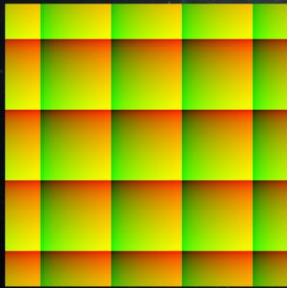
Well, what if we could just move the xform statement outside of the HLSL block?

This is now defining a new parameter, called 'xform', with a hard ASSIGNMENT.

Assigned parameters are NOT presented in the shader node graph interface, instead they use the value of the expression that is assigned to them.

These expressions can make use of other parameters we have defined, like center.

## Optimizing Parameters



```
// artist parameters for scale/offset transform
```

```
float2 scale    @default(float2(1.0f, 1.0f));  
float2 center   @default(float2(0.5f, 0.5f));  
float2 offset   @default(float2(0.0f, 0.0f));
```

```
float4 xform= float4(  
    scale,  
    offset + center - center * scale);
```

```
#hls1  
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0  
{  
    float2 transformed_texcoord=  
        frac(ps_in.texcoord * xform.xy + xform.zw);  
  
    return float4(transformed_texcoord, 0.0f, 1.0f);  
}  
#end
```

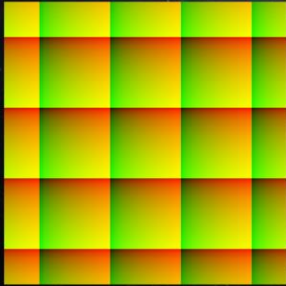
BUNGIE

DESTINY

In other shader systems we've worked with, optimizations like this, that pack or pre-compute values, had to be manually added to either the shader baking code, or the runtime submit code.

But here it is as easy as modifying the parameter assignments in the file.

## Optimizing Parameters



```
// artist parameters for scale/offset transform
float2 scale      @default(float2(1.0f, 1.0f));
float2 center     @default(float2(0.5f, 0.5f));
float2 offset     @default(float2(0.0f, 0.0f));

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float4 xform= float4(
        scale.x,
        offset.x + center.x - center.x * scale.x);

    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

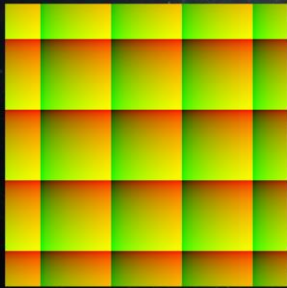
    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

This is really neat because... in moving that one line of code, we've effectively moved the operation from the GPU ...

## Optimizing Parameters



```
// artist parameters for scale/offset transform
float2 scale      @default(float2(1.0f, 1.0f));
float2 center     @default(float2(0.5f, 0.5f));
float2 offset     @default(float2(0.0f, 0.0f));

float4 xform = float4(
    scale,
    offset + center - center * scale);

#hlsl
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_lexcoord =
        frac(ps_in.texcoord * xform.xy + xform.zw);
    return float4(transformed_lexcoord, 0.0f, 1.0f);
}
#end
```

BUNGIE

DESTINY

to the CPU, yielding better GPU performance. FTW!

Alternatively – you could force the artist to precompute these values ahead of time but that’s painful (and they won’t like you for it!)

This mechanism allows us to do dynamic values at runtime efficiently in artist friendly user interface.

## TFX Expressions

```
// artist parameters for scale/offset transform
float2 scale      @default(float2(1.0f, 1.0f));
float2 center     @default(float2(0.5f, 0.5f));
float2 offset     @default(float2(0.0f, 0.0f));

float4 xform= float4(
    scale,
    offset + center - center * scale);

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * xform.xy + xform.zw);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end
```

BUNGE

DESTINY

The way this works is that TFX understands a fairly good subset of HLSL expression syntax, and can evaluate these expressions at bake time, or at runtime if necessary.

This is really neat, because we can use this mechanism not only do to parameter optimizations like these, but also as a flexible mechanism to link various data sources together..



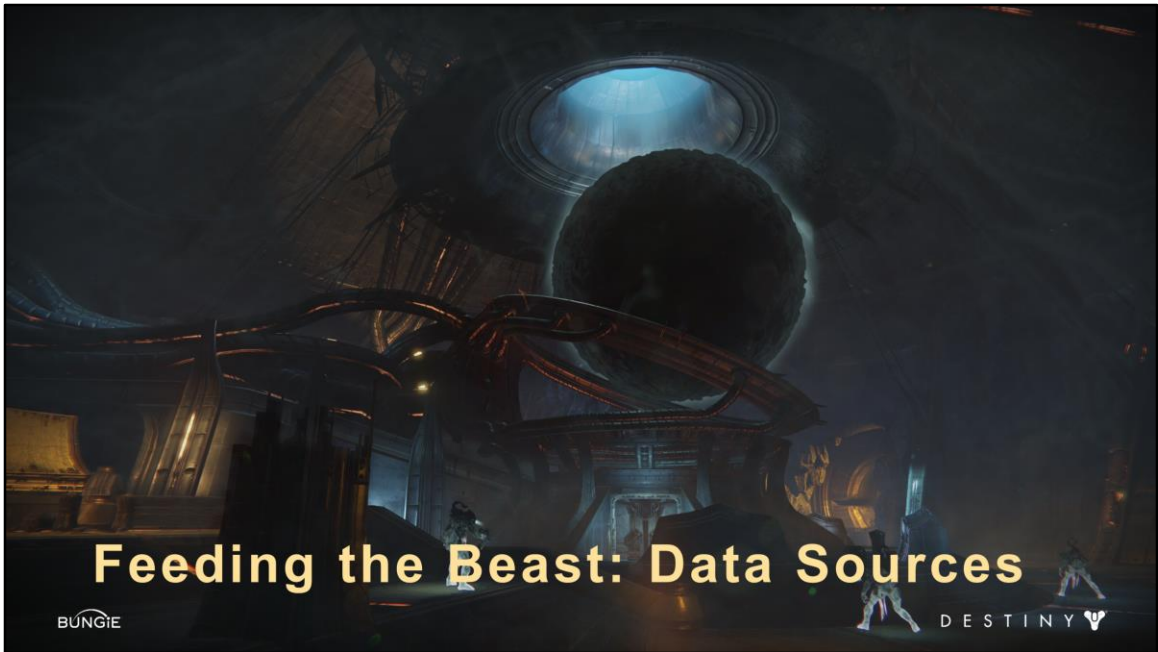
## TFX Expressions

- Basic HLSL expressions:
  - + - \* /
  - min, max, negate, and, or, not
  - sin, cos, atan, exp2, log2
  - floor, ceil, round
  - clamp, saturate, abs, sign, frac
  - dot, length, normalize,
  - lerp, step, smoothstep
  - Permutations and concatenations:
    - float4(a.xy, b.zw)
- Additional functionality:
  - Piecewise cubic spline eval
  - Custom functions:
    - triangle wave
    - jitter, wander
    - square\_random
    - smooth\_random
  - Helper functions
    - build rotation matrix
    - gradient interpolation

BUNGE

DESTINY 

And there a lot of expressions we support such as for example here  
But this leads us to ...



Now let's talk about how we actually put data into our shader engine: what data sources we use for generating values for our GPU state

## Shader Data Sources

- **Content sources**
  - Artist or programmer authored
    - Expressions, constants, curves, textures
  - From TFX Source, or Node Graph

BUNGIE

DESTINY 

There are a number of different data sources we want to use for GPU state.

First we have data that is provided by content.

This is constants, animation curves, expressions, and textures that are specified in the TFX source directly or by artists in a node graph.

## Shader Data Sources

- Content sources
  - Artist or programmer authored
    - Expressions, constants, curves, textures
  - From TFX Source, or Node Graph
- **Dynamic sources**
  - Bound by the engine @ runtime
  - Render target textures
  - Engine values (game\_time, camera matrix)
  - Object values (health, velocity)

BUNGIE

DESTINY 

Then we have the 'dynamic' sources, that are only available at runtime.


For example, render target textures, or engine values like game\_time, or object values like 'health' or 'velocity'.

TFX gives you the power to combine all of these arbitrarily controlled from content or code, which is really powerful!.

...

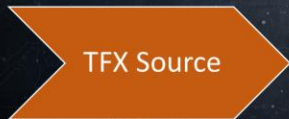
## TFX Data Flow From Content to Runtime

BUNGE

DESTINY 


Let's take a look at how the data flows through the TFX system all the way from content generation to runtime.

## TFX Data Flow From Content to Runtime



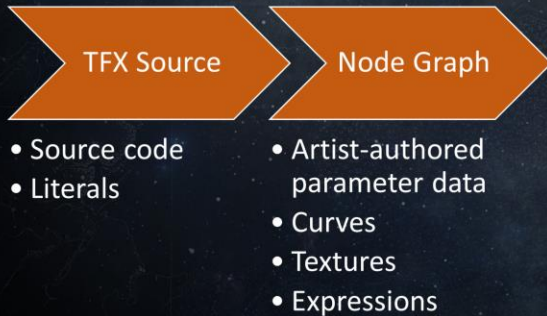
- Source code
- Literals

BUNGIE

DESTINY 

In content we specify data via source code (turning it into literals), ...

## TFX Data Flow From Content to Runtime

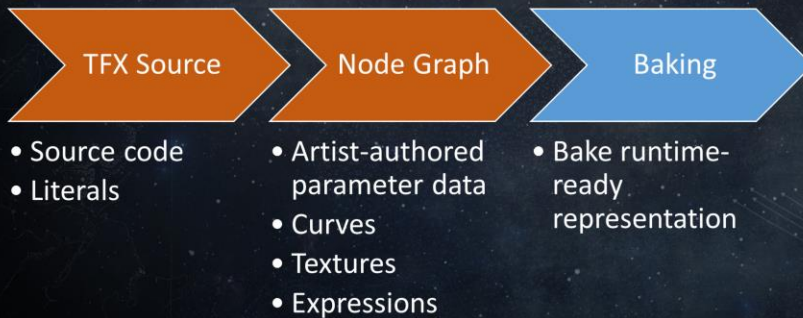


BUNGIE

DESTINY 

...then in **the node graph editor**, artists author parameter data which turns into curves, textures, and expressions, along with parameter values.

## TFX Data Flow From Content to Runtime



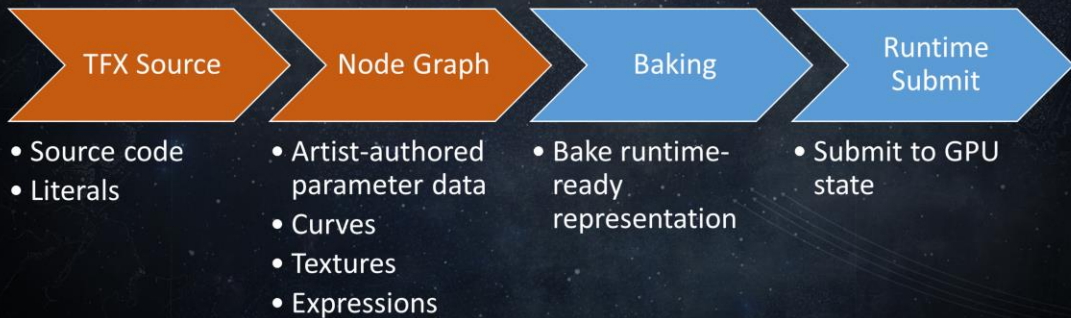
BUNGIE

DESTINY 

**At bake process**, we convert these into runtime ready representations – generate texture slot descriptors, fill out constant buffers, and anything else necessary for a given platform to store the state.



## TFX Data Flow From Content to Runtime

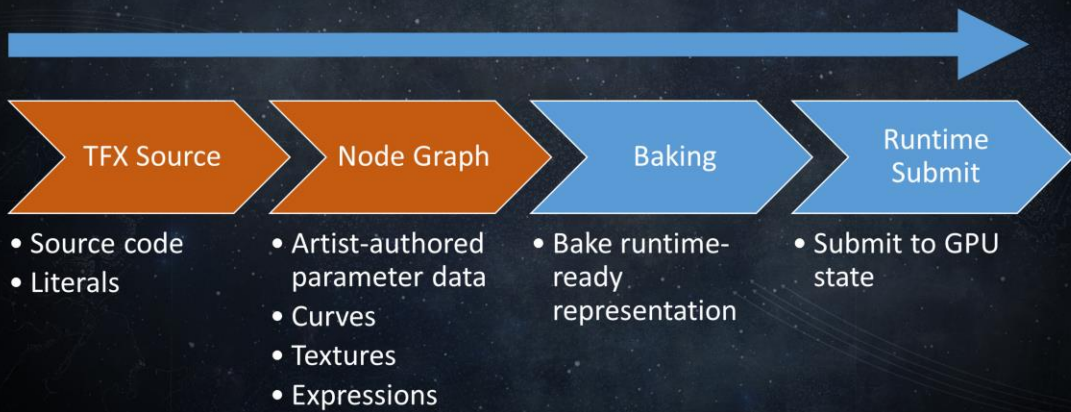


BUNGIE

DESTINY 

Finally, at the last stage at **runtime**, we convert this state to GPU-ready format filling out the registers, binding resource slots and so forth.

## TFX Data Flow From Content to Runtime



BUNGIE

DESTINY

So the **flow** is always from content to runtime like so.

## TFX Dynamic Data Sources



BUNGIE

DESTINY

Dynamic data sources are computed entirely at runtime and submitted by the engine to the corresponding GPU state through TFX mechanisms. Let's look those.

## TFX Dynamic Data Sources

- Externs
- Object channels
- Global channels

BUNGIE

DESTINY 

TFX has three ways to access dynamic data from the engine.

The three methods are:

- Externs
- Object Channels
- and Global Channels

## TFX Dynamic Data Sources

- Externs
  - Engine-provided runtime data
  - Tightly bound to C++ code (modifying requires recompile)
- Object channels
- Global channels

BUNGE

DESTINY 

Externs are used for engine-provided runtime data; these are tightly bound to the C code, any change requires a recompile of the engine and rebake of the affected shaders.

...



The recent armor sets made use of the `time_of_day` externs provided by the engine at runtime to animate glowing sections on the armor at night.

Here is what the armor looks like during the day



And when it's night time. Note that we automatically generate glow in pre-specified regions (based on a mask) based on the time of day extern.

## TFX Dynamic Data Sources

- Externs
- Object channels
  - Generalized per-object values
  - Controlled by content, script, or C code
- Global channels

BUNGIE

DESTINY 

Object channels are a system that provides generalized per-object values, that can be controlled by content, script, or C code.

There are a lot of object systems that will write data to channels, making it easy to source data about the object for driving GPU state.

In addition, artists can define new channels on objects from within content. For example, they can create an animated channel that synchronizes an effect across different shaders or different render systems.

...





Chroma armor makes use of object channels that control the color tint which can change based on the specific object's property, for example, glowing blue in the armor here.

## TFX Dynamic Data Sources

- Externs
- Object channels
- Global channels
  - 'Global' values
  - Controlled by content, script, or C code

BUNGIE

DESTINY 

Global channels are a similar system, but they aren't tied to an object, they are effectively 'global' data that can be content controlled. You can think of it as channels that are driving by global systems (for example, atmosphere).

## Data Sources: Global Channels



Global channels were often used for global effects:

- Autoexposure controls
- Sun direction / color / halo
- Global ambient lighting controls
- Atmospheric fog controls

They were most often used by sky / time-of-day artists but can be also used in any shaders. Here you see an example, where the sky shaders were tied to time of day and sun direction global channels for Mars sky rendering.

## Linking Data: Dynamic Expressions

- Expressions can use externs and/or channels

```
float my_value= sin(extern(game_time)) * 0.5f + 0.5f;  
float obj_velocity= channel(velocity);
```

BUNGIE

DESTINY 

All of these data sources can be linked together using expressions. These expressions can use externs and/or channels – for example here we have an **expression** that's dependent on a game time extern, and another expression which allows us to link this object's **velocity** channel to a shader parameter.

## Linking Data: Dynamic Expressions

- Expressions can use externs and/or channels

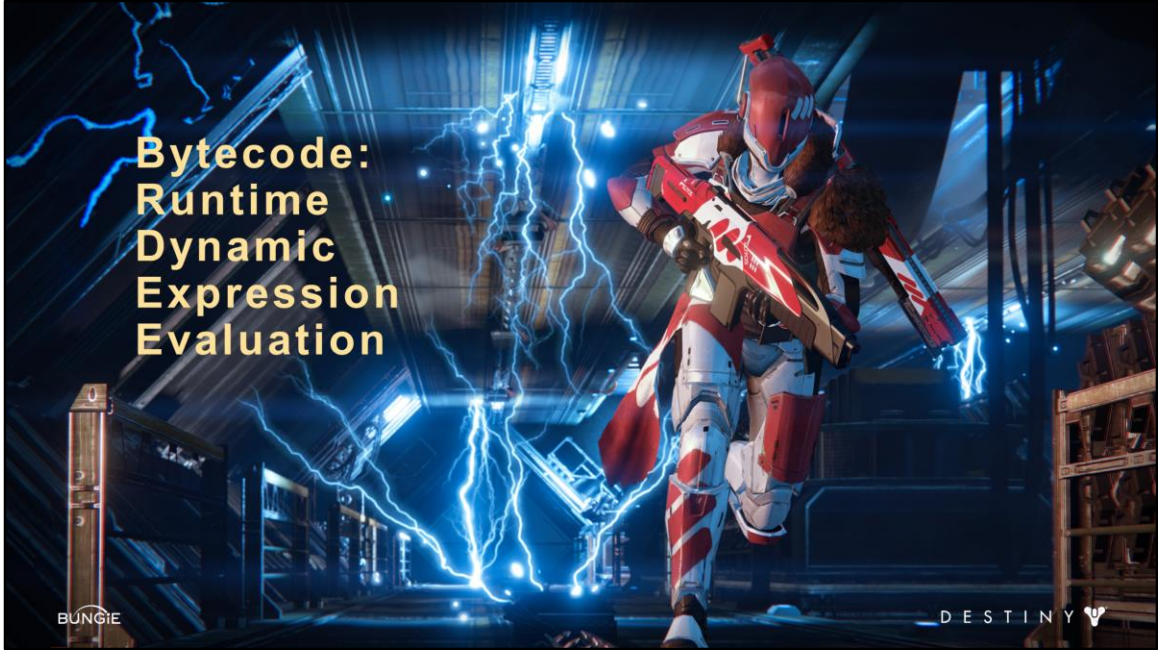
```
float my_value= sin(extern(game_time)) * 0.5f + 0.5f;  
float obj_velocity= channel(velocity);
```

- But these expressions are now DYNAMIC
  - Must be evaluated at runtime
  - And efficiently!
  - How?

BUNGIE

DESTINY 

Linking expressions to these data sources makes them dynamic because the data sources themselves are dynamic. This means that they must be evaluated at runtime. Naturally, we want this operation to be performant. How do we do it?



The answer is Bytecode!

## Why Bytecode?

Expressions are very similar to script

'code' defined by content

standard solutions:

Bytecode interpreter

JIT

Compile to bytecode

Interpret bytecode at runtime

Write output to GPU

BUNGIE

DESTINY 

Why bytecode?

It's the standard way to solve the problem of having your 'code' defined in content, which is exactly what we have.

The other option is JIT, which we considered, but would be much more complex to build, and potentially runs into code signing issues on console platforms.

So we keep it simple and compile our expressions to bytecode, and interpret it at runtime, writing the output to the GPU.

## TFX Bytecode

- **Stack-based**
  - Operations modify top of stack
  - 128-bit stack elements (generally SIMD 4 x float32)
- **Lean and mean**
  - Ops are 1-3 bytes
  - Separate data buffers referenced by bytecode
    - constants & references
  - Branchless (parallelizable)

BUNGIE

DESTINY

TFX bytecode uses a simple stack-based interpreter, where each operation modifies the top of the stack.

The stack contains 128-bit elements, which are generally SIMD 4 float values.

And most of the operations use SIMD vectorization on each platform.

It is a fairly lean implementation of bytecode – the ops are between 1 and 3 bytes, we separate the constant data into separate buffers, And it doesn't support any flow control or branches, the idea being we could parallelize execution for things like particle systems, where we may want to go wide across the particles.



## Building Bytecode: Inputs

1. A set of **expressions** (string or expression-tree form):

Ex: "value= `lerp(0.0, 0.4, saturate(cos(game\_time\*1.0+0.0))\*0.5+0.5)`"

2. A description of the **target output layout(s)**:

Ex: "value" should be put at cbuffer offset 8

Ex: "my\_texture" should be set to pixel shader register 3

BUNGIE

DESTINY

Input to the bytecode compiler is quite simple. It consists of

1. A set of expressions (string or expression-tree form):

value= "lerp(0.0, 0.4,  
saturate(cos(game\_time\*1.0+0.0))\*0.5+0.5)"

2. A description of the target output layout(s):

1. "value" should be put at cbuffer offset 8

2. "my\_texture" should be set to pixel shader register 3

## Compiled Bytecode Example

```
52,0  push_const_vec4(0)
60,1,0 push_extern_input_float
      s_tfx_externs_frame+0
31    cosine
52,1  push_const_vec4(1)
52,2  push_const_vec4(2)
18    multiply_add
35    saturate
3     multiply
77,0  push_object_channel_vector(0)
34,0  permute (.xxxx)
35    saturate
34,0  permute (.xxxx)
55,3  spline4_const 3
35    saturate
3     multiply
70,0  pop_temp(0)
69,0  push_temp(0)
52,8  push_const_vec4 8
69,0  push_temp(0)
3     multiply
29    negate
12    merge_1_3
67,0  pop_output(0)
```

BUNGIE

DESTINY 

When TFX compiles this to bytecode, here is the output byte stream, and it's translation.

Currently all the commands are either one or two bytes.

We do a quick optimization pass, but it's not great. Luckily the bytecode eval has never been a huge bottleneck,

## Compile Optimizations

- Evaluate static sub-expressions, pre-fill static outputs

`game_time * 2.0 * 0.3` → `game_time * 0.6`

- Save & reuse common sub-expressions to scratch buffer

BUNGIE

DESTINY

We did add a handful of optimizations while compiling bytecode.

We pre-evaluate all of the static sub-expressions, and any purely static outputs are handled in native code.

We also save & reuse common sub-expressions to a scratch buffer, if it reduces overall bytecode cost.

## Bytecode Interpreter

- SIMD CPU Interpreter (SSE / VMX / SPU)

BUNGIE

DESTINY 

We actually have a few different bytecode interpreters:

- We have SIMD CPU interpreters for each platform, making use of the native SIMD instructions.

## Bytecode Interpreter

- SIMD CPU Interpreter (SSE / VMX / SPU)
- GPU Interpreter (editing only)
  - quick GPU particles iteration by replacing bytecode

BUNGIE

DESTINY 

- We also have a GPU interpreter. This is a fairly expensive path (the shader that implements this is over 4000 ALUs), but is useful for doing quick iteration on GPU particles in editor, by simply updating bytecode on the fly. Then, in released builds, we..

## Bytecode Interpreter

- SIMD CPU Interpreter (SSE / VMX / SPU)
- GPU Interpreter (editing only)
  - quick GPU particles iteration by replacing bytecode
- Generate HLSL from bytecode

BUNGE

DESTINY 

...convert that bytecode into HLSL, for native evaluation to the GPU.

## Bytecode Interpreter

- SIMD CPU Interpreter (SSE / VMX / SPU)
- GPU Interpreter (editing only)
  - quick GPU particles iteration by replacing bytecode
- Generate HLSL from bytecode

### Runtime optimization:

- Interpreter stack caching (keep top of stack in a register)
  - [M.A. Ertl. Stack Caching for Interpreters.]

The only runtime optimization we did for the CPU interpreter was stack caching – which is where we keep the top of the stack in a register, and there's a link to a paper on that.




So, now that we know what to put into GPU state, let's organize it.  
let's say we want to render a scene – at the lowest level, we are issuing a number of draw calls to the GPU, and we want to make sure that for each draw call, the GPU is in the desired state for that draw.



## Drawcall / Dispatch Shader GPU State

- Shaders (pixel, vertex, compute, ...)

BUNGE

DESTINY 

When we look at any given draw or dispatch that our engine needs to execute, we see that the GPU state consists of:

- The shaders themselves (vertex, pixel, compute, etc.)

## Drawcall / Dispatch Shader GPU State

- Shaders (pixel, vertex, compute, ...)
- GPU state for each shader stage:
  - Mesh data (vertex & index buffers)
  - Constant Buffers
  - Textures
  - Samplers
  - UAVs / Buffers
  - Render States
  - Render Targets (output)
  - Blend Modes

BUNGE

DESTINY 

- Any GPU resources namely
  - Mesh data
  - Constant Buffers
  - Textures
  - Samplers
  - UAVs / Buffers
  - Render States
  - Render Targets (output)
  - Blend Modes

## GPU State Control

- Global states are handled by the engine layer
  - Ex: render targets, viewports, scissor rects
- TFX encapsulates
  - Constant buffers
  - Textures / buffers / UAVs / samplers
  - Render states

BUNGE

DESTINY 

Render Targets, Viewports and Scissor Rects are 100% controlled by C++ engine runtime, TFX does not encapsulate control for that.

TFX provides encapsulation for constant buffers, textures / general buffers or UAVs and sampler setup, and render / blend states

## GPU State Control Priority

- Three-layer priority:
  - 1) C++ specifies default states
  - 2) Technique can override default
  - 3) C++ can override technique
- TFX components or techniques can supply state

BUNGE

DESTINY 

For controlling GPU state, we use the following scheme of three-layer priority:

- Our C++ supplies default states. This is typically set before any technique is to be rendered to ensure there aren't any missing states (and avoid tripping up low-level API validation layers).
- Each technique can optionally override the default states with encapsulated states. Note that we can specify state in the TFX components or in the technique itself
- Lastly, even after the technique is activated, we can still override the state in C++ in the runtime engine. This is particularly useful when forcing some behavior or when implementing debug or validation modes

## Organizing GPU State with Scopes

- A 'Scope' is a set of related GPU state
  - Encapsulates description of resources *plus* their destination
    - Resource type (CB, texture, ...\_)
    - Register to bind this resource to

BUNGIE

DESTINY 

We are going to organize this GPU state using the concept of Scopes.

A scope is a set of related GPU state – it encapsulates the description of resources as well as their destination. We specify which constant buffer location we bind each constants to, for example, or which texture slots to bind textures to. This is used to both build the underlying resources under the hood by TFX at runtime, as well as ensure that we don't stomp on the state (more on that later).

## Organizing GPU State with Scopes

- A 'Scope' is a set of related GPU state
  - Encapsulates description of resources *plus* their destination
    - Resource type (CB, texture, ...\_)
    - Register to bind this resource to
- Resource descriptions can vary per shader stage and platform

BUNGIE

DESTINY 

Note that we actually can describe resources in the scope separately for each shader stage and platform. This is needed so that we can manage the slots independently – we might not have identical texture slots free, for example, on Xbox 360 as we might on PlayStation 4.



Let's take decals as an example, and look at the GPU state when we go to draw a decal.

## GPU State for a Decal Technique

### Constants

- material texture transforms
- matrices
- game\_time

### Textures

- material textures
- depth/normal render targets

### Samplers

- common or custom

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	aniso_wrap
mesh_to_world_matrix	base_normal	trilinear_wrap
depth_constants	detail_normal	point_clamp
screen_texcoord_to_world	depth_buffer	
projection_matrix	normal_buffer	
game_time		

BUNGE

DESTINY

The GPU state consists of a number of float constants, like the material texture transforms, projection matrices, and perhaps an animation parameter like game\_time.

It also includes textures, like the decal material textures, as well as render target textures from the gbuffer pass, like depth and normal buffers.

And we'll need a few sampler states used to read those textures.



## Shared or Unique?

'Unique' state



'Shared' state



Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY 

First let's separate this into 'unique' state, specific to this decal, and 'shared' state, that may be used by a number of draws in the decal pass.

## Shared or Unique?

'Unique' state  
Stored by technique

'Shared' state

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGE

DESTINY

The unique state will be stored by the technique assigned to the decal,

## Shared or Unique?

'Unique' state  
Stored by technique

'Shared' state  
Move to scopes

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGE

DESTINY 

Whereas the shared state we will move into scopes.

## Shared or Unique?

'Unique' state  
Stored by technique

'Shared' state  
Move to scopes

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY 

Generally we will want to define the scopes based on similar lifetimes and frequency of submission.

## Determine Scopes

Split shared state by frequency of submission:

■ frame scope

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGE

DESTINY 

For example, 'game\_time', and the common sampler states, are applicable to the entire frame, so let's put them in a 'frame' scope.

## Determine Scopes

Split shared state by frequency of submission:

- frame scope
- view scope

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY 

The projection matrix is valid for the entire view, so assign it to the view scope.

## Determine Scopes

Split shared state by frequency of submission:

- frame scope
- view scope
- decal pass scope

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY

Then we can put the state specific to the decal pass into it's own scope.

## Determine Scopes

Split shared state by frequency of submission:

- frame scope
- view scope
- decal pass scope
- mesh instance scope

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY

The mesh\_to\_world transform matrix is really a special case here, in that it may be changed MORE frequently than the technique.

If we have multiple instances of the same decal, we can setup the technique once, then change only the mesh\_to\_world matrix and issue another draw call to draw the same decal in a different location.

So we'll put this in it's own scope, the mesh instance scope.



## Determine Scopes

Split shared state by frequency of submission:

- frame scope
- view scope
- decal pass scope
- mesh instance scope

Optimize for native submit

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGE

DESTINY

If we want to optimize the submission of this high frequency submitted state in tight inner loops, we have an option to supply C++ scope submission path instead of going through the bytecode submit for faster CPU.

## Unique State → Technique

Split shared state by frequency of submission:

- frame scope
- view scope
- decal pass scope
- mesh instance scope
- technique

Constants	Textures	Samplers
base_xform	base_texture	custom
detail_xform	detail_texture	
	base_normal	
	detail_normal	
mesh_to_world_matrix		
depth_constants		
screen_texcoord_to_world		aniso_wrap
projection_matrix	depth_buffer	trilinear_wrap
game_time	normal_buffer	point_clamp

BUNGIE

DESTINY

Then, all of the unique state belongs to the technique.

## Sharing State via Scopes

### Sharing GPU state has many benefits

- Improves CPU perf
- Improves GPU perf
- Less command buffer memory
- Less technique memory
- Lower bandwidth

BUNGE

DESTINY 

Sharing state this way is good!

- Less CPU perf
- Less GPU perf (less context rolls)
- Less command buffer memory
- Less technique memory
- Less bandwidth

## Scopes vs. Techniques

	Scopes	Techniques
Contain:	GPU state	GPU state & <b>shaders</b>

BUNGIE

DESTINY 

There are a few differences between scopes and techniques:

- 1) Scopes never contain shaders (scopes plug *into* shaders)

## Scopes vs. Techniques

	Scopes	Techniques
Contain:	GPU state	GPU state & shaders
Layout:	Explicitly declared	From compiled shader reflection

BUNGIE

DESTINY

The scope layout, that is the set of state that it handles, is explicitly declared, whereas techniques will just grab everything else, so we often let the shader compiler decide where to put the state.

## Scopes vs. Techniques

	Scopes	Techniques
Contain:	GPU state	GPU state & shaders
Layout:	Explicitly declared	From compiled shader reflection
Binds:	All declared state	All state not in a bound scope

BUNGIE

DESTINY 

As we said before, scopes will bind all of the state that they declare, whereas techniques get 'everything else'.

## Scopes vs. Techniques

	Scopes	Techniques
Contain:	GPU state	GPU state & shaders
Layout:	Explicitly declared	From compiled shader reflection
Binds:	All declared state	All state not in a bound scope
Submit:	TFX/bytecode or custom C++ code	Always TFX/bytecode

BUNGE

DESTINY 

And a final difference – techniques always run our TFX submit path (which may execute bytecode for dynamic values), whereas scopes have the option to use custom C++ code to directly submit their state.

## Scopes vs. Techniques

	Scopes	Techniques
Contain:	GPU state	GPU state & shaders
Layout:	Explicitly declared	From compiled shader reflection
Binds:	All declared state	All state not in a bound scope
Submit:	TFX/bytecode or custom C++ code	Always TFX/bytecode

Scopes can be used to optimize submission for high frequency inner loop state

BUNGIE

DESTINY 

This is useful for frequently changed state, like the mesh instance scope in the example earlier. In cases like that, if we want to optimize the submission of that state for high frequency inner loops, this lets us move the submission to native code.



## TFX Scope Declaration

```
scope decal_pass
{
    platform all:
    shader vertex, pixel:
        registers(texture, 0-1);
        cbuffer(11);

    texture_2D depth_buffer= extern(depth);
    texture_2D normal_buffer= extern(deferred_normal);

    float4x4 screen_texcoord_to_world=
        extern(target_texture_to_world);

    float4 depth_constants= extern(depth_constants);

    #hls1
    ...
    #end
};
```

BUNGIE

DESTINY 

The scopes are declared in the TFX language, So effectively, they can be changed easily since just are treated just like content

## TFX Scope Declaration

```
scope decal_pass
{
    platform all:
    shader vertex, pixel:
        registers(texture, 0-1);
        cbuffer(11);

    texture_2D depth_buffer= extern(depth);
    texture_2D normal_buffer= extern(deferred_normal);

    float4x4 screen_texcoord_to_world=
        extern(target_texture_to_world);

    float4 depth_constants= extern(depth_constants);

    #hls1
    ...
    #end
};
```

BUNGIE

DESTINY 

This is declaring the **'decal\_pass'** scope.

## TFX Scope Declaration

```
scope decal_pass
{
    platform all:
    shader vertex, pixel:
        registers(texture, 0-1);
        cbuffer(11);

    texture_2D depth_buffer= extern(depth);
    texture_2D normal_buffer= extern(deferred_normal);

    float4x4 screen_texcoord_to_world=
        extern(target_texture_to_world);

    float4 depth_constants= extern(depth_constants);

    #hls1
    ...
    #end
};
```

BUNGIE

DESTINY

First it declares all of the state that it wants to claim:

on **all platforms**,  
in **both** the vertex and pixel shader,  
we are reserving:  
- **texture** registers 0 through 1, and also **cbuffer** 11

## TFX Scope Declaration

```
scope decal_pass
{
    platform all:
    shader vertex, pixel:
        registers(texture, 0-1);
        cbuffer(11);

    texture_2D depth_buffer= extern(depth);
    texture_2D normal_buffer= extern(deferred_normal);

    float4x4 screen_texcoord_to_world=
        extern(target_texture_to_world);

    float4 depth_constants= extern(depth_constants);

    #hls1
    ...
    #end
};
```

BUNGIE

DESTINY

then, it declares what that state consists of:

- the first **texture** will be called 'depth\_buffer'
- the second **texture** will be called 'normal\_buffer'
- the first element of the **cbuffer** will be a 4x4 matrix called 'screen\_texcoord\_to\_world'
- and **so on and so forth**

## TFX Scope Declaration

```
scope decal_pass
{
    platform all:
    shader vertex, pixel:
        registers(texture, 0-1);
        cbuffer(11);

    texture_2D depth_buffer= extern(depth);
    texture_2D normal_buffer= extern(deferred_normal);

    float4x4 screen_texcoord_to_world=
        extern(target_texture_to_world);

    float4 depth_constants= extern(depth_constants);

    #hlsl
    ...
    #end
};
```

BUNGIE

DESTINY

These **expressions** on the right hand side of each declaration here will be run through our expression/bytecode system, just like for regular shaders.

## Beware of Stomping on GPU State

- You *can* declare overlapping scopes
  - Re-use state in different passes
- We need to ensure that scopes don't stomp on each other



BUNGE

DESTINY 

There is an important consideration – since scopes manage GPU state locations, they can easily write to the same destination. This can be ok, if that's what you intended, but if not, bad things can happen.

How do we ensure that there are no conflicts?



To solve this problem, we built the GPU state validation system

## Why Validate GPU State?

- Errors hard to detect & debug
- 2 Types of Errors:
  - Missing state (never set)
  - State stomps (register collisions)
- Both result in rendering artifacts

BUNGE

DESTINY 

Why would you want to validate GPU state?

State errors can be very difficult to detect and debug.

They generally fall into two categories: state that is missing, if no one ever set it, or state that WAS set, but then someone came along and stomped on it, because of register collisions.

Both of these result in rendering artifacts, which may not even be noticed for a long time



## Why Validate GPU State?

- Errors hard to detect & debug
- 2 Types of Errors:
  - Missing state (never set)
  - State stomps (register collisions)
- Both result in rendering artifacts
  - Or GPU crashes ☹ ☹ ☹
- Tracking every register is expensive...

BUNGIE

DESTINY

or in some cases, especially with some of the lower-level graphics APIs on consoles, they can result in GPU crashes, .. which are even more 'fun' to debug.

So some kind of validation to automatically catch these errors would be great.

We could track every active register, **but** that would be pretty expensive...

## Scopes to the Rescue!

There's only 23 scopes in Destiny total!

```
frame          view          rigid_model   editor_mesh   editor_terrain
skinning       speedtree     chunk_model   decal         instances
terrain        postprocess  transparent   ui_font       cui_hud
cui_view       cui_object   cui_bitmap    cui_standard
speedtree_instance_data  speedtree_lod_drawcall_data
sdsm_bias_and_scale_textures
transparent_advanced
```

BUNGIE

DESTINY


But hey, if all of our state is already organized into scopes, why don't we just track those scopes?

We only ended up with around 23 scopes total in Destiny, so we can track a bit-vector of them in a single DWORD.

## Scope Validation

Runtime submit context:  
Bit-vector of ACTIVE scopes

BUNGIE

DESTINY 

Basically, the way it works is we track each ACTIVE scope in a bit-vector in the runtime submit context.

## Scope Validation

Runtime submit context:  
Bit-vector of ACTIVE scopes

Scope:  
Bit-vector of COMPATIBLE scopes  
Validate scope → scope stomps

$(ACTIVE \ \& \ \sim COMPATIBLE) == 0$

BUNGE

DESTINY 

Then in each scope, we can store a bit-vector of the compatible scopes.  
That is, other scopes that have no overlapping state.

This lets us quickly validate that activating a **new** scope won't stomp any of the **existing** active scopes.

## Scope Validation

Runtime submit context:  
Bit-vector of ACTIVE scopes

Scope:  
Bit-vector of COMPATIBLE scopes  
Validate scope → scope stomps

$(ACTIVE \& \sim COMPATIBLE) == 0$

Technique:  
Bit-vectors of COMPATIBLE *and* USED scopes

- Validate technique → scope stomps
- Validate missing state

$(ACTIVE \& \sim COMPATIBLE) == 0$   
 $(USED \& \sim ACTIVE) == 0$

BUNGIE

DESTINY 

Then, in each technique, we can do the same thing, make sure everything is compatible..  
and we can **also** store a bit-vector of the scopes that the technique directly **USES**.

This lets us quickly validate that none of the required state is missing.

This kind of validation is extremely low overhead, and helped tremendously in catching errors in development.



Now that we have the shader code and the shader state sorted, how do we integrate it with the rest of the renderer?

## TFX Material Assignment

- Shader node graph is the material definition

BUNGIE

DESTINY 

In our system the shader node graph is the material definition.

## TFX Material Assignment

- Shader node graph is the material definition
- Materials can be assigned to any scene element

BUNGE

DESTINY 

Materials can be assigned to any element in a scene - Meshes / mesh parts; Whole objects; Particle systems; Screen passes (postprocessing, screen FX)



## TFX Material Assignment

- Shader node graph is the material definition
- Materials can be assigned to any scene element
- **Assignment defined by the client system**

BUNGE

DESTINY 

Assignment is actually entirely defined by the client system – TFX shader pipeline does not control or limit this in any way.

Assignment starts with a node graph TFX shader

At bake time, client system just requests a technique from the node graph

## Materials and Render Pass Relationship

- Node graph → one or more render passes
- Render pass == TFX technique

BUNGE

DESTINY 

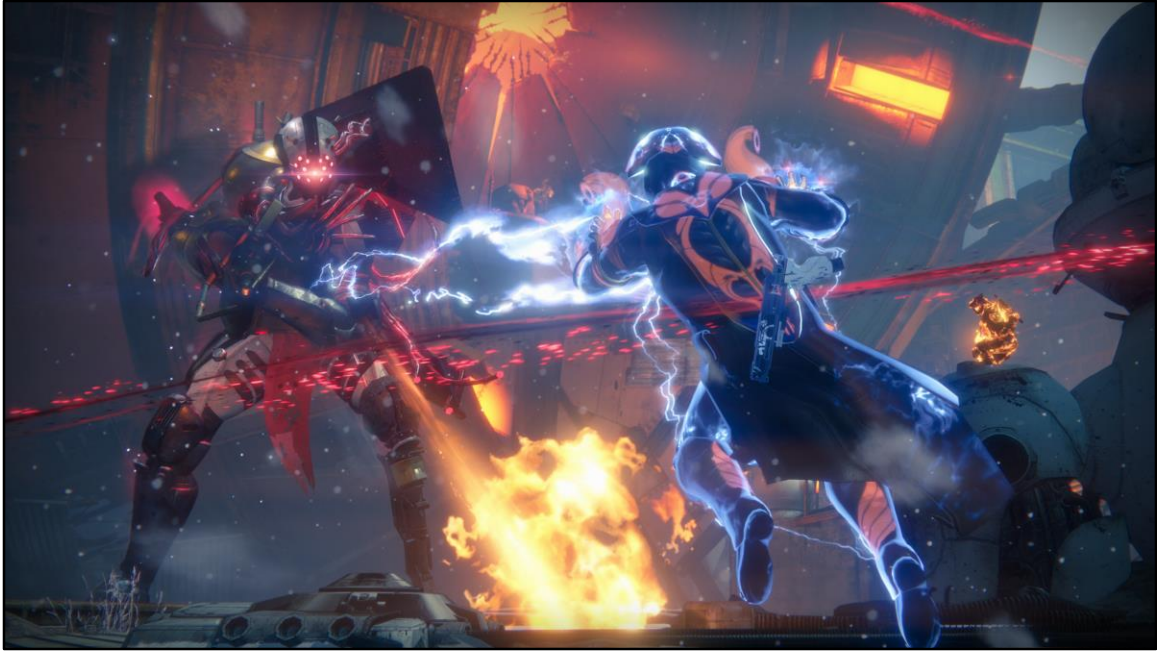
A shader node graph can generate one or more passes, where each pass corresponds to a technique which is defined in the source TFX shader



For example, in this scene, each opaque material generates custom Gbuffer, Depth-only, Shadow passes



Another example are these multi-layer decals which also can generate multiple render passes



GPU particle systems can have many passes (spawn / update / render layers)

Now that we've figured out how to assign materials, lets take a look at runtime submission...



If we look at breakdown of a frame, you'll quickly see that we build the frame from <a number of passes> (as you see in this list on the right). Note that these passes are different depending on the state of the game and the content visible.

## Submitting a Frame

- Set frame state
- For each view:
  - Set view state
  - For each render pass:
    - Set pass state
    - For each visible mesh
      - Set technique (shaders)
      - Draw!

BUNGIE

DESTINY 

At high level in pseudocode, each frame can be roughly broken down like so – we set global frame-specific GPU state, then for each view (for example, the main view, shadows views, reflection views, etc.) we setup that view's render state, then render passes in that view.

## Runtime TFX Submit Entry Points

BUNGIE

DESTINY 

For all the operations above we use only a few TFX entry points to control the runtime submission.



## Runtime TFX Submit Entry Points

- CPU-side data sources management  
`{ push | pop } externs`

BUNGIE

DESTINY 

We setup CPU-side data sources via push and pop operation for the externs. Pushing creates space in the corresponding resource for CPU to upload the data; popping just notifies that we no longer need that particular state.

## Runtime TFX Submit Entry Points

- CPU-side data sources management  
  { push | pop } externs  
  set { object | global } channels

BUNGIE

DESTINY 

For channels we simply set them using the corresponding set entry points (for global or object channels). Set is used purely for copying resource data.

## Runtime TFX Submit Entry Points

- CPU-side data sources management
  - `{ push | pop } externs`
  - `set { object | global } channels`
- GPU-side data management
  - `{ activate | deactivate } scope`

BUNGIE

DESTINY 

GPU state is controlled via activation and deactivation of scopes, which executes bytecode for any dynamic expressions and copies resultant data into appropriate GPU resources (like constant buffers), ...

## Runtime TFX Submit Entry Points


- CPU-side data sources management
  - `{ push | pop } externs`
  - `set { object | global } channels`
- GPU-side data management
  - `{ activate | deactivate } scope`
- Runtime shader entry points:
  - `activate technique`

BUNGE

DESTINY 

...and we activate the technique to set its shaders and setup its local state (and also execute any local state's dynamic expressions using the bytecode interpreter)

## TFX Frame Submit Control

- Set frame state
  - For each view:
    - Set view state
    - For each render pass:
      - Set pass state
      - For each visible mesh
        - Set technique (shaders)
        - Draw!
- Push externs
  - Activate scopes
- 

BUNGIE

DESTINY 

We setup the externs and activate scopes at these phases. Note that we want to move any shared state as high up the chain as possible to avoid submitting it multiple times per frame

## TFX Frame Submit Control

- Set frame state
- For each view:
  - Set view state
  - For each render pass:
    - Set pass state
    - For each visible mesh
      - Set technique (shaders)
      - Draw!


BUNGE

DESTINY 

We pop externs and deactivate scopes to notify our GPU state tracking that the respective state is no longer needed after each loop is finished

# TFX Submission and Render Multithreading

BUNGIE

DESTINY 

One aspect we'll touch on briefly is how we had to extend our shader pipeline to support fully multi-threaded submission.

## TFX and Render Job Multithreading

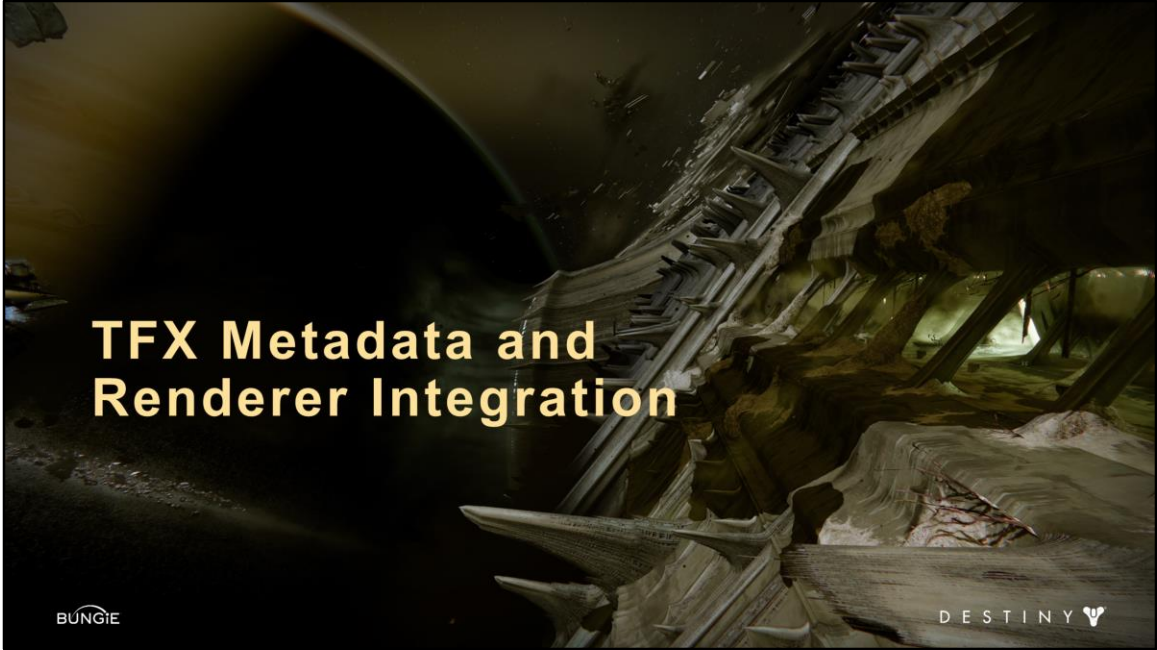
- Our renderer executes entirely in jobs (see [[GDC2015](#)] for more)
- Each job submitted to deferred context
  - But we relied of previously set state for correctness
  - Some platforms required full state setup for each job
- TFX GPU state tracking allowed per-job correct state setup
  - Based on scope and extern tracking
- Ensured each job was setup correctly automatically
  - Without overly redundant state setup

BUNGIE

DESTINY 

As mentioned in the GDC 2015's talk (<http://www.gdcvault.com/play/1021926/Destiny-s-Multithreaded-Rendering>), Destiny renderer is run entirely from jobs. Each job submits to a flavor of deferred context. However, on some platforms, each job had to have the entirety of the full previously set state setup in the job itself to run correctly. But remember that we were controlling state submission frequency at higher level. To ensure that we did this correctly, we relied on the main TFX tracking for GPU state using the scope and extern tracking. Each job would enter and setup all the missing scope based on the required globally set scopes and externs and copied state from the parent job. The encapsulation of GPU state in tracked TFX scopes helps a great deal with ensuring correct multi-threading state setup for render jobs. On each render jobs, we check which scopes the system expects to be set and reset for that deferred context. This keeps render jobs always in sync automatically (another nice advantage of the scope tracking). Even though the state was originally controlled from a previous job or even serialized code





Next let's touch base on a mechanism for extending our TFX system for variety of useful subsystems

## Flexible TFX Metadata System

Anything starting with @ is metadata

Metadata used for:

- UI controls
- Default values
- Render stage markup
- Preprocessor definitions / labels
- Splicing and compilation directives
- Client directives

Highly extensible

- Can add new metadata at any time

BUNGIE

DESTINY 

Anything in TFX that started with @ sign is treated as metadata markup. We use it extensively for a number of markup:


- UI controls
- Default value specification
- Render stage markup
- Preprocessor definitions / labels
- Optional splicing or compilation directives
- Client directives

One of the advantages of TFX metadata was that it was a highly extensible system, and we basically added new use cases as we found them.

## Metadata for UI Control

- Metadata markup drives Node Graph UI behaviors
- Builds UI using parameter metadata
- Provides:
  - UI description: name, help text
  - UI controls: slider, color picker, min/max values
  - UI organization: grouping, sorting, collapsing

BUNGIE


DESTINY 

We provided a number of metadata to markup shader parameters to specify UI behavior, which drove the automatic generation of node graph UI. We supported the basics (UI name, description, and semantic UI controls: color, slider, min / max values) but also UI organization (hiding parameters from artist interfaces, allow grouping shader parameters into UI groups and sorting them in arbitrary order within each group, we also support markup for marking groups as collapsed or open for better UI organization)

## Metadata for UI Control

- Allows fine-tuning artist interface
  - Drive toward intuitive and easy to use interfaces
  - Decouple shader parameters / component organization from UI

BUNGIE

DESTINY 

This allowed us to decouple shader parameters in code and the UI for the artist (i.e. component organization from UI exposure), which gave us power to further fine-tune artist interface to make the node graph interfaces more intuitive to the artists

Another use of TFX meta data was our render stage management...

## Render Stage Mechanism

- In depth covered in [\[GDC2015\]](#) talk
- Render stage is our runtime submission management
  - How the mesh decides which view / pass it renders in
  - Which technique to render this mesh with

DESTINY 

Render stage is our mechanism for filtering runtime submission for passes and views. We actually cover the deep details of the render stage mechanism in the GDC 2015 but we'll touch on a couple elements here related to TFX language... At its core it is about shader management (selecting the right techniques at the right time) and filtering the visible list in a given view (which comes down to mesh filtering for actual drawcall generation using the right shader we've just selected). Of course we want it to be automatic and transparent to leaf feature writers.

## Metadata for Render Stage

Flag each technique in TFX source with metadata:

```
@render_stage(...)
```

Client mesh queries metadata at bake time  
to determine render stage

DESTINY 

This is a Data-driven mechanism on the shader side; the runtime engine declares the render stages that it supports (corresponding to render passes), and then render stage is specified in the TFX shader source, tagged to each technique.

At bake time, we query the metadata from the technique to determine the render stage, and assign the corresponding mesh to that render stage automatically.

## Render Stage Metadata

```
import "main_vs.tfx"
import "interpolators.tfx"

#hls1
float4 main_ps(s_ps_in ps_in) : TFX_TARGET0
{
    float2 transformed_texcoord=
        frac(ps_in.texcoord * 4.0f + 0.5f);

    return float4(transformed_texcoord, 0.0f, 1.0f);
}
#end

technique my_technique
{
    @render_stage(gbuffer)

    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}
```

BUNGIE

DESTINY

Shader metadata specifies render stage in shader code on the technique level. Here is the example of the render stage markup in our original simple shader – we simply add the render stage metadata to our technique.

## Multiple Render Stages per Node Graph

```
technique default
{
    @render_stage(gbuffer)

    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);
}

technique shadow_pass
{
    @render_stage(shadows)

    compile_shader(all_platforms, vs, shadow_vs);
}

technique depth_prepass
{
    @render_stage(depth_prepass)

    compile_shader(all_platforms, vs, depth_prepass_vs);
}
```

We can also specify multiple render stages for each technique in the material. In fact each of our standard opaque materials support these stages by default.

Next, let's touch on another use of metadata for TFX labels...



## TFX Labels Metadata

Bake-time queryable properties of shaders or components

Components supply any number of labels

Client systems query label presence to drive local decisions

BUNGIE

DESTINY 

TFX Labels provide a generic mechanism for querying material properties at bake time.

Each component can supply any number of labels, and the client systems can query those labels to make local bake-time decisions in a data-driven manner, dependent on the content of the components in the shader.

We used labels for a bunch of different applications..

## Driving Visibility from Labels

Destiny visibility is based on Umbra  
Required visibility occluders specification

Certain materials should never occlude visibility

BUNGE

DESTINY 

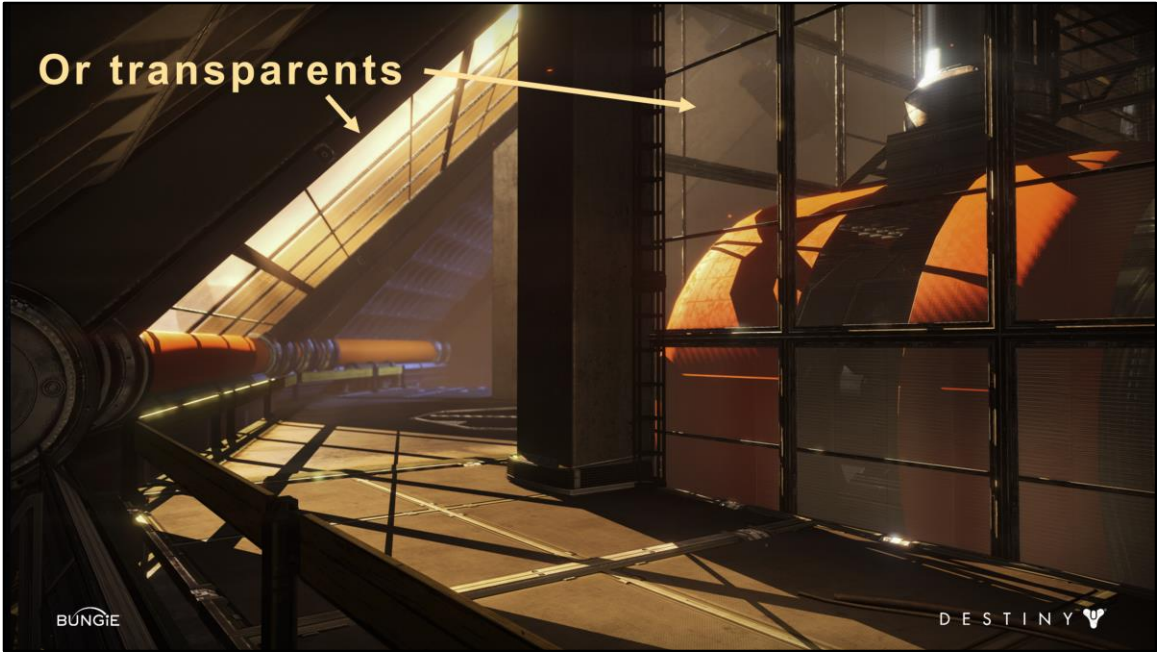
Our visibility solution was based on Umbra, which required bake-time specification of visibility occluders. However, certain materials should always be excluded from occluders ...



For example, this alpha tested environment geometry should never be marked as occluders



Or the water geometry shouldn't occlude



Same for the transparents

## Labels

```
component c_alpha_test:from_texture
{
    @add_label(VISIBILITY_OCCLUDER_DISABLED);
    ...
}

component c_alpha_test:disabled
{
    // no label metadata here
    ...
}
```

BUNGIE

DESTINY

Here is an example of adding a label to our alpha test component, with the goal of being able to query at bake time whether a shader can be used as a visibility occluder or not. The alpha test:from\_texture component specifies that it should not be used for visibility occlusion by providing VISIBILITY\_OCCLUDER\_DISABLED label, ....

## Labels

```
component c_alpha_test:from_texture
{
    @add_label(VISIBILITY_OCCLUDER_DISABLED);
    ...
}
```

```
component c_alpha_test:disabled
{
    // no label metadata here
    ...
}
```

BUNGIE

DESTINY

whereas the `c_alpha_test:none` component does not have the label metadata, so a query for the `VISIBILITY_OCCLUDER_DISABLED` label would return false and an object with this shader can be used as visibility occluder safely.

This mechanism gave us a simple way for components to drive which materials should be excluded from visibility occluders automatically, without any manual artists markup.

Another use case for labels was to help us fit into memory and get better GPU performance....

## Labels: Optimizing Geometry

Platform-specific optimization for memory and GPU / CPU perf

Reduce memory footprint for geometry

Ensure we don't generate garbage data

Generate performant runtime representation

- Faster GPU throughput
- Faster CPU submit

BUNGE

DESTINY 

... By using shader labels to help us optimize geometry at bake time.

We had to keep geometry footprint as tight as possible on last generation consoles to fit into memory.

Aside from keeping the memory footprint lean, this also helped us avoid generating garbage data when source geometry elements were missing (for example, when artists forgot to setup vertex color data in content). And by reducing the number of streams we had to setup, we slimmed down our CPU setup and increased GPU performance.



## Labels: Optimizing Geometry

Optimize vertex streams based on labels  
Strip unused vertex elements

Ensure source geometry provides required vertex elements

- At bake-time
- Avoids garbage data, GPU crashes

BUNGE

DESTINY 

At object bake time, we verify that source geometry has vertex channels that the shader requests. This allowed us to strip out vertex channels that shaders do not require for that geometry (which was used, for example, to strip out tangent space from depth prepass or shadow shaders for faster GPU processing for those stages). We also could provide dummy – safe- data when we found that a shader required a particular channel but the source geometry did not provide it due to incorrect content setup. This allowed us both to avoid GPU crashes (due to missing vertex data when shader wanted to sample it) and avoid rendering errors.

## Other Uses of Labels

- Physics materials
- Audio responses
- ...

BUNGIE

DESTINY 

We also used labels for a number of other systems – to figure out physics and audio materials from shaders automatically, for example..



... We also use metadata to help drive the shader bake process...

## Baking controls

Enable/Disable component splicing:

```
@splice_if(...)
```

Expression (...) can query labels or preprocessor variables  
Defined by client code or in TFX

BUNGIE

DESTINY 

The splice\_if metadata marks up components with an *expression* that we can use to enable or disable the *splicing* of that component.

These expressions can query labels, or other preprocessor variables, as defined by client code, or techniques or components elsewhere in the shader.

## Baking controls

Enable/Disable shader entry point compilation:

```
@compile_if(...)
```

Applied to `compile_shader()`

BUNGIE

DESTINY

We also can enable or disable the compilation of entire entry points using the `compile_if` metadata.

This is applied to the shader compile commands within a technique declaration.

This lets us conditionally compile shader stages, again, depending on shader options or client requests.

## Conditional Shader Compilation

```
technique default
{
    @render_stage("gbuffer");
    ...

    compile_shader(all_platforms, vs, main_vs);
    compile_shader(all_platforms, ps, main_ps);

    compile_shader(v2, hs, main_hs) @compile_if(TESSELLATION_ENABLED);
    compile_shader(v2, ds, main_ds) @compile_if(TESSELLATION_ENABLED);
    ...
}
```

BUNGE

DESTINY

For example, here, we can toggle tessellation, based on a declared label by tagging the hull and domain shaders with `@compile_if` like so.

This is saying, if someone defined `TESSELLATION_ENABLED`, then compile the hull and domain shaders; otherwise, don't.

Here you can see we might have been able to build some language constructs to do the same thing – put first class if statements in the technique -- and maybe you would get a bit better result out of that -- but it was just easier to throw on some metadata for it. This IS laziness on our part. 😊



Next, we want to take a deep look at how we dealt with the wide variety of shaders for our game

## Goals of Managing Shader Permutations

- Build a variety of shaders
- Minimize busy-work & maintenance
- Allow sharing & re-use

BUNGIE

DESTINY 

We want a system that allows us to build variety of shaders quickly and easily, yet we want to minimize the repeated, tedious setup work and all of the maintenance that comes with it.

We also want a system that allows us to create the building blocks once and then reuse them in many other shaders, and reduce the manual copy/paste by using references and instantiation whenever possible.



## Composability: TFX Components

Tool for providing choice via node-graph connections

BUNGE

DESTINY 

The first mechanism for composability is, of course, our TFX components. This is a great tool for managing permutations of shader functionality, but their boundaries are defined in code, which is not flexible enough for artist-centric workflow

## Composability: Content-Facing Blocks

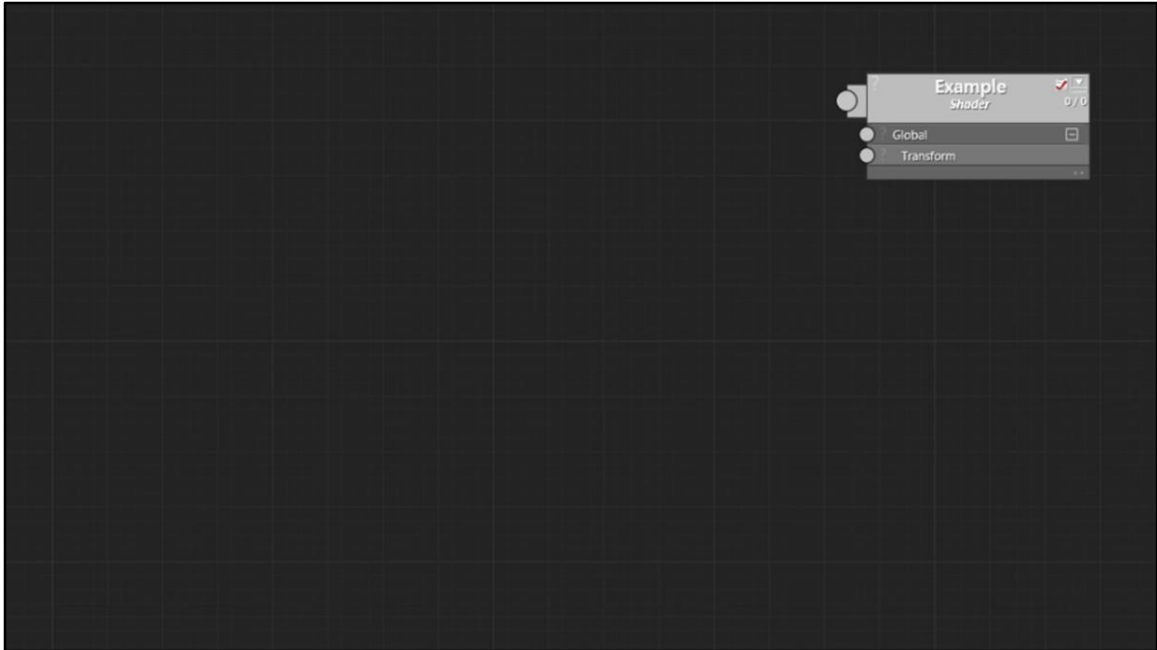
- Artist-facing sharing and re-use building blocks
  - Subgraph templates
  - Shader variants

BUNGIE

DESTINY 

We wanted to create a mechanism for artists to own that sharing and reuse.

The two main mechanisms we developed for this purpose were shader subgraph templates and shader variants.



**(video4)** Subgraph templates allow artists to reference other node graphs as subgraphs in their node graph files.

So we can just take any node graph file, and drag it in and drop it into OUR node graph file.

And you can see that other node graph. Now you can't change any of these nodes, but you can connect your nodes to them, so we can use their combined transform as our transform, by connecting it up like so.

And, you can also make overrides, like say we don't want a sphere warp, we want some other transform, like a radial shear. We can just override that connection in the template with a local node.

So you can connect to, and override any part of the graph; they can use 'just the middle' if they want.

This is a very simple example, but the artists used this to be able to create reference shaders, containing pieces of node graphs they wanted to share, or even things like libraries of reference colors and shared parameter animations. If every shader is

looking at the same file for those things, then you can change all of them in one place.

## Shader Variant Layers

- Often artists want to vary minor elements of the material
  - Paint colors
  - Metalness amount
  - Dust overlays
  - ...
- Multiple axes of these variations
- Results in **combinatorial growth of possibilities** !
  - A huge amount of files / data to manage

BUNGIE

DESTINY

Another thing we found, was that artists frequently wanted to vary minor elements of their materials.

For example, they might have some peeling paint in their shader, and they wanted to have a variety of paint colors available for different areas in the game.

Or they might have material parameters such as metalness that they want to vary, or they want to create dust overlays on Mars.

This was a very frequent desire from our palette artists for handling different destinations or enemy factions.

And, they wanted to have multiple axes of these variations.

Which quickly results in a **combinatorial growth of the possibilities** !

If you were to copy/paste your shader and modify those values manually, this is a huge amount of files & data to manage, and changing anything later becomes a nightmare.

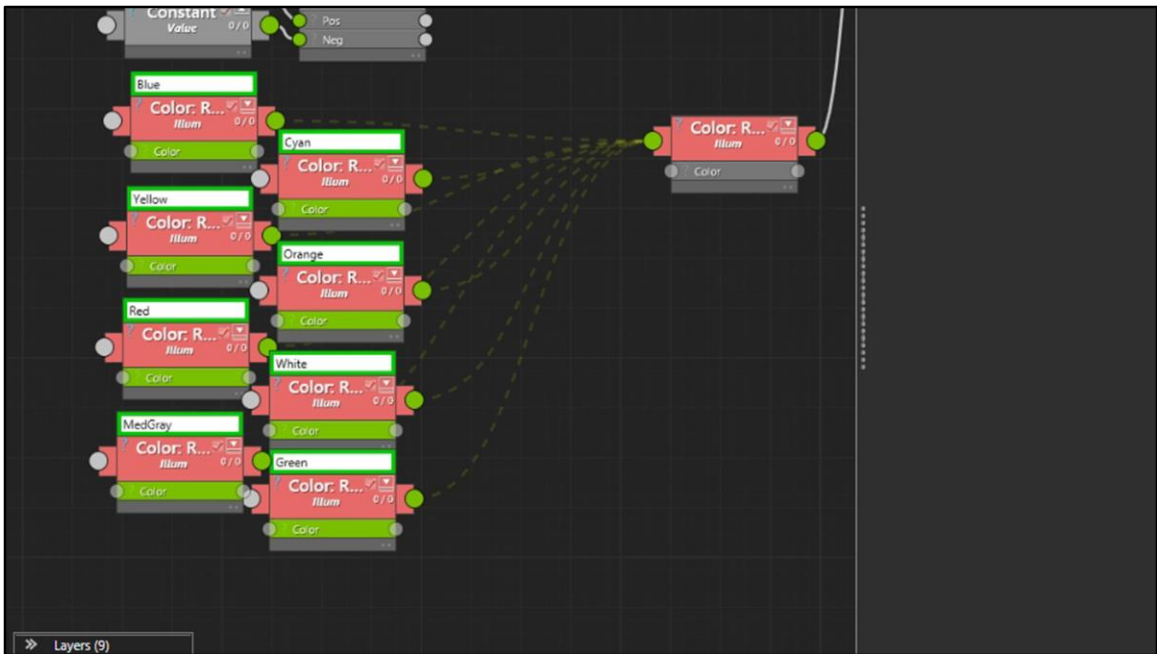
## Shader Variant Layers

- Define shader variants from a single shader node graph
- Layers override node connections and/or parameters
- Each layer has a set of conditions in which the layer is active

BUNGE

DESTINY 

What if we instead define all shader variants from a single node graph file, where we encode rules about how to modify the shader in a set of layers, conditioned on some logic. Each layer can override any node connections in the node graph and change parameter values. This lets the artists define variants of the shader within a single file



**(video5)** This is a concrete shader, where the artists have set up a worn painted section.

But they want to have a variety of different colors of paint. And rather than make a different shader for each possibility, they've instead made use of variant layers.

Each of these connections here is a connection to one of these different color nodes, and the connection belongs to a variant layer.

The way it works is they have defined a number layers, each one is active only when the conditions are met: color is blue, color is cyan, etc.

When you activate a layer, it's connections become active, defining the color to be the appropriate value.

This is a very simple example of overriding just a color, but you can make any number of connections in a layer, and potentially produce very different graphs if you wish.

In this way, the artists can define a great variety of looks within a single node graph file.



## Shader Node Graph Variant Layers

### PROS:

- Easily setup variety of slightly permuted shaders
- Extremely convenient for content creators to manage

BUNGE

DESTINY 

PROS: This allows the artists to set up a huge variety of shaders easily and manage this complex set of choices in an easy UI. Such a large diversity of shaders would be a huge management pain without a variant system.

## Shader Node Graph Variant Layers

### PROS:

- Easily setup variety of slightly permuted shaders
- Extremely convenient for content creators to manage

### CONS:

- Combinatorial possibilities
  - We've seen 4000+ potential variants in a single file
  - DON'T build ALL variants if you can help it

BUNGIE

DESTINY 

The cons however – is you have just enabled them to make 4000 variants in a single file; which, by itself, is not really a problem if you don't build all the variants. Keep this in mind just in case, because the artists should be aware of their choices.



Of course, no system is ever perfect, and we had a number of challenges along the way, and we found a lot that we want to improve in the future.

## Managing Splicing Order

HLSL requires declarations before use  
C-style

Splice must be ordered correctly or compilation fails

Not good..

BUNGIE

DESTINY 

The HLSL language, and all of the similar shader languages, use C-style declarations; which means you must **declare** something before you can use it.

This means that every HLSL fragment must be spliced in correct order, or else the compilation of the shader will fail.

## Managing Splicing Order

Early on, we chose to splice in TFX declaration order  
breaks almost immediately

So we added metadata:  
`@splice_order(sort_key)`

BUNGIE

DESTINY 

Initially, we did the simple thing and spliced in whatever order things were declared in TFX.

This quickly broke, so we added metadata to control splice order explicitly.

You could tag components or parameters with `@splice_order`, giving it a sort key, and it would splice in sorted order.

And this worked for a while...

## Managing Splice Order

Sort keys and conditionals are high maintenance  
Complexity grows quickly when adding more options

### Better solution: splice in dependency order

Automatic, no maintenance  
Better, more minimal results

BUNGIE

DESTINY 

But, as complexity grew, and we added more permutations to our shaders, the custom splice ordering became increasingly complex to maintain and challenging to debug.

A much better solution, we found, was to splice on demand, in dependency order.

This just solves the problem automatically, no maintenance necessary. and it also trims down your generated HLSL, removing all the unnecessary declarations.

We have a working prototype of this, but we haven't gotten around to implementing it in our pipeline due to ship schedules and other priorities.

## Distributed options

Options that modify several locations in code

- per-vertex data options
  - declarations & fetch code
  - interpolator definitions
  - pixel shader access

- 'global' shader state structures

BUNGIE

DESTINY 

Another issue we ran into was concepts like per-vertex data options – that is, options that want to modify several different locations in the generated HLSL code.

For example, adding a `vertex_color` to your vertices requires modifying the vertex stream declarations and fetch code, the interpolator definitions, and pixel shader access functions.

This doesn't fit well in a simple component.

## Distributed options

### Our Solution:

Use #defines to enable/disable pieces of code (old school)  
Hide all the ugliness inside a component

Fragile & hard to maintain  
Fine for small number of permutations  
Quickly becomes monstrous as you add more

BUNGIE

DESTINY 

Our solution was basically to just fallback to the old-school solution:  
- use preprocessor definitions to enable and disable sections of code,  
- and stuff the whole thing inside a giant component that manages everything.

While this let us address this problem during ship, it was very fragile and a pain to maintain as we scaled up.



## Distributed options

Better Possibilities?

Ability to splice to / from named fragments?

Higher level code to pack resources?

Interpolators?

BUNGIE

DESTINY

A better solution, I think, is to add the ability to splice into a named fragment, allowing us to splice together options from various different components, and then splice THAT fragment into the generated code on demand.

In this way we could build up definitions of streams, interpolators and structs, from a bunch of pieces coming from different components.

Another option is to write a higher level, more intelligent system to do packing of things like this and generate the HLSL for it.

<transition>



... so to conclude...

## TFX Shader System Conclusions

Data-driven design enabled performant rendering

Unleashed the creativity of artists for material and look variety

Provided ease of cross-platform development

BUNGIE

DESTINY 

We designed TFX to be a heavily data driven system, which enabled fast iteration but at the same time it can also yield performant rendering from the deep encapsulation of data and control of frequency of submission.

The flexibility of our system allowed us to give the power to the artists and unleashed their creativity which was crucial to achieve the variety of destinations and material types for Destiny.

TFX was one of the key components for tackling multiplatform development for Destiny, allowing to write shaders once and quickly create platform-specific resources using optimal representations for each destination.

## TFX: Future Work

- Black-box Templates
  - Top-down control of how your template is used
  - Easier to modify template later
- Parameter-only templates
  - Reduce compile steps, which are the most expensive
- Splice on demand
  - Reduce shader GPU state size
  - Get rid of splice ordering issues
- Bytecode inspection / decompiler would be great

BUNGE

DESTINY 

Of course, we are never done, and there are a number of future work that we hope to see TFX evolve to, such as improvements to the shader template system to do black boxing of templates, or parameter-only templates; we want to support on-demand splicing to reduce GPU state and improve our splice order issues. And although bytecode was a tremendous advantage to our ability to drive GPU state dynamically, inspecting was challenging, and we want to extend the system to support a better bytecode decompiler to make it easier to debug.



## Slides

- <http://advances.realtimerendering.com>
- Twitter: @mirror2mask
- [ctchou@bungie.com](mailto:ctchou@bungie.com)

BUNGIE

DESTINY 