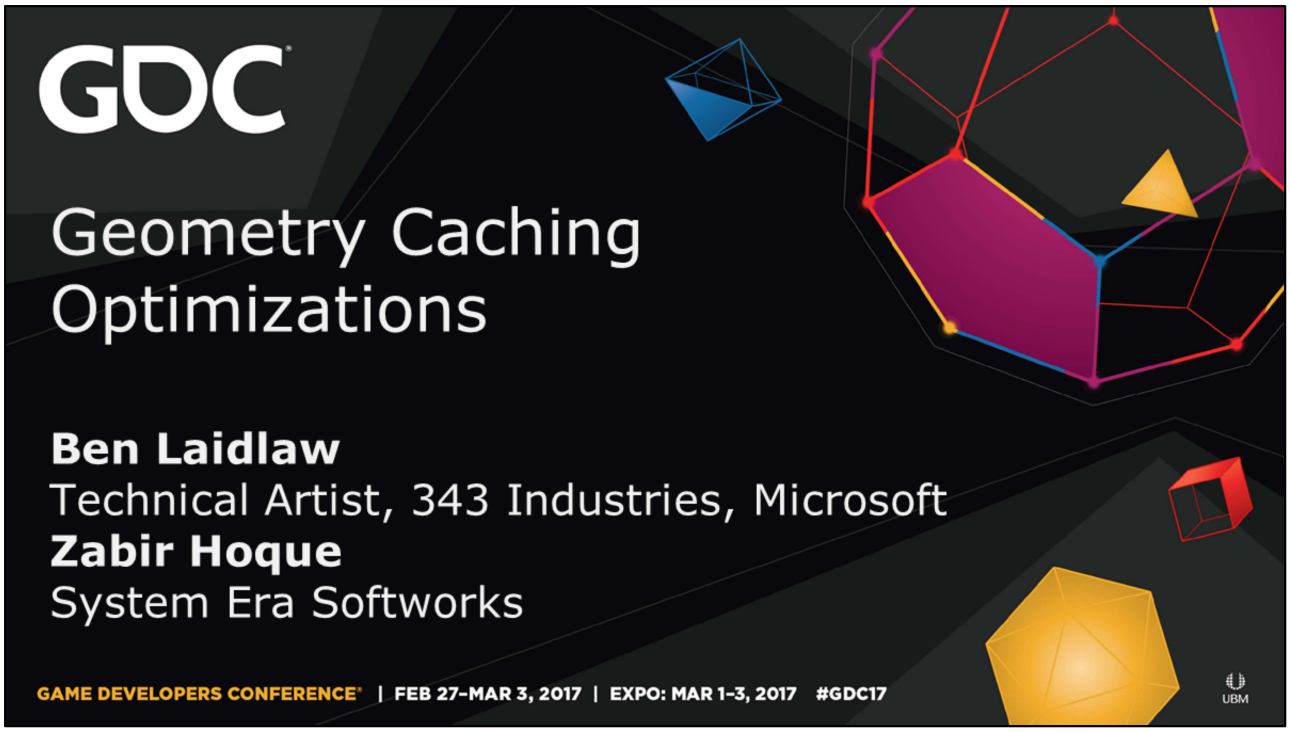


- 00:00, 00:45, 58:15

Hi all,
Welcome.

My name is **Ben Laidlaw**

My name is **Zabir Hoque**

And we are here to talk about **Geometry Caching Optimizations** that were implemented **in Halo 5**: Guardians.

- 00:00, 00:45, 58:15

- For those that are **new** to the subject.

Geometry Caching is **the storage of vertex motion for quick playback**.

- It is like **motion capture** for character animation

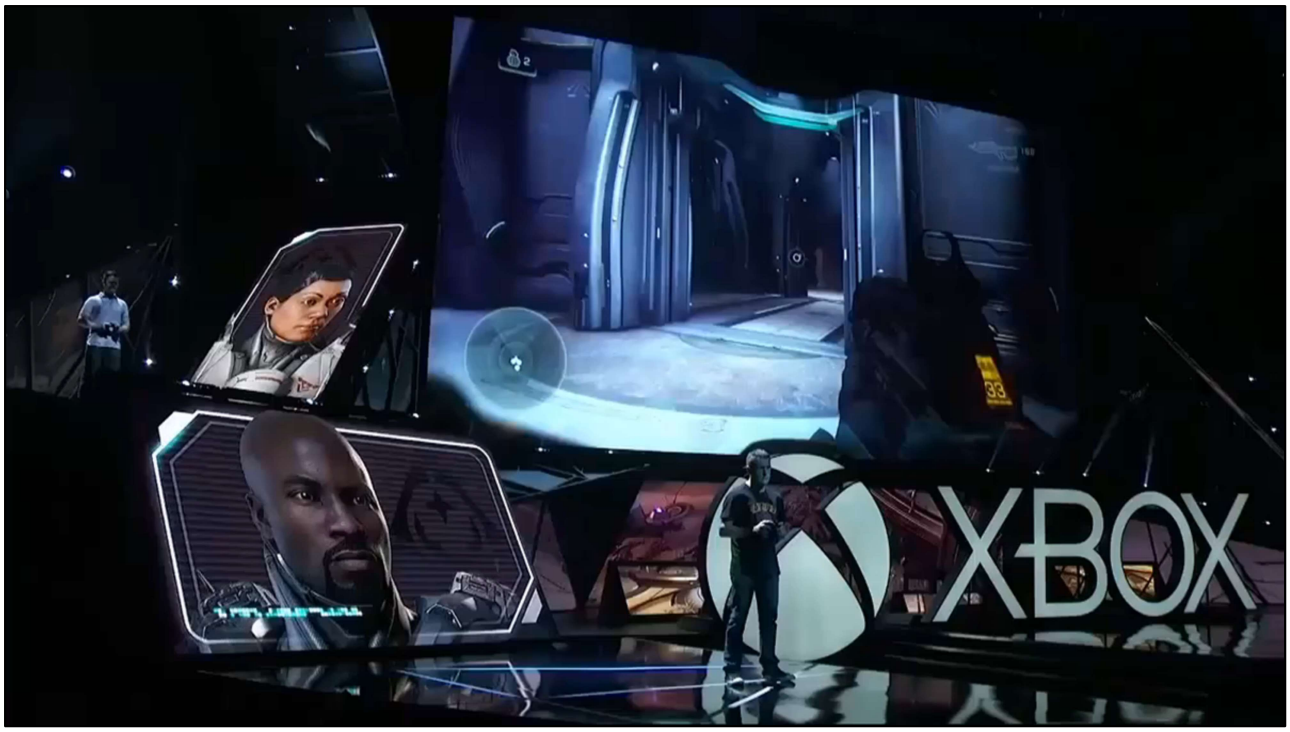
it allows us to create vastly more complex animations in games
but **authored in a convenient format.**

- For those **Maya** animators out there, they'll be familiar with the term,

when they want to do a 3-D playblast of their scene.

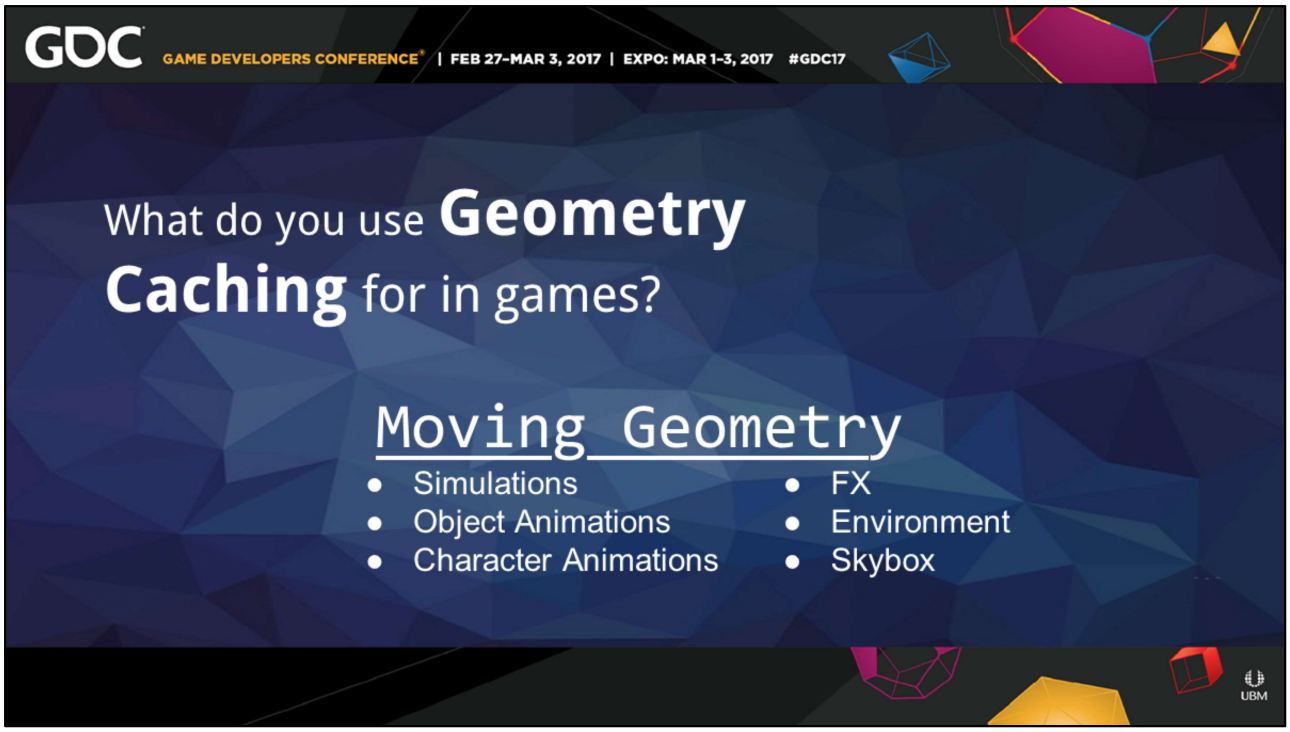
- Let me show you an in **game example** from Halo 5: Guardians,

from our E3 Headliner in 2015.



- 00:45, 01:00, 57:15

Halo 5 E3: Headliner



- 01:45, 00:35, 56:30

- Beyond being able to do massive scale destruction.

- What we found is Geometry Caching can be leveraged for **any kind of Kinematic** based geometry.

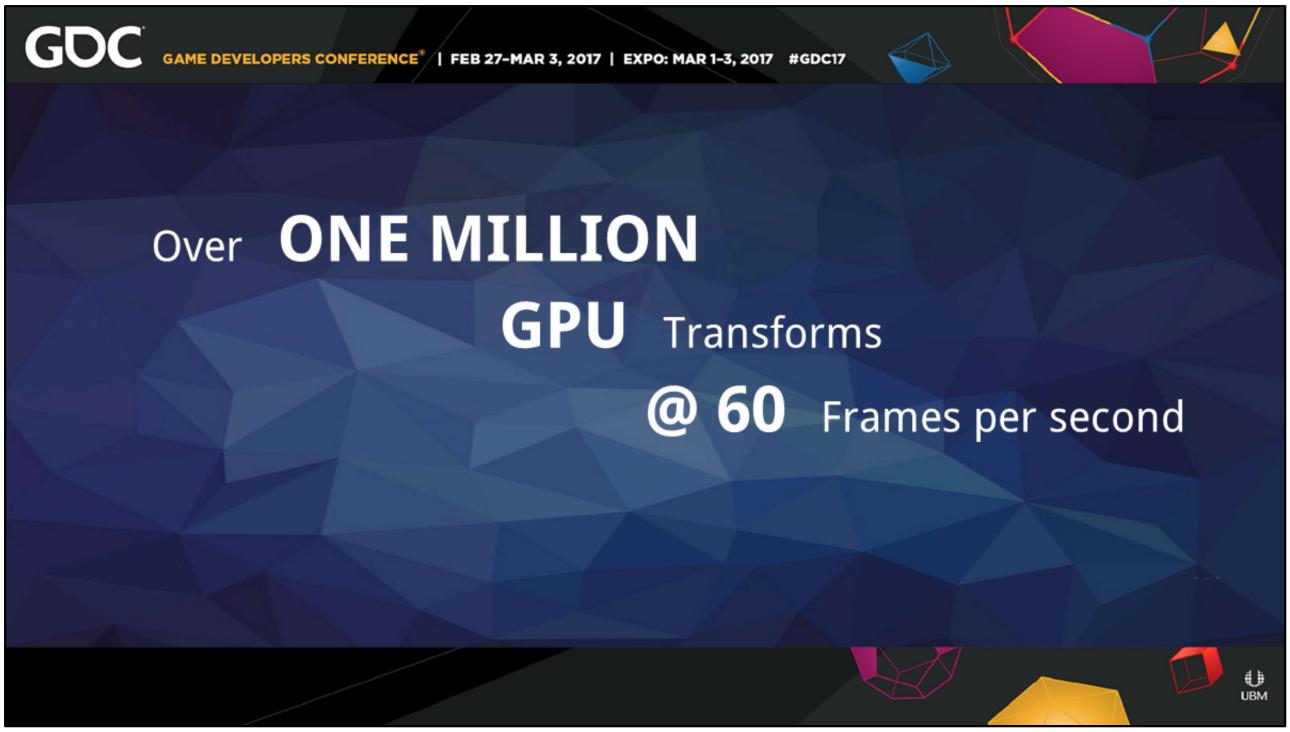
Such as, Animating characters or objects, FXs, Environments, Skyboxes, and crowd systems,

- The **authoring teams** of this content can run the gamut of your studio,

it's more of a **compression method** than a team specific task.

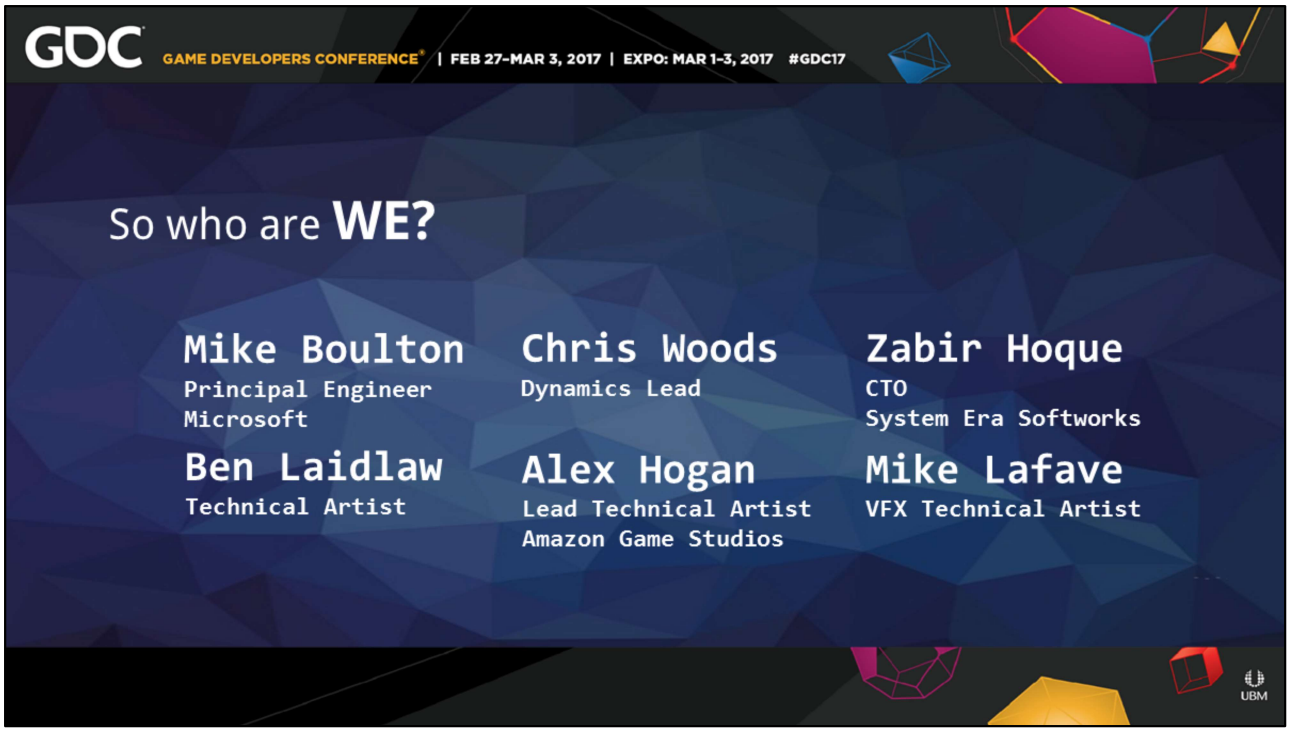
- What we are here is to really talk about

is **moving** geometry around on the GPU as fast as possible.



- 02:10, 00:10, 55:55

- We are talking about moving **over a million** gpu transforms
at 60 frames per a second gameplay
that can be driving up to 4 million polygons.



    - 02:30, 00:45, 55:45
- Zabir and I didn't do this all alone, so here is a quick **shout out** to the core team people

- **Michael Boulton** was the engineer whom made our first homebrew geometry caching pipeline

-the fore-father of what we are presenting today.

- **Chris Woods** was our fearless lead.
- **Zabir Hoque** was our white knight engineer that came in during the heart of crunch,

-I'll pass the torch to him in a little time.

- **Alex Hogan** was the Technical Artist that helped with the initial testing and implementation.
- **Mike LaFave** was a content creator for our project, that thoroughly put the process through its paces.
- My name is **Ben Laidlaw** I'm a Technical Artist that became in charge of the content authoring story.
- I also want to give props to **all the other people** that worked on us with this at 343.

- 03:15, 00:45, 55:00

- For the overview of this talk, we're going to hit up how we derived our **concept** on geometry caching

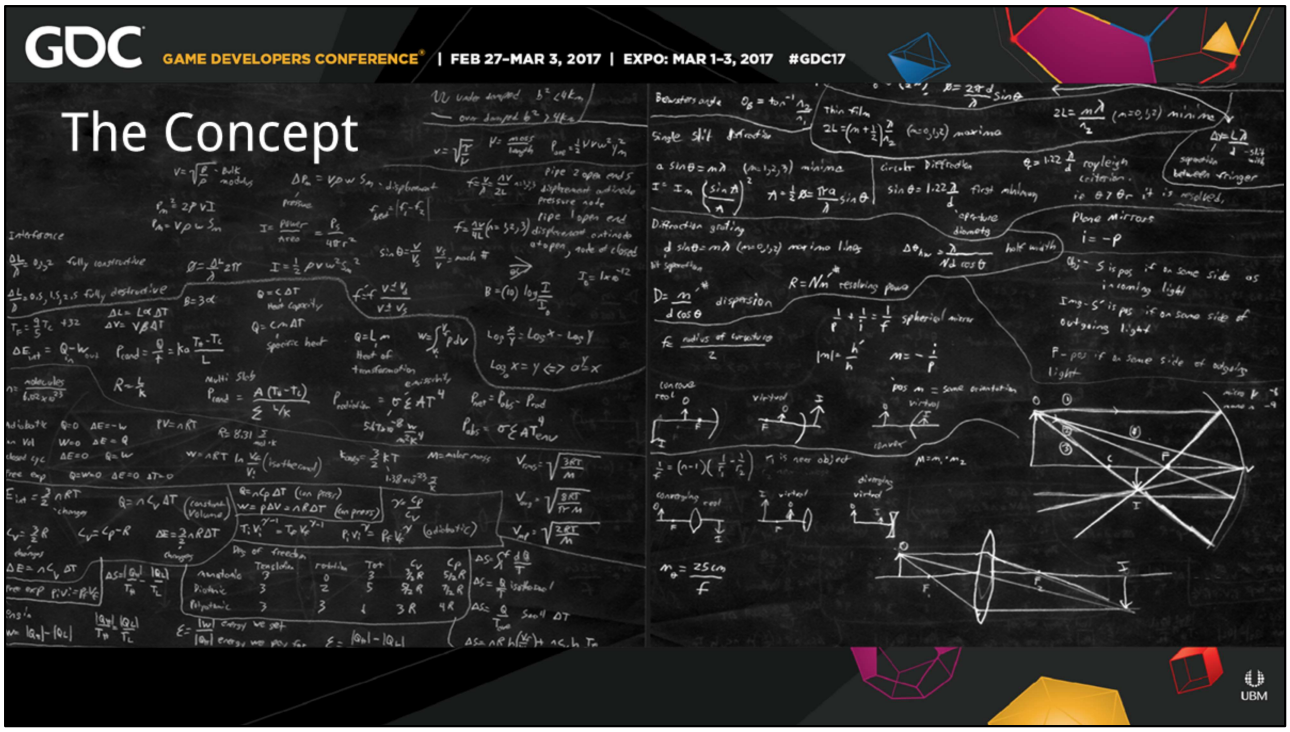
and how we mentally broke down what we optimized

- With a little **background story** in how we got there.

- Then we will break down the **optimizations**,

First by the **core architecture**
Followed by the **components**

- And then will wrap it up, with some **time for questions**.



- 04:00, 00:30, 54:30

::Breath::

- In the spring of 2015 we were **coming in hot** for the E3 Headliner.

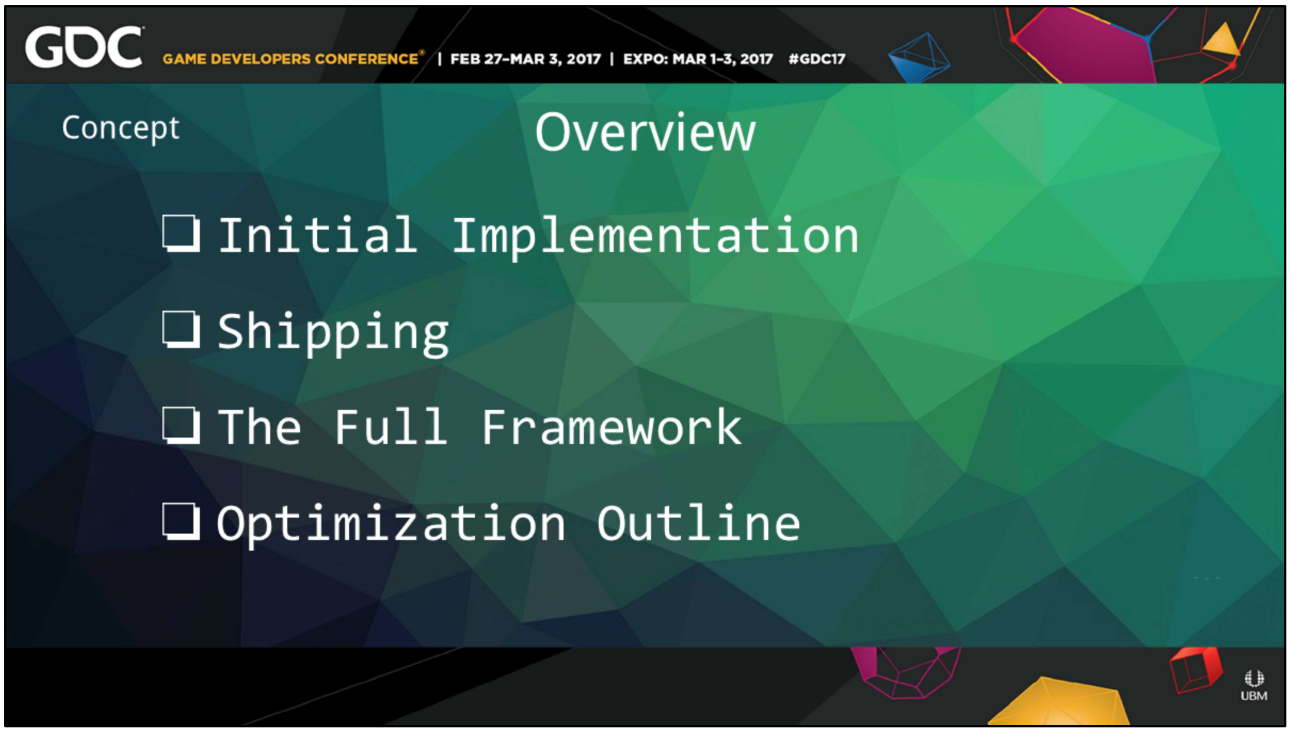
It was going to be the **opening act** for E3, so all eyes were on us.

- Like most studios, our **frame budget spiked**

as we tried to squeeze **the best** we could out of the game
On the other side, we had production eyeing to **cut any assets**
that would loosen the gordian knot.

- You know the usual, **eye of production** fun. The eye of sauron.

So we needed to **improve** our work



- 04:30, 00:30, 54:00

- For those wondering why we needed to improve.

Let me give you a little background.

- I will explain how we ended up developing our geocaching tech over time.

From our **initial implementation**, through to **shipping** of Halo 5: Guardians

- and beyond to what our **full framework** was to continue our development

- 05:00, 00:45, 53:45

- Our very first implementation of geometry caching, is what we have come to call **Fluid.**

It is a geometry sequence where **each frame can potentially change its topology**.
Overall it is the catch all, for all types of complex motions
I have a deer that transforms into a man, into a gun, you can pretty much do anything.

- We based this method on what we had seen other people do.

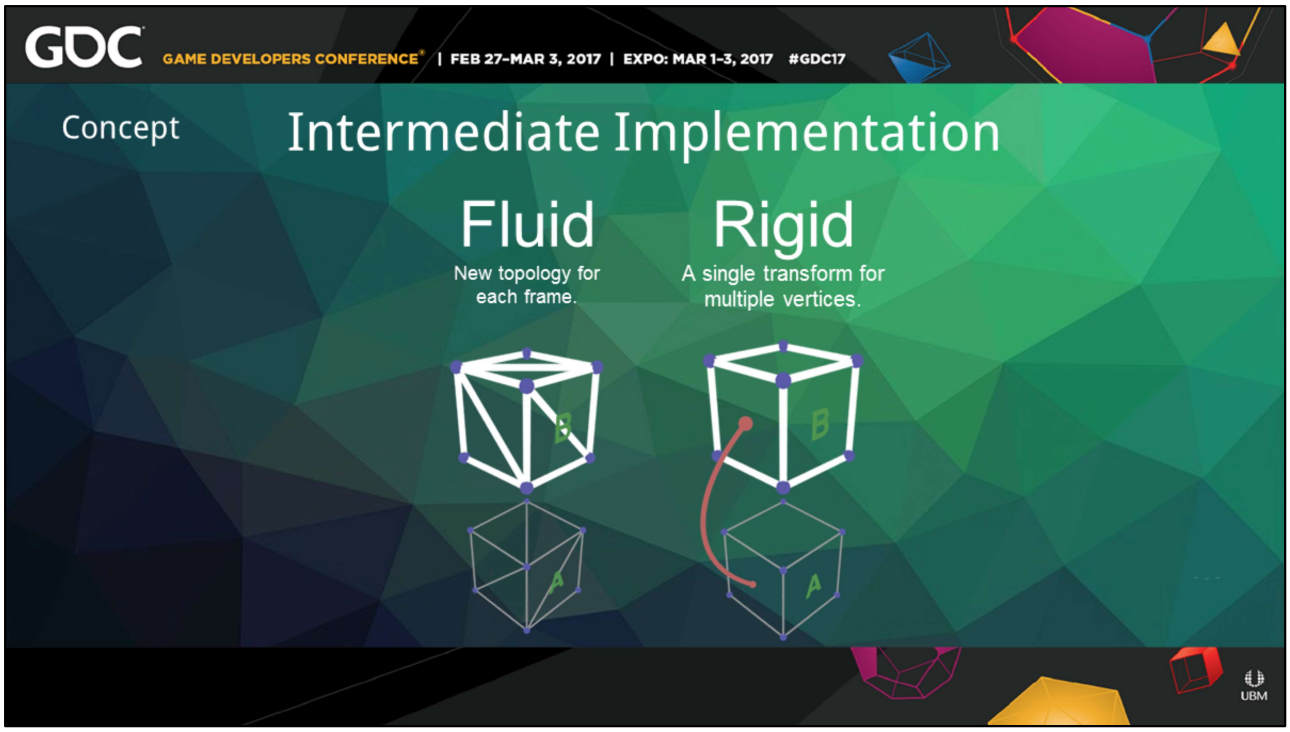
This seems to stem from how the VFX industry write simulations **frames to disk**.
A practical derivative of this work is the read/write from an **alembic** file
For instance, **Maya's geometry cache** method used by animators

- This is a good initial working model, but it's **not** a practical **shippable** method

due to the memory and perf cost associated with these frames at 60 fps.
The files are a bit too big, and a bit to slow. If you need a small set of these in your game it will work at 30 fps.

- So one of the last valiant acts from **Michael Boulton** before he got pulled off

was to implement a second option for us called Rigid.



- 05:45, 00:45, 52:30

- **Rigid** is what we defined as a sequence of geometry where **multiple vertices share a single transform**.

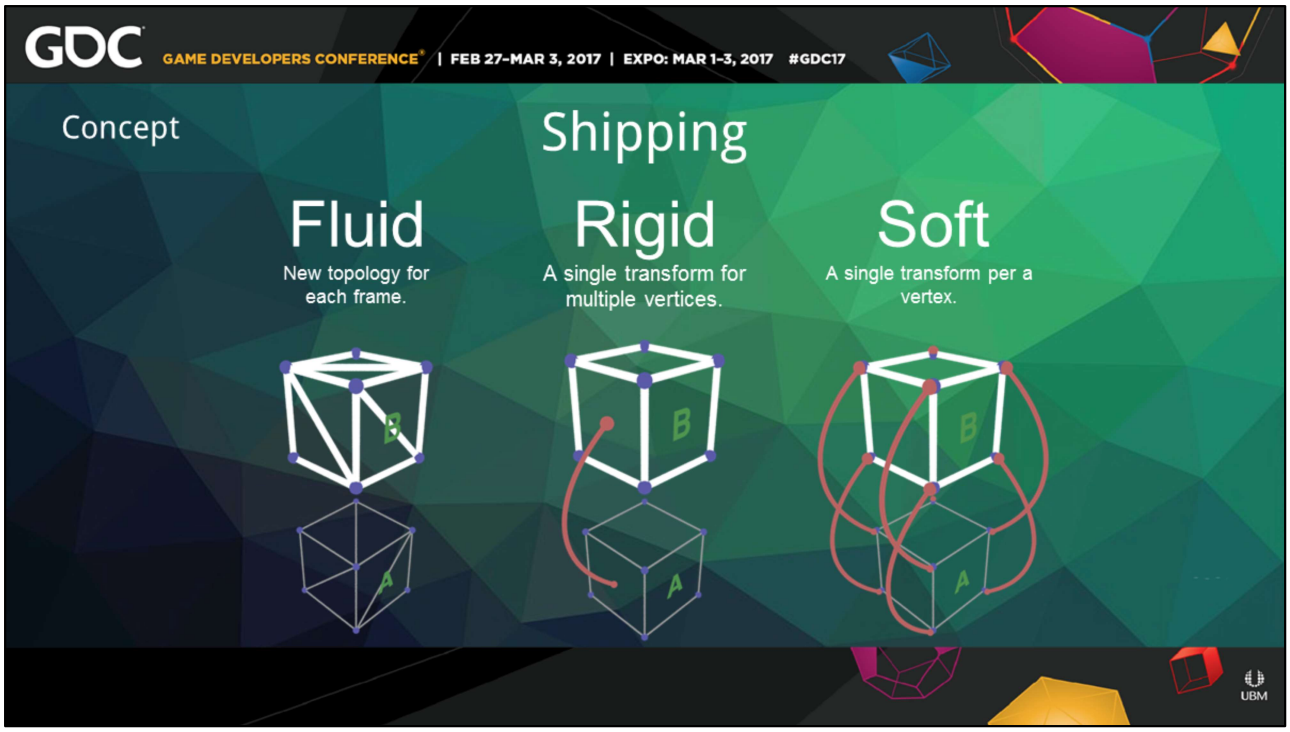
i.e. the concept of bones in object transforms, particles with instanced geometry, your standard rigid body simulation, Flocks, Swarms, and Leaves blowing in the wind.

- This became our **golden child**.

Rigid accounted for a majority of our geometry cached assets we shipped.

- However these two methods are not enough to ship.

We were still in the need for **deformation** style motion, Bending metals and such and one night Zabir was doing some **crazy scientist** math at home, and he realized our next evolution in this pipeline.



- 06:30, 00:45, 51:45

- Thus the creation of what we called **Soft**. Soft is **a single transform per each vertex**.

This allowed us to **leverage** the hundreds of thousands of rigid transforms we had easy access to
and they were applied to each vertex instead.

- Creating nice **bending geometry**. like metal deformation, tarps and flags waving or even lava.

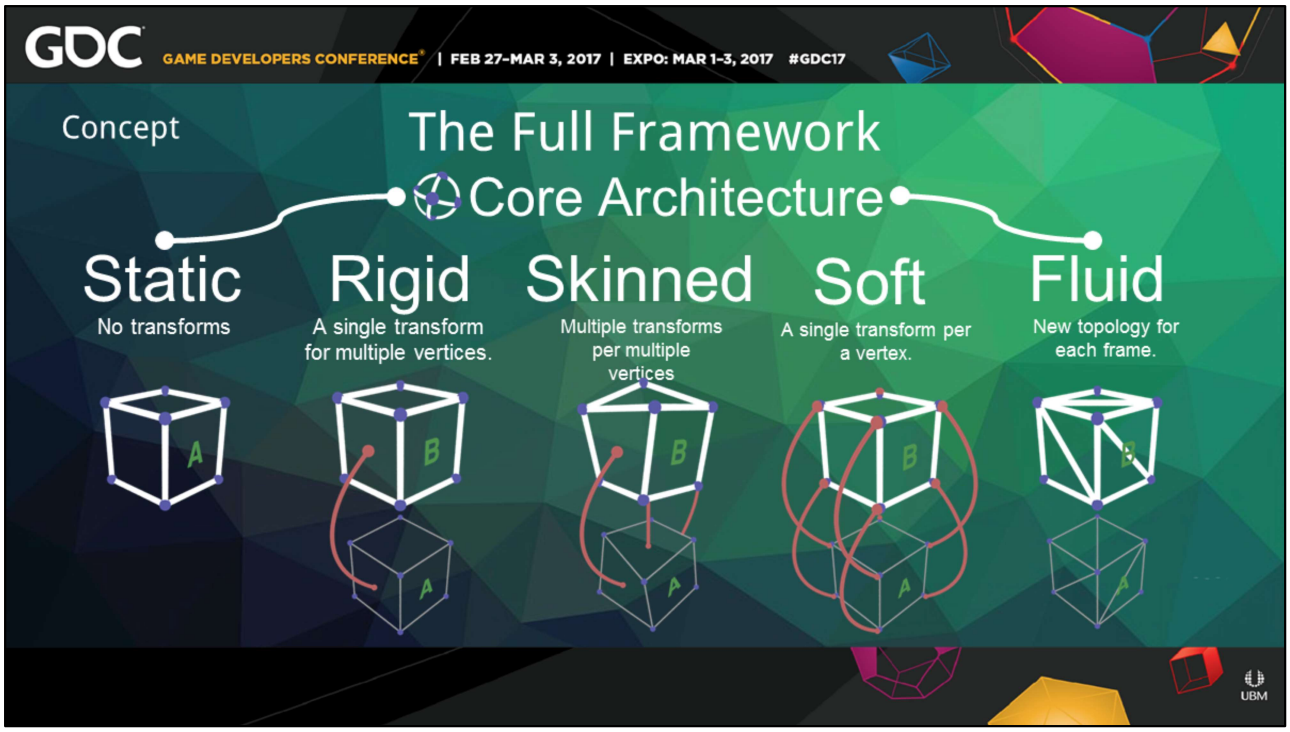
So with Fluid, Rigid, and Soft **we shipped** Halo 5: Guardians.
::BREATH::

- However this is not where we stopped.

To further develop our system, we created **a framework** for geometric motion
at least as far as how we thought about it

- which lead to how we optimized it, and **how YOU** can further expand upon it.

::point to people::



- 07:15, 00:45, 51:00

- So with the **full framework**. we present that there are **five types** of inherent geometry caching methods,

Static, Rigid, Soft, Skinned, and Fluid. This is designed to be read top to bottom, left to right.

- Beyond our shipping methods we added three more contexts
- **Core Architecture** is our bridge across all the methods,

These are the shared heuristics, such as, geometry, transforms, and rendering

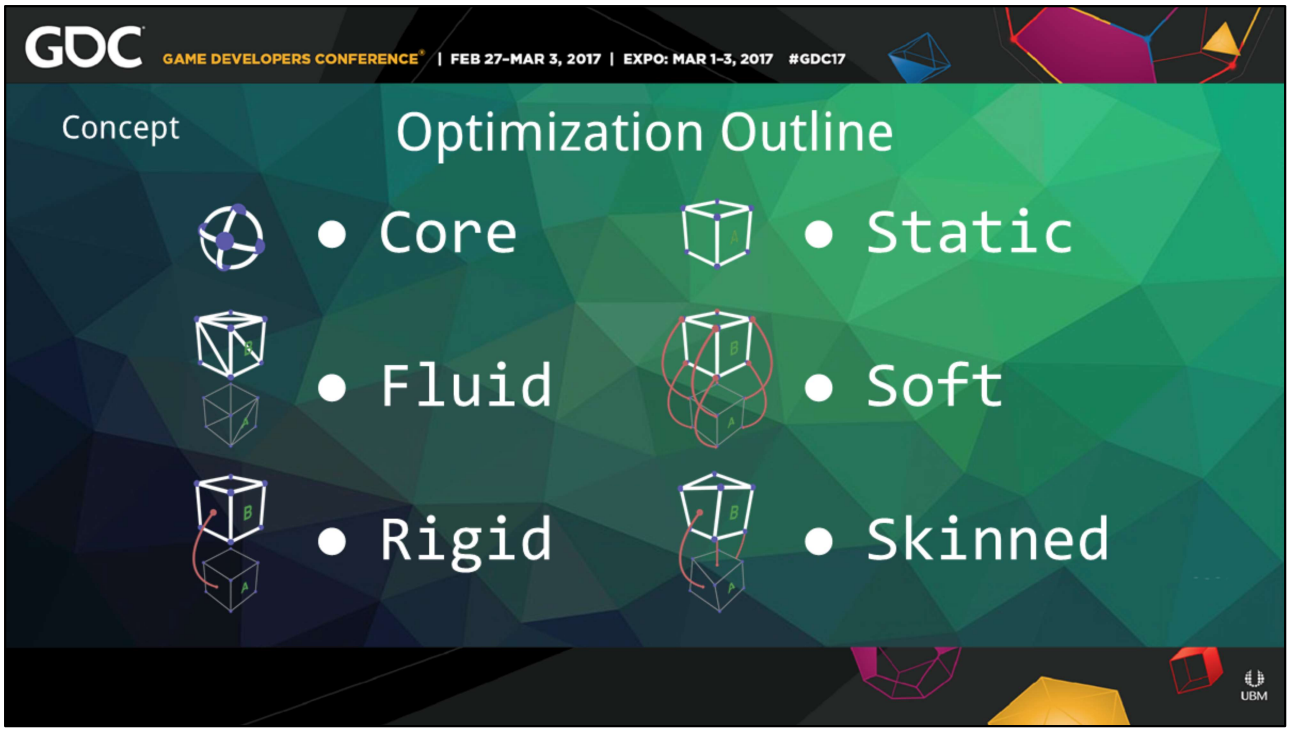
- **Static** geometry is an additional method being called out, As static geometry has no transforms over time,

This becomes very important in cost reduction. I believe we are all inherently familiar with environment geometry

- **Skinned** is as you may imagine deals with character like deformation

this is where we use **multiple transforms upon multiple vertices**,
This is way cheaper than soft, but not as cheap as rigid.
but more complicated for artist to author and implement.

- Also you can blend these types as **hybrids** versions, too.



- 08:00, 00:30, 50:30

- The **order** in which we will present these optimizations
is a little different than how we abstract their relationships
based on their use cases and ease of implementation.

- The **Core Architecture** is our first point of attack,
It not the panacea, but it does help every single component
Zabir will talk in depth on that.

- Followed by **Fluid** as this is where everyone usually starts off, building their geometry caching pipes

Then comes our golden child **rigid**,
Followed, by **Static**, **Soft** and **Skinned**

Core

```
uint grpFirstIdx = _xb_make_uniform(Gid.x << log2(GROUP_THREAD_COUNT));
uint grpLastIdx  = _xb_make_uniform(min(grpFirstIdx + GROUP_THREAD_COUNT, ve
uint grpFirstXFormIdx = VertexBuffer[grpFirstIdx].XFormIdx;
uint grpLastXFormIdx  = VertexBuffer[grpLastIdx ].XFormIdx;
if (GI <= (grpLastXFormIdx - grpFirstXFormIdx))
{
   LDSPartXForms[GI] = CalculatePartTransform(grpFirstXFormIdx + GI);

}
```

Core

Overview

- ❑ Data Structures
  - ❑ Transform
    - ❑ Alternate Representations
  - ❑ Geometry
    - ❑ Implicit Surfaces
- ❑ Import Pipeline
  - ❑ Transform/Geo Extraction
  - ❑ Packing & Sort Order
- ❑ Runtime
  - ❑ Interpolation
  - ❑ Pipeline
  - ❑ AsyncCompute
  - ❑ Shader

Core

Data Components

- Transforms & Geometry
  - Static
  - Rigid
  - Skinned
  - Soft(Implicit)
  - Fluid

Transforms

Core

Uncompressed Transform

```
struct Transform
{
    float4 rotation;
    float3 translation;
    float  scale;
};
```

- 8, 32-bit float

  i.e. 32 bytes

Core

1st Alternate Representation

- Per Frame Delta Position/Rotation
  - \+ Much better compression
  - \- Join in Progress
    - Not confident in “Blast Playing”
- Retrospectively, right decision not to rely on blast playing.

Core

2nd Alternate Representation

- Per Frame Velocity/Acceleration
  - \+ Temporally Coherent
  - \- Physics Simulation Stability
  - \- Still requires “Blast Playing” for JIP

Core

3rd Alternate Representation

- Major Frames (ala video codec) + Delta Compression
  - Promising
  - But, high complexity
- 5 GB on XboxOne
  - “We’ll never run out”
  - I was very wrong

Core

Uncompressed Transform

```
struct Transform
{
    float4 rotation;
    float3 translation;
    float  scale;
} ;
```

- 8, 32-bit float

  i.e. 32 bytes

Core

Compressed Transform

```
struct Transform
{
    uint32 packedQuat;
    uint16 translateX;
    uint16 translateY;
    uint16 translateZ;
    uint16 scale;
} ;
```

- ❑ 1 Unsigned 32-bit integer
- ❑ 4 Unsigned 16-bit integer

  i.e. 12 bytes

  32 bytes -> 12 bytes

Core

Translation Quantization

- ❑ AABB per frame
- ❑ Quantize to 0 to 1 within AABB
- ❑ Save full precision AABB per frame

```
Translation = AABB.min + AABB.extents * quantizedTrans;
```

Core

Quaternion Compression

$$x^2 + y^2 + z^2 + w^2 = 1$$

Core

Quaternion Compression

$$w = \sqrt{1 - x^2 - y^2 - z^2}$$

Core

Quaternion Compression

```
if (w < 0)
    <x, y, z, w> = <-x, -y, -z, -w>
```

Core

Quaternion Compression

Drop max(x, y, z, w)

Store x = 0, y = 1, z = 2, w = 3

Common

Quaternion Compression

$$\text{sqrt}\ (k^2 + k^2 + 0^2 + 0^2) = 1$$

$$k = 1/\text{sqrt}(2) = \sim 0.707$$

$$\langle x_0, x_1, x_2 \rangle \ / \ k$$

Core

Quaternion Compression

$\langle x_0, x_1, x_2 \rangle$: 30 bits

$\langle i \rangle$: 2 bits

Core

Scale Quantization

- 1 Unsigned 16-bit integer
- Quantized to 0 - 1
  - Only allowed shrinking
  - Importer normalized
- More than enough precision
- Scale 0 allows deletion of parts (Rigid/Skeletal)

Geometry

Core

Geometry Representation

- Requires 2 vertex representations
  - Pre GeoCache Transform
  - Post GeoCache Transform
- Want to integrate into existing rendering pipeline post transform
- Rendered as static geo

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    Uint32 color;
};
```

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

Transform

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint32 color;
};
```

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

Passthrough

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint32 color;
};
```

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

Transform

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint32 color;
};
```

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

Transform

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint32 color;
};
```

Core

Vertex Format

```
struct GeoCacheVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint16 xformID;
};
```

Look Up →

```
struct WorldVertex
{
    float3 position;
    half2  UV;
    UDecN4 normal;
    UDecN4 tangent;
    uint32 color;
};
```

Core

Geometry Representation

- ❑ Started with cheapest mesh rendering w/ dynamic lighting
  - ❑ For us world vertex
  - ❑ Pass through vertex shader*
- ❑ Lit using light probes
- ❑ Dynamic Shadows
  - ❑ Rendered into CSM

Core

Vertex & Index Buffers

- Vertex Buffer
  - 65k parts, used 16bit transform indices
- Index Buffer
  - 32bit due to vertex count
  - XboxOne really doesn't like this
- Use 16bit indices
  - Continually bandwidth bound
  - Split assets to stay within 16bit index
    - More work on content creators/tools

Core

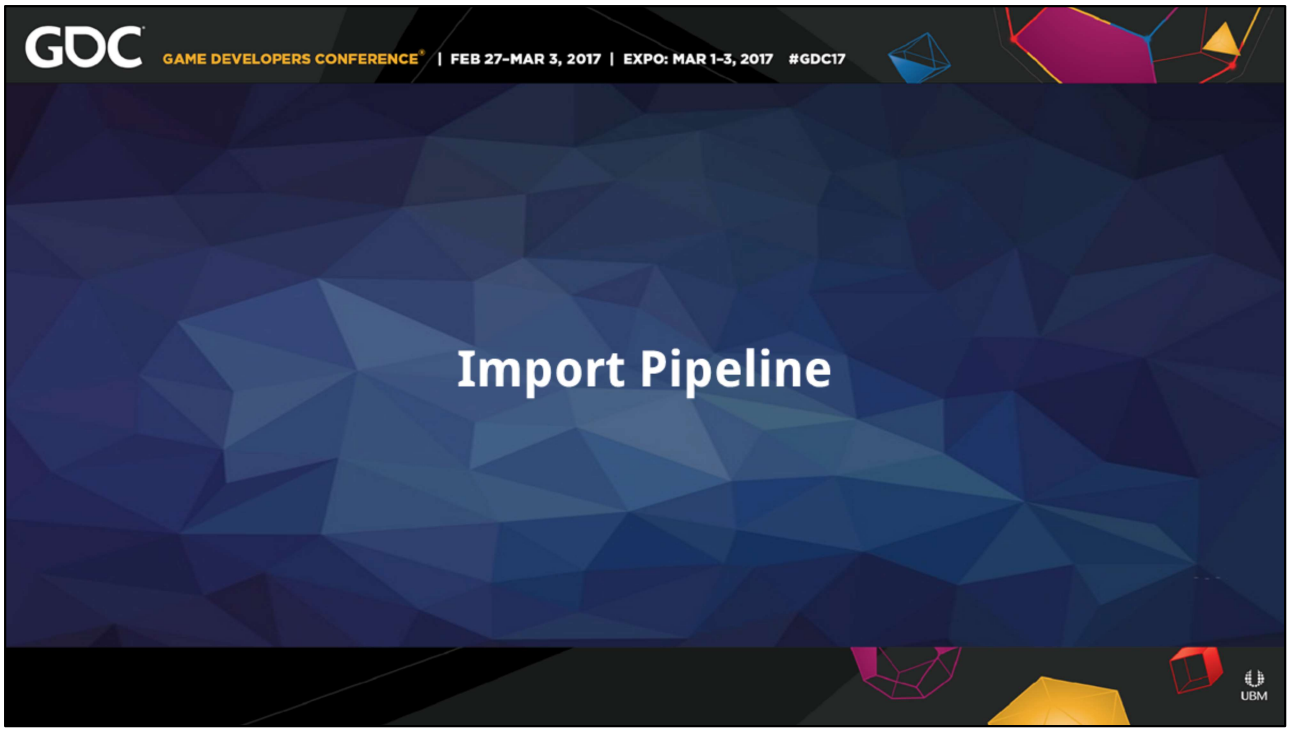
Implicit Surfaces

- Topology is implicit for uniform plane
- Used SV_VertexID and derived the partID
  - Used vertex shader instead of Async Compute
- No LDS, but unique transform per vertex anyway

Core

Additional Vertex Attributes

- Transforms are driving dynamics
- Additional per transform data can drive shading & events
  - E.g. temperature, viscosity, color, etc.


GDC
GAME DEVELOPERS CONFERENCE | FEB 27–MAR 3, 2017 | EXPO: MAR 1–3, 2017 #GDC17
Import Pipeline
UBM

Core

Import Pipeline

Core

Alembic Representation

Frame 0 Frame 1 Frame 2 Frame 3

Core

Analytic Part Extraction

- ❑ Interframe Analysis
  - ❑ Compute covariance matrix of frame 0 & N
  - ❑ Compute SVD
  - ❑ Apply SVD Result to frame 0
  - ❑ Compute least square error between 0 & N
- ❑ Only 85% accurate
- ❑ Error Tolerance
  - ❑ import time vs incorrect motion

Core

Explicit Extraction

- DCC already tracking inter-frame transforms
  - Export to custom alembic attribute
  - 1000x Speed Up
  - 100% accurate, except artist error

Explicit Extraction

- Export transform hierarchy & geometry separately
  - Bind together during import
  - Allows independent iteration
  - Requires more custom Alembic attributes

Core

Mesh Optimization

- Condition meshes on import!
  - DirectX Mesh Optimizer, +3% GPU time
  - Added less than 500ms per import

Core

Packing for Runtime

- Importer generated:
  - Pallet of geometry parts
  - Transform per part per frame while alive
- Best packing strategy?
  - Minimize bandwidth!

Geometry Packing for Runtime

- Pack for optimal live range into single buffer
  - Sort by birth frame first
  - Sort intraframe by death frame second
  - Right edge moves ahead as parts are born
  - Left edge moves ahead as parts die
- Still can have dead part sandwiched between live parts
  - Removed via null transform

Core

Geometry Packing

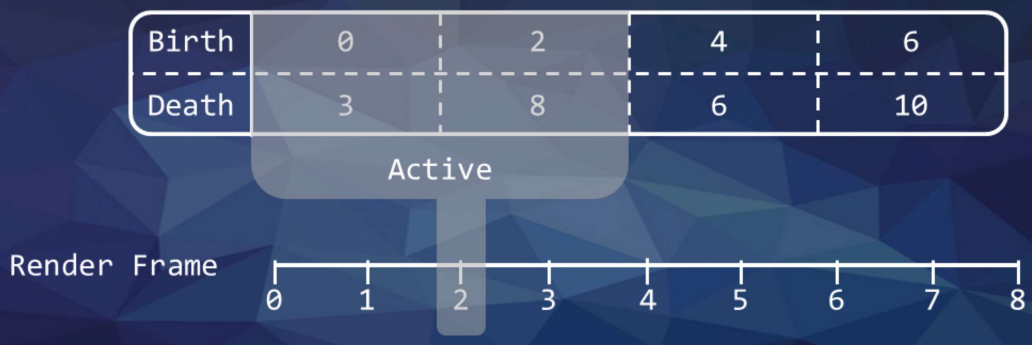
Birth	0	2	4	6
Death	3	8	6	10

Core

Geometry Packing

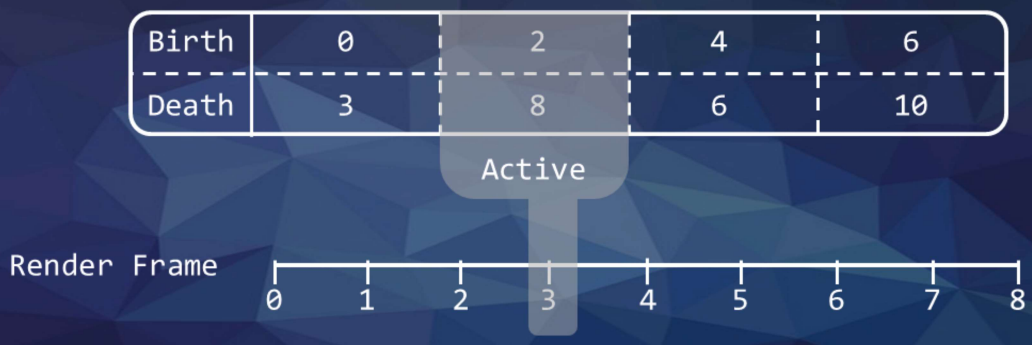
Core

Geometry Packing



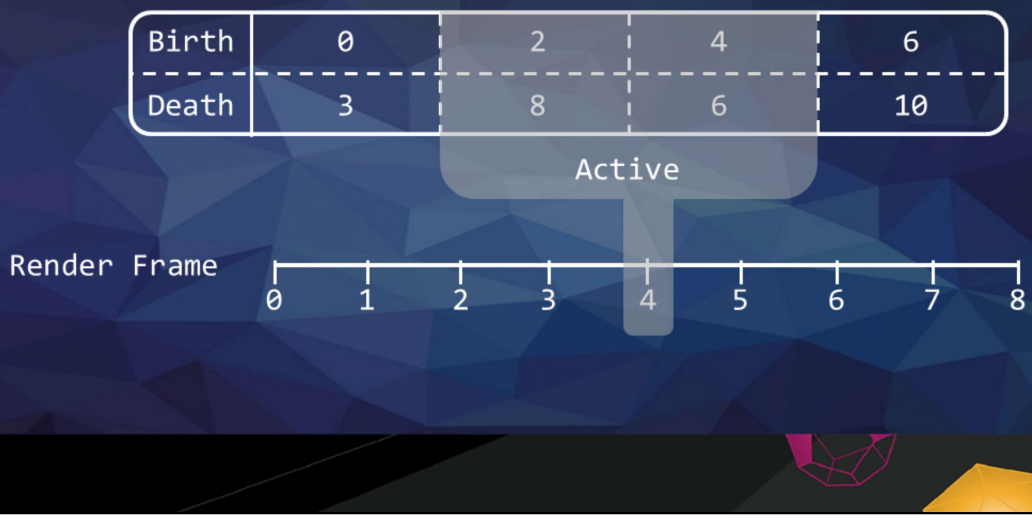
Core

Geometry Packing



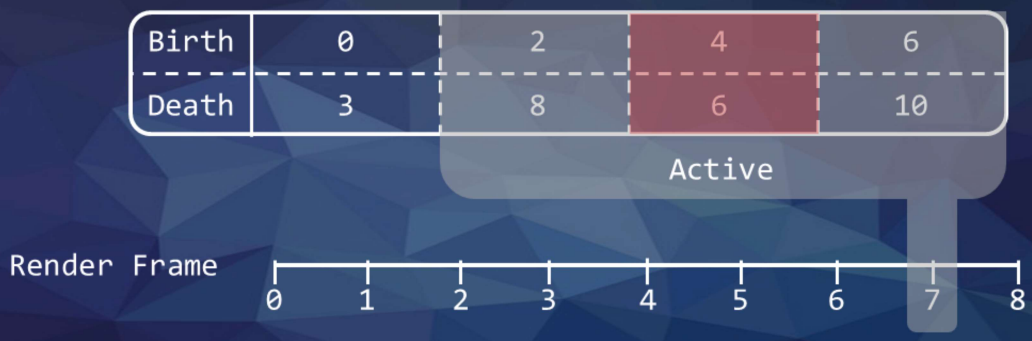
Core

Geometry Packing



Core

Geometry Packing



Transform Packing for Runtime

- ❑ Pack all active transforms per frame together
- ❑ Pack all frames next to each other

Birth	0	2	4	6
Death	3	8	6	10

Frame 0 Part 0	Frame 1 Part 0	Frame 2 Part 0	Frame 2 Part 1	Frame 3 Part 1	Frame 4 Part 1	Frame 4 Part 2

Core

GPU Packing

- Packing into 1 giant buffer
  - Binding offset'ed
- CPU vs GPU Endianness
  - D3D Structured Buffers will byte swap
  - D3D Byte Address buffer
    - Preserve bit for bit CPU order

Runtime

Core

Interpolation

- Lerp everything
  - Even quaternions
  - 30 -> 60, slerp = lerp
- Cuts bandwidth 50% or more
- Deletion/Creation rules can get hairy
  - Experiment, not 100% happy with our rules
- Not an option for Fluid Assets

Core

GPU Pipeline

- 1st approach series of compute shaders
  - Decompress & blend transforms
  - Transform verts
  - Compact & feed into render pipeline
- Killed bandwidth

Core

GPU Pipeline

- 2nd attempt Uber Compute Shader
  - Perform all stages
  - Sync using GroupMemoryBarrierWithGroupSync()
  - Utilize LDS memory
- Speed up around 3x

Core

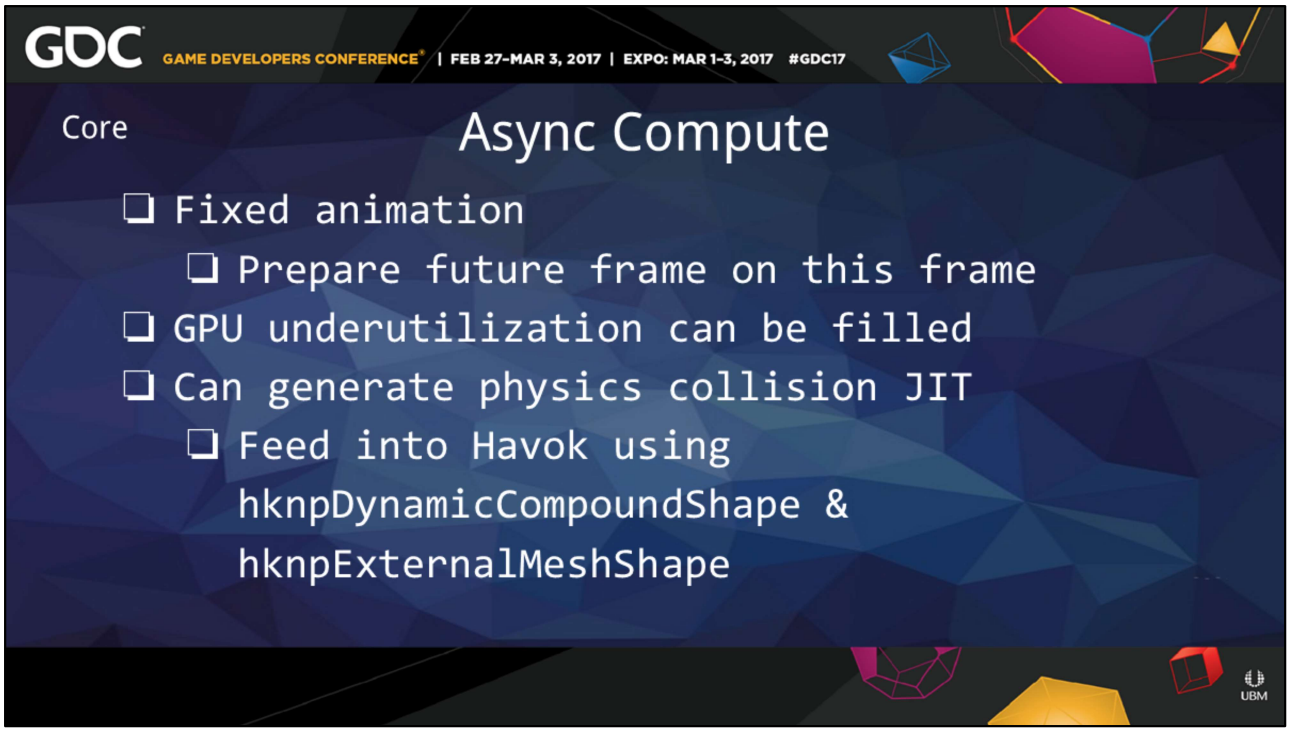
GPU Pipeline

- Transform decompression
  - Limited LDS
  - Active range of transforms limited to 64 per wave
  - Enforced in importer
  - Decompress, blend, and write to LDS

Core

GPU Pipeline

- Dispatch size selection
  - Use as large as dispatch possible without idle threads
  - Selected best 2^n based on max 64 transform per wave & vertices left to process
  - Almost always 1024 except for last wave



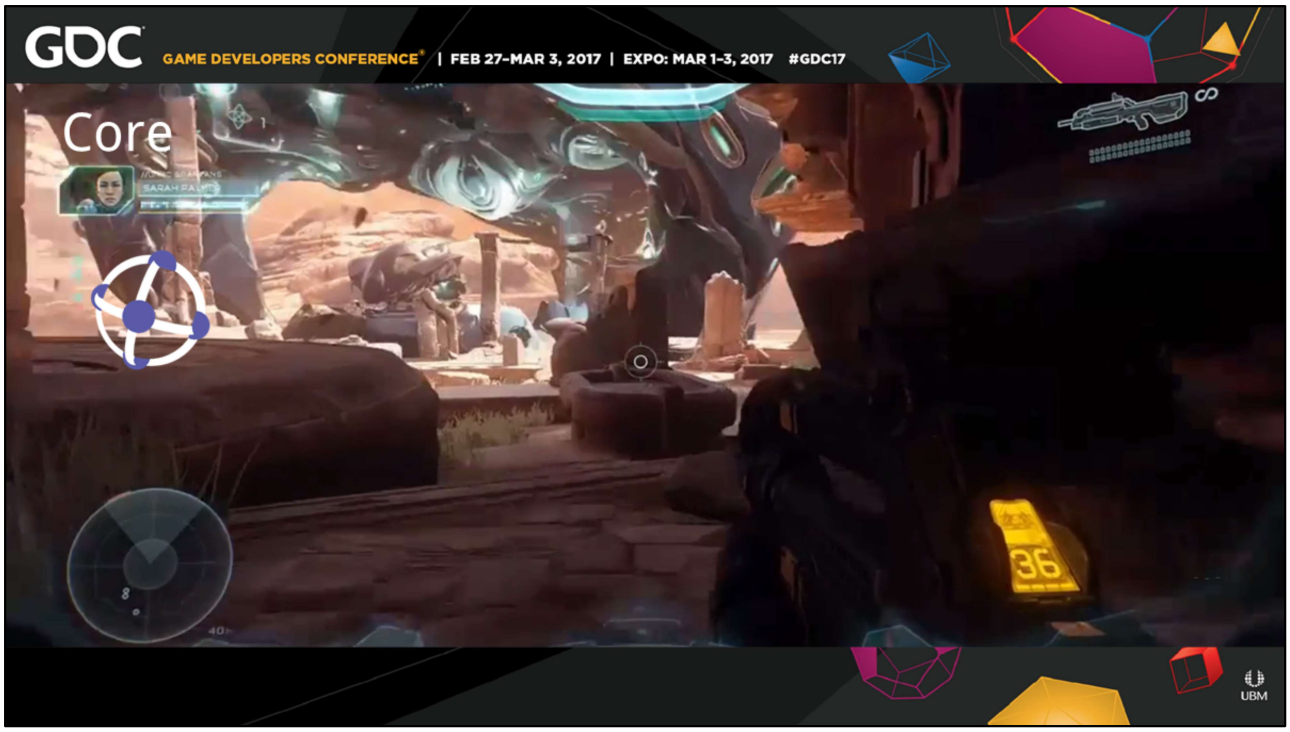
- 29:30, 01:00, 28:30
- +1:00 padding currently, we can add more if need be

- Since this is a canned animation Leverage XB1 DX11.x async compute
- Start generating next frames VB cache this frame.
- I didn't fully finish this before I left, but it was mostly working.

Also limit the render calls of small triangles, with this type of moving geometry you can have something quickly move from a hero object to off in the ackground, LODing will help will get to that kater, but this should be done in addition too.

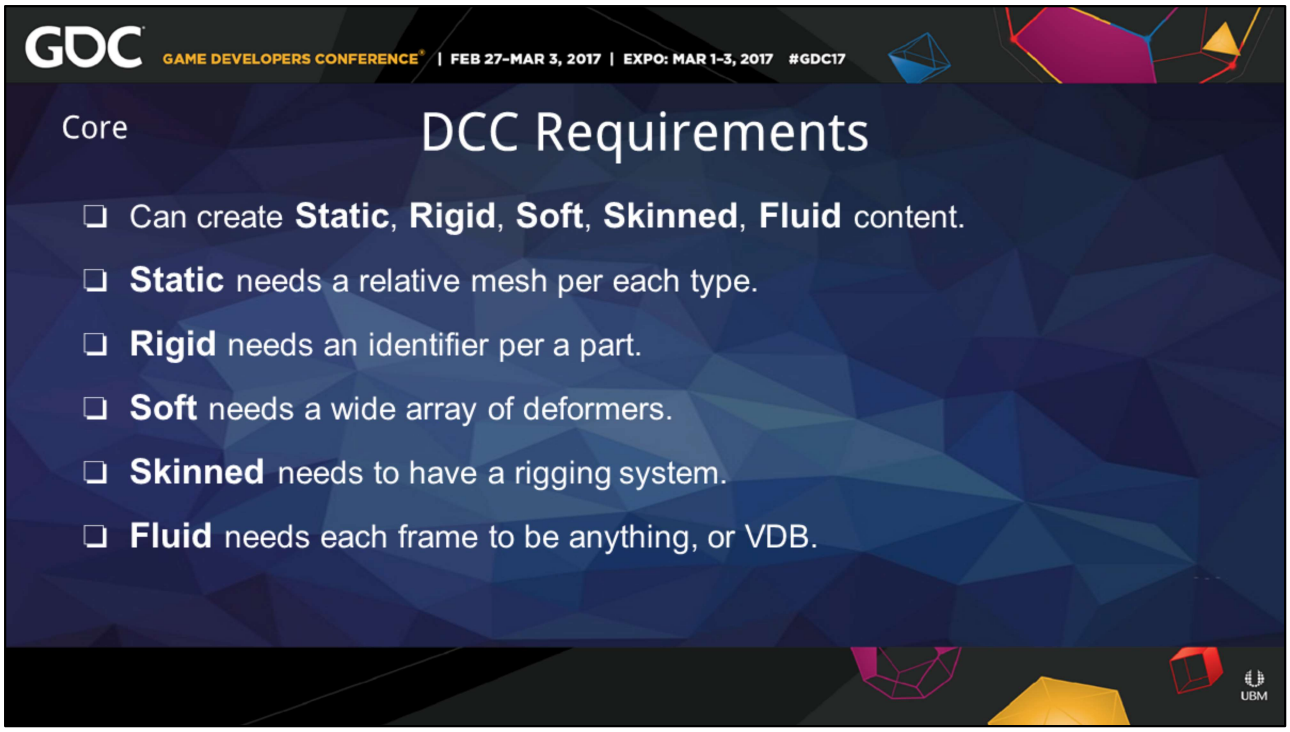
-16-20

```
[numthreads(GROUP_THREAD_COUNT, 1, 1)]
void TransformVertexPerPartCS(         const in uint3 Gid        : SV_GroupID,
                                                                 const in uint  GI         : SV_GroupIndex,
                                                                 const in uint3 DTid : SV_DispatchThreadID)
{
            uint grpFirstIdx = _xb_make_uniform(Gid.x << log2(GROUP_THREAD_COUNT));
            uint grpLastIdx  = _xb_make_uniform(min(grpFirstIdx + GROUP_THREAD_COUNT, VertexCount) - 1);

            uint grpFirstXFormIdx = VertexBuffer[grpFirstIdx].XFormIdx;
            uint grpLastXFormIdx  = VertexBuffer[grpLastIdx ].XFormIdx;

            if (GI <= (grpLastXFormIdx - grpFirstXFormIdx))
            {
                        LDSPartXForms[GI] = CalculatePartTransform(grpFirstXFormIdx + GI);
            }

            GroupMemoryBarrierWithGroupSync();

            if (DTid.x < VertexCount)
            {
                        PartXForm partXForm = LDSPartXForms[VertexBuffer[DTid.x].XFormIdx - grpFirstXFormIdx];
                        TransformVertex(partXForm, DTid.x);
            }
}
```


GDC
GAME DEVELOPERS CONFERENCE | FEB 27–MAR 3, 2017 | EXPO: MAR 1–3, 2017 #GDC17
Core
SARAH PALMER
36
UBM

- 31:45, 00:45, 26:30

Kraken Reveal



    - 32:30, 01:00, 25:30
- At this point we are going to switch over to the **authoring side** for a bit.

- There are many **content creation softwares** you can use. We used **Houdini** for its procedural nature,

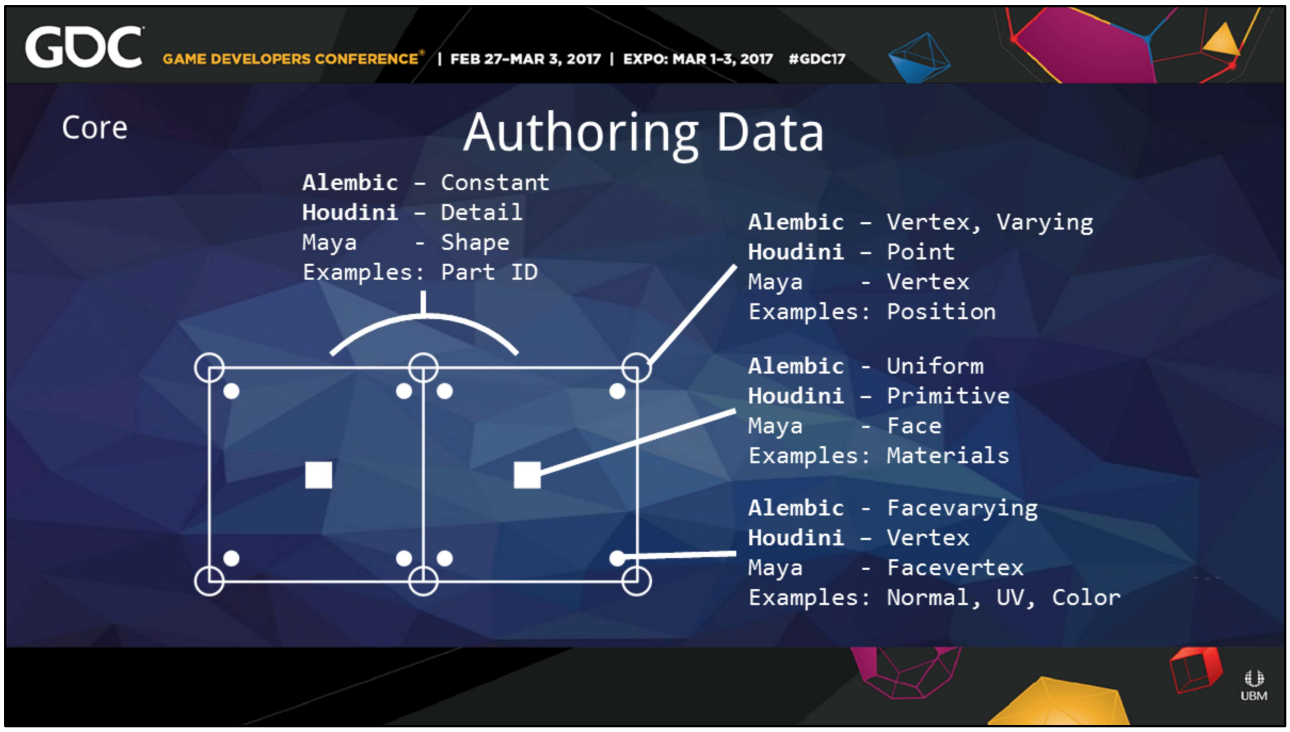
However at each studio, you have **your own** pipeline so keep these issues in mind.
Static needs to handle a series of motion dependent procedural **culling** methods
Rigid's needs an **id per a piece**, In Houdini we used the name attribute
In Maya you can use an id for each bone, or shape node
Plus you need access to as many simulation, particle, and animation tools as you can get your hands on.

- For **soft** you need a series of **deformers**, the ability to ray cast, and a flowing VDB mesher.

Skinned certainly requires a **rigging** system. Generally the more procedural rigging you can do the better.

- Finally **Fluid** needs the ability to change the geometry **every frame**. This is more than just VDB remeshing.

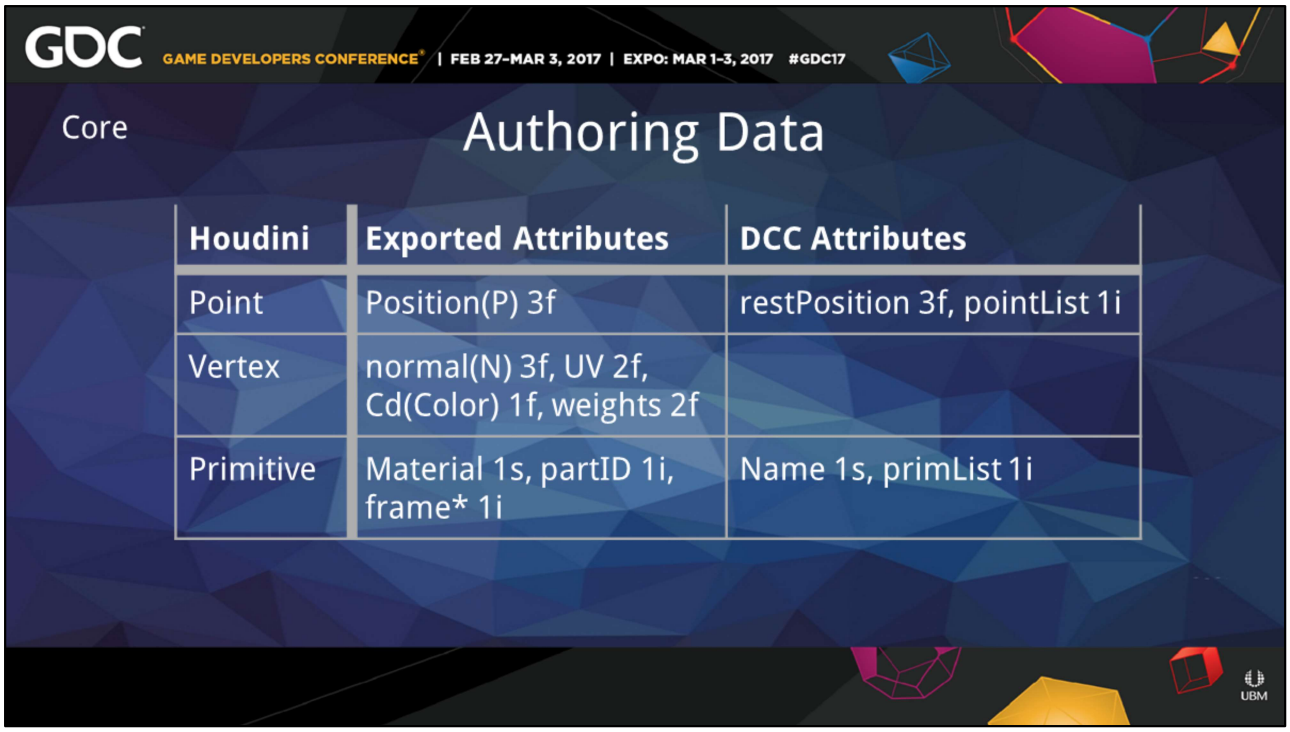
You need the creative ability to morph to any shape.



    - 33:30, 01:00, 24:30
- Regardless of your authoring packages you need to **maintain** specific attributes in your workflow.

I want to point out that, Houdini and Maya have two **different sets** of terminology for data levels

And your intermediary file will have it's own. This **screwed us** up, so be careful.

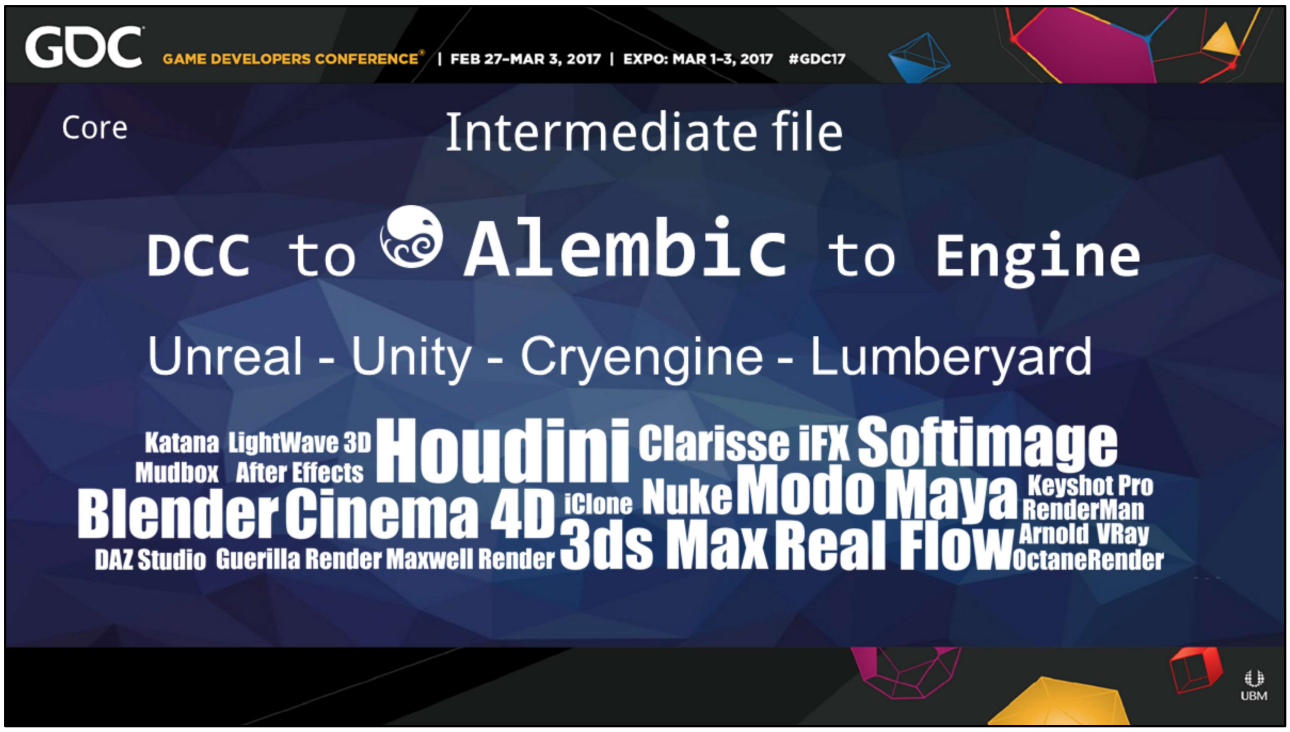


- 33:30, 01:00, 24:30

The data to be **aware** of is:
*__Position__ - your basic XYZ location per a point
*__RestPosition__ - it is like your T pose for that geometry
*Plus you need to maintain an **ordered list** of points,
*so that for export your point order over time remains constant

*__UVs__, __Normals__, Color, and Skin Weights
*are regular attributes you need to maintain
*Along with **materials** at the Primitive level
*__PartId__, or string based Names, are used to identify the parts per a transform
*And maintain a **primitive list** similar to your point list in order to maintain the data hierarchy for export

*One thing to be cognisant of, is some of these attributes can be maintained at **different levels**
*Such as, materials so be aware of that for when you export your data

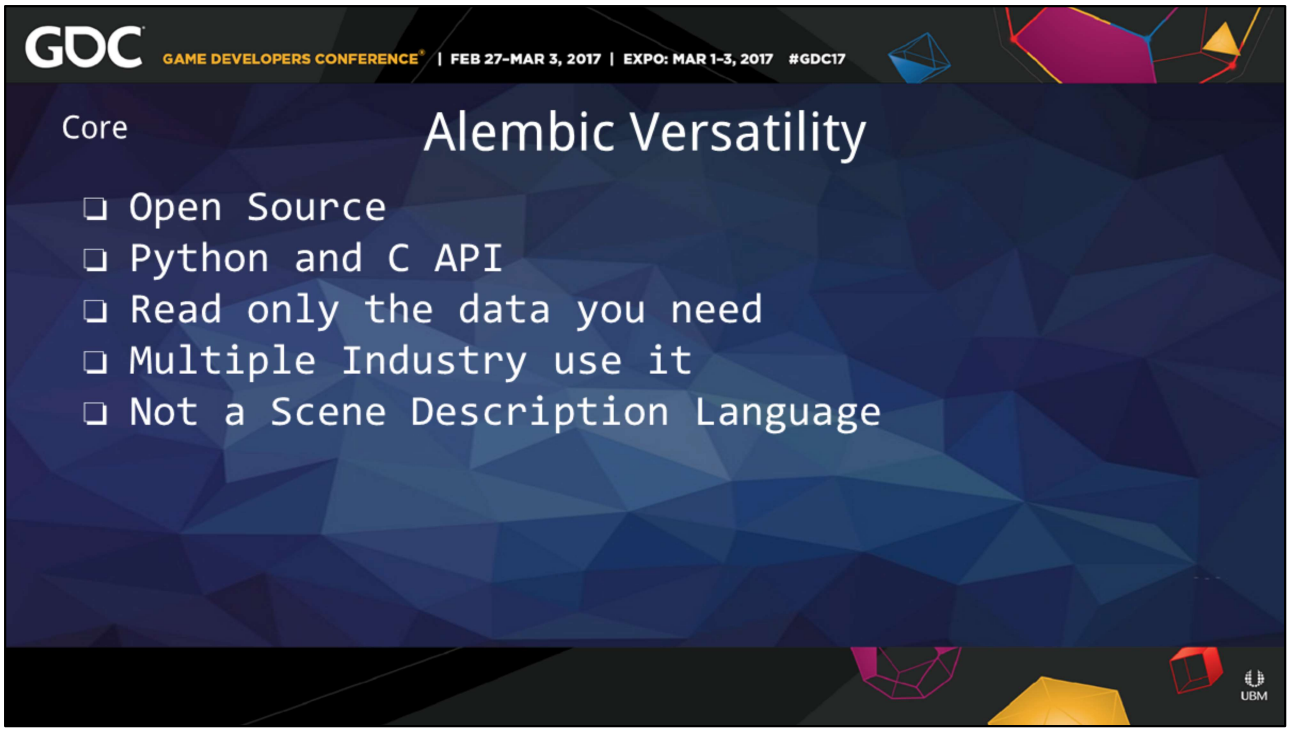


- 34:30, 00:30, 24:00
- Coming back to the Intermediate file formats, we use, and strongly recommend **Alembic**

- We did not use, nor recommend **FBX**, for this pipeline,
With a big **asterisk** for using the **skin** method

- **Alembic** is **used** across a host of major content creation packages,
and has some practical inputs into a lot of off the shelf engines nowadays.

- We use it because it's **fast**, Maya even uses it for it's Geometry Caching.
I **do not** recommend using it **directly** in your engine itself, thus our talk,
but technically you could for Fluid.



- 35:00, 00:45, 23:15

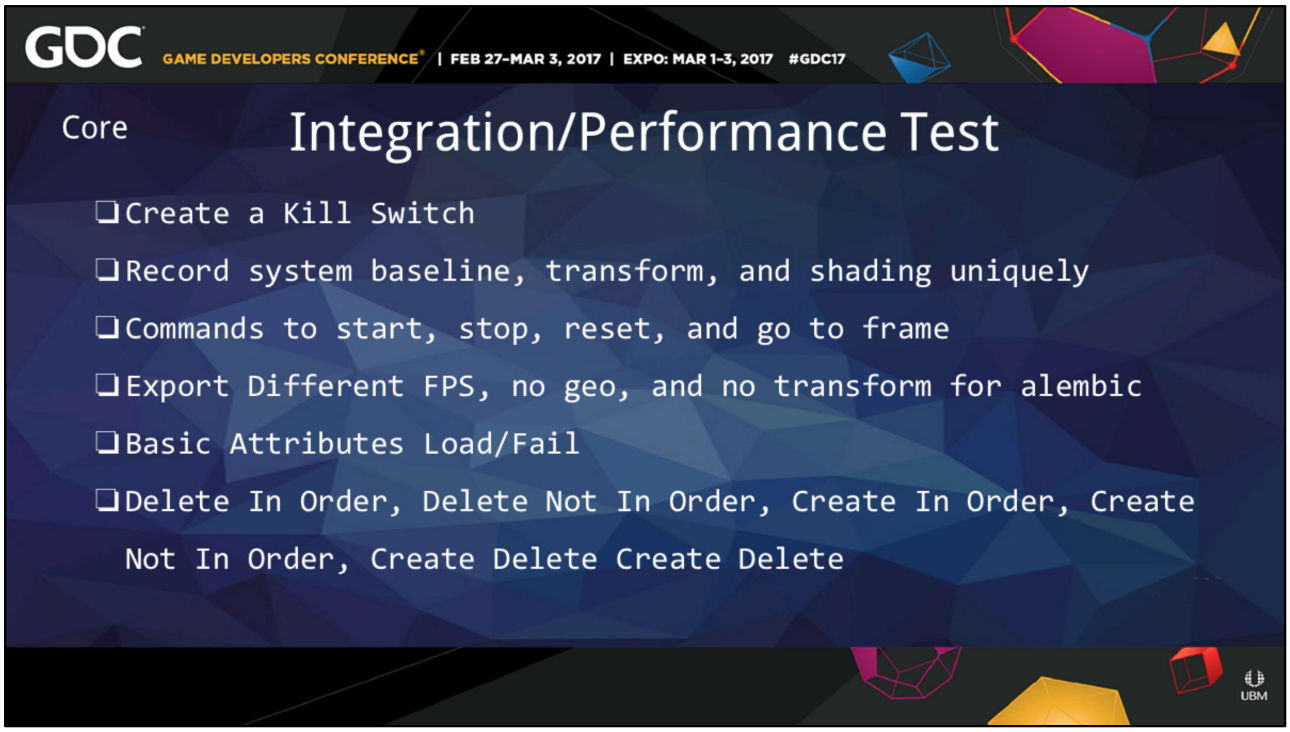
- Let me hit a few **major points** on Alembic. First it is **open source**.

You no longer need to maintain .**xml** files in your pipe.
Nor do you have to **precondition** your scene file before export.
I've **heard** multiple times, "It's pleasant to work with"
It's not a black box, it has python and C bindings so you can access anything.

- You will **not** be using it **alone**. The VFX industry uses it, too. Even the scientific community uses it…

- I will admit it is not a **scene description** file format, like USD, FBX, or Collada

But the beauty is, it's not trying to be. It **handles geometry** and it does *bleeping* well.
For sequences of geometry the file format is bigger.
However, you can read only the data you need, on the frame you need.
You don't need to load a 3,000 frame sequence, to read one frame and one line of data.



- 35:45, 01:15, 22:00

- To **wrap up** this core architecture section, I want to hit up one last topic. This is **integration** and **performance testing**.

You will want to run these test **every time you update** the code, or your build.
Geometry Caching is a tiny part of your engine. And it shares most of it's time with many other systems.

- Make sure you build the system with a **kill switch**, for the entire system, for de-bug-ing

Being the new kid on the block, you'll get picked on and blamed as the culprit a lot
You'll want to make sure you fix the right issues

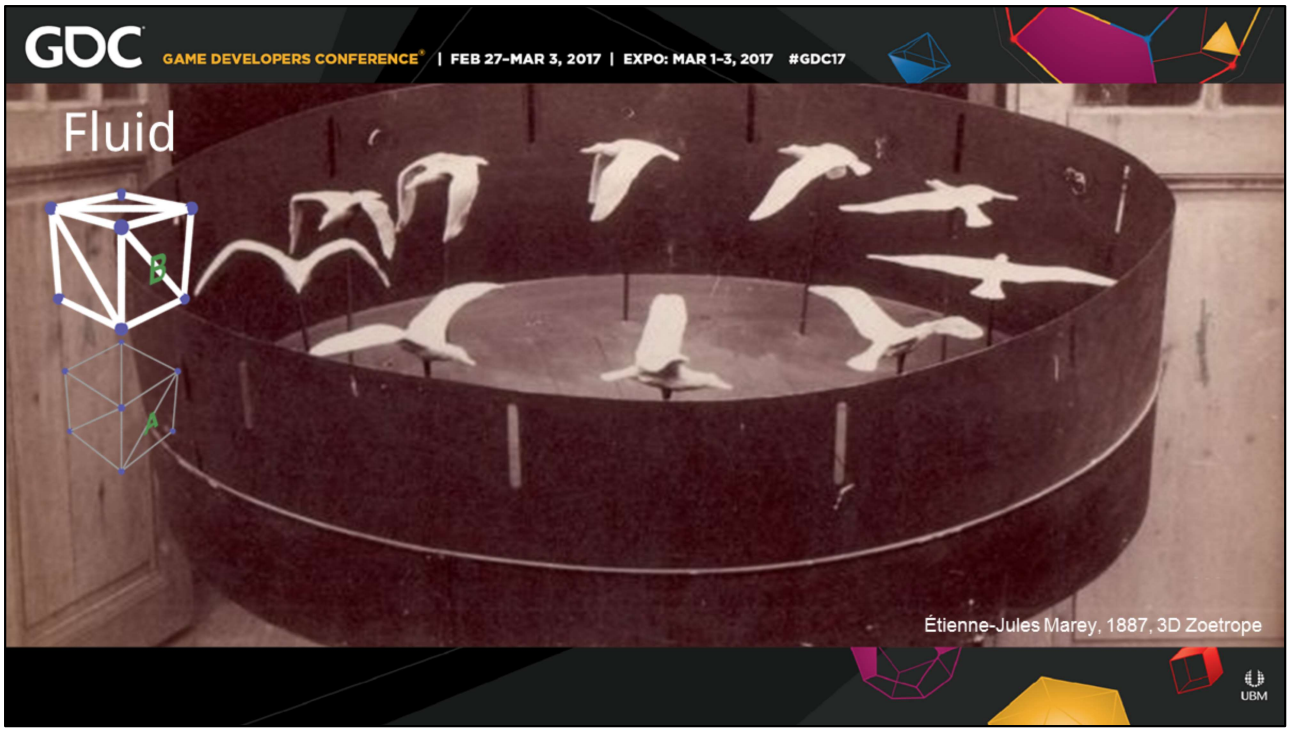
- As for **alembic** and your **export/import** method

Make sure to test with different frame rates, with no geometry or transforms, and with or without attributes.
Alembic will **restructure** itself for the best storage method.
And you want to avoid issues related to cross program terminology, collapsing alembic files, and artist screw ups.
As noted before "**deletion**" i.e scaling to zero, can get hairy with interpolations
so test these series of deletion and creation options.

- As for the actually **microsecond**, pix style measurements, understand your **baseline**

when our system was active, but not transforming, it was 2 microseconds,
Record your transforms calculations and shading uniquely, together these are the net changes on you system.

Since this is a sequential asset, make sure you have run **commands for start**, stop, reset, and go to frames



- 37:00, 00:30, 21:30

- Ok everyone **take a stretch** for a minute!

::PAUSE::

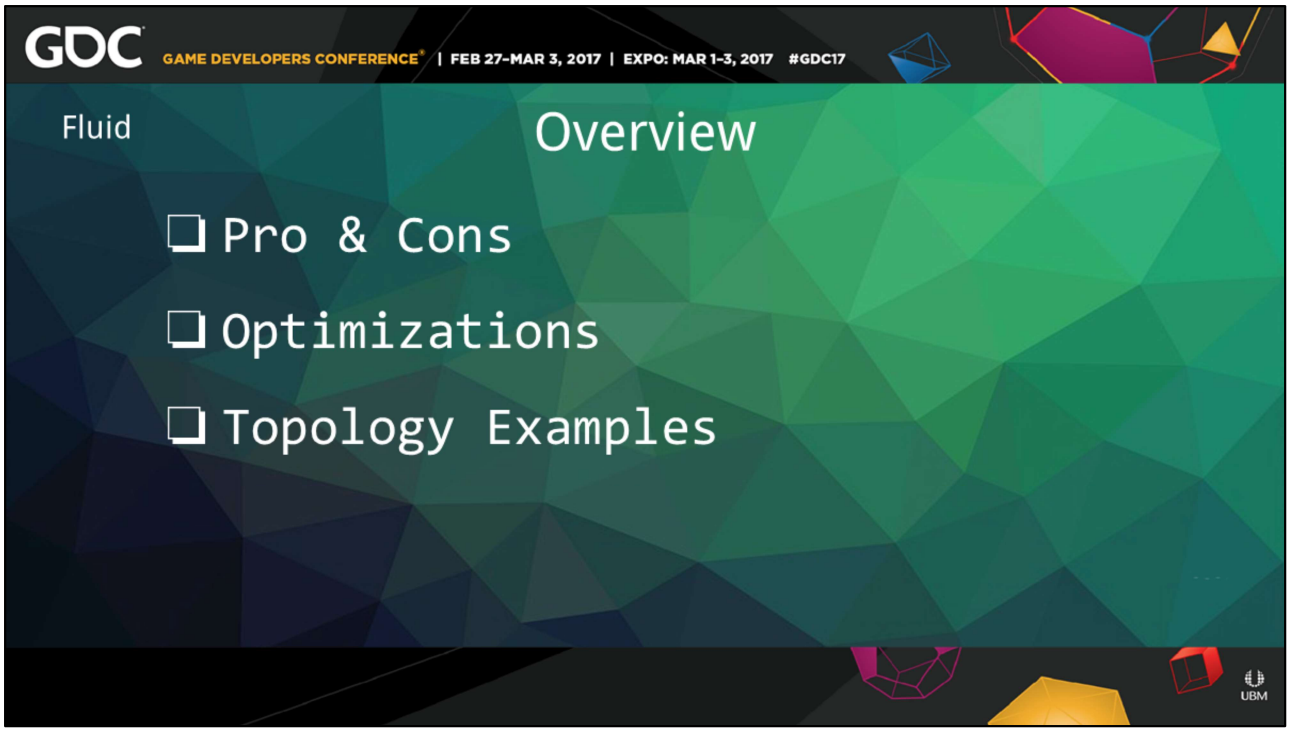
We are going to start diving into the **component systems** now.

- To prepare yourself for **Fluid** it is best to think of this component type as a **3D Zoetrope**

These poses of a **bird flapping** are spun around,
and as you look through a slit you perceive a bird in flight...

- This concept was thought of over **130 years** ago.

Today we use this concept with VDB meshing, and other artistic motions.



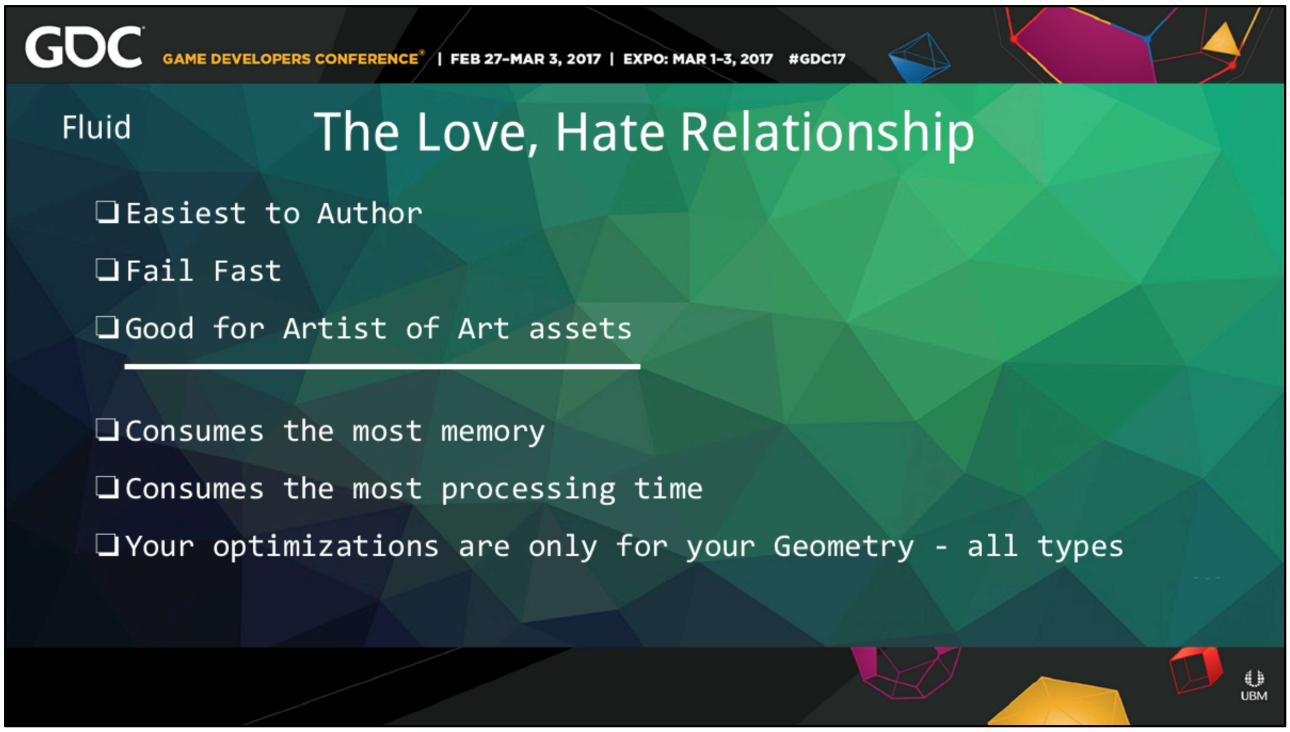
- 37:30, 00:15, 21:15

- I have an extreme **love hate relationship** with fluids,

So that is how we are going to define our Pro's and Con's

- It's usually the first component type implemented,

So it has it's own **unique optimizations** that will help



- 37:45, 00:45, 20:30

- **Fluids** were our **first love**, or really the first relationship we had. We thought fluid were going to be our one true love.

And as it turns out it was more of an **infatuation**, a learning experience to say the least.

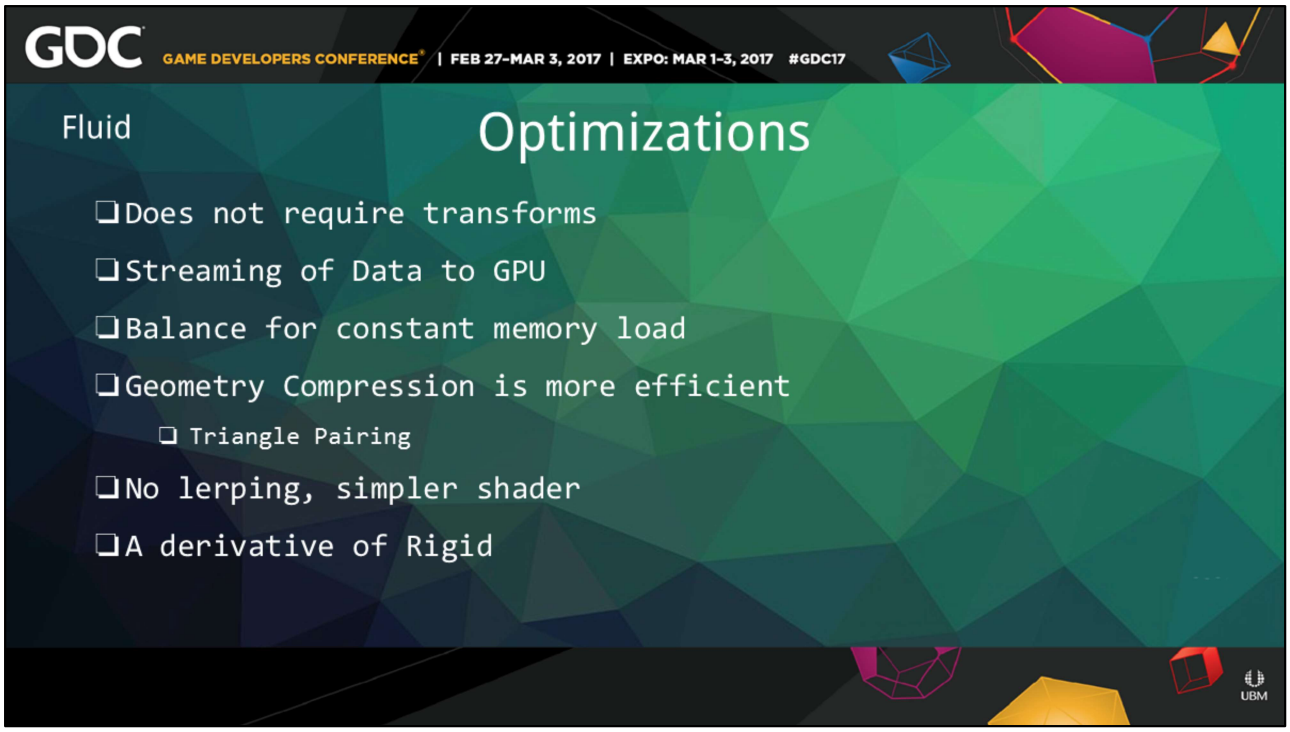
- It is the **easiest** to export, as each frame has no history with the last.

It allows you to **rapid**ly prototype, and get it in the game.
And there is no limit on the type of asset you can export.

- However in games the **microseconds** matter. and per frame changing topology is **expensive**

Fluid consumes a lot of memory and processing time.
We found anything we did, that we **thought** needed to be fluid,
Could be done be done with rigid or soft, and put into a sequencer.

- However, this is not a reason to **discount** fluid. It has a time and a place for art and for rapid prototyping. Plus it can be optimized far greater than we did.



- 38:30, 00:50, 19:40

- In order to optimize **Fluid** further Fluid Topology inherently does not need **transforms**,

so don't include them in your data components
In large rigid assets this is where all the bloat comes from with zero transforms.

- Subsequently having a **good streaming system** to the GPU matters,

There is no inherent connections between each frame, just a very predictable read from memory.
So the shader does not need to deal with interpolation

- This is similar to an **alembic** file, however you may be tempted, don't use alembic directly in your engine

All the previous core optimization would be thrown away.

- An important thing to mention is we **did not over optimize** Fluid,

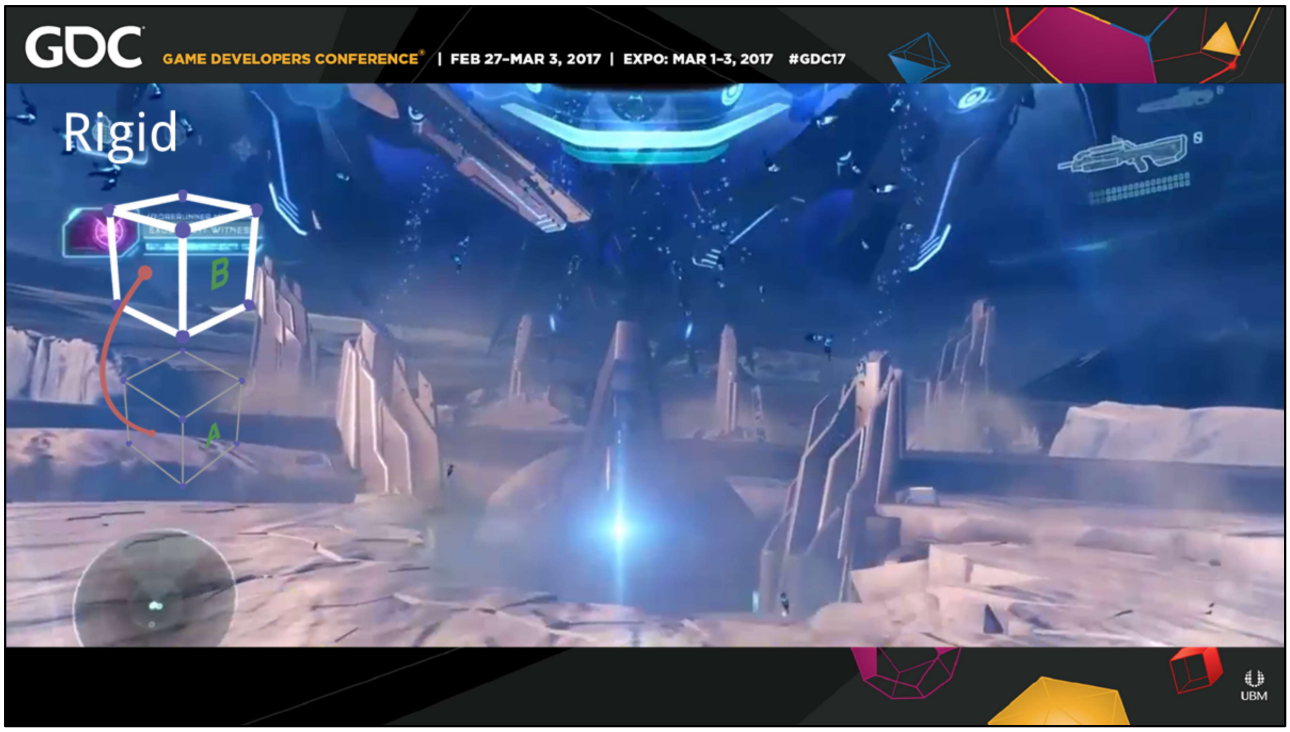
Once we started using Rigid and Soft we stopped researching Fluid
So there can be more hybrid methods out there.
I'll reference two talks at the end, at this GDC alone, that can help.

- 39:20, 00:20, 19:20

- Before we exit from Fluids, I wanted to show one **example** of the topology of fluid.

- Notice the **changing topology** of this fluid mesh

as will come back to this shape **later** to **optimize** it.



- 39:40, 00:20, 19:00

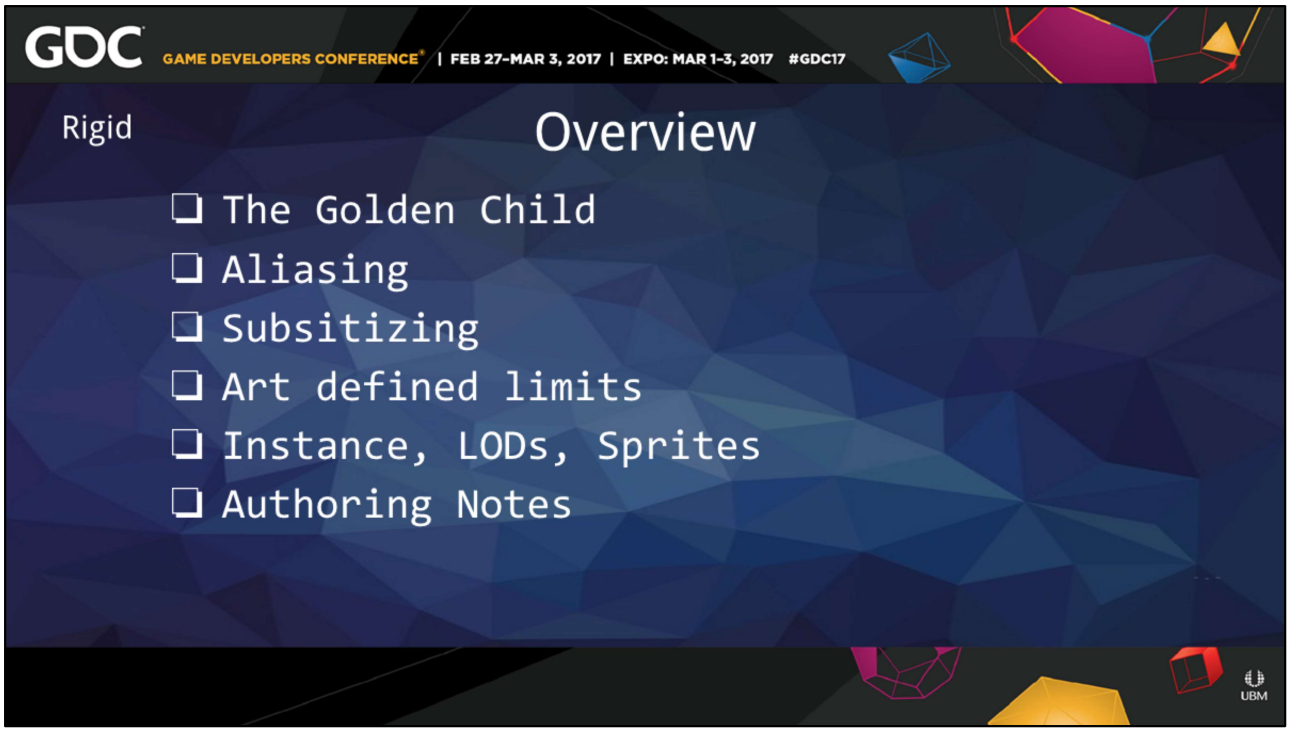
- Moving on to **Rigid**, here is a video of where we strictly used our rigid method at the end of the game.

- The funny things is we **originally** had an even **larger swarm** on the screen, but our outsourced vendor whom did the sequence after ours only showed a small fraction of a swarm at end

- So we had to significantly **reduce** the amount we had on screen.

- 40:00, 01:00, 18:00

Halo 5: Guardians End Game



- 41:00, 00:20, 17:40

- With Rigid, I want to cover the **benefits** of the golden child, Plus some of it's inherent **problems**,

How we use **subsitizing** to simplify our authoring for rigid
Why we have some of our art defined programming **limit**
And how to maximize rigid more with **Instances**, **LODs**, and **Sprites**
and a few additional **authoring notes**

    - 41:20, 00:45, 16:55
- **Rigid** hands down was our **best method**, we used it for everything from building debris, to flocks of birds, to leaves blowing in the wind.

- We even realized we could leverage it in places **traditionally** reserved for our fx, flocking, and animation tools.

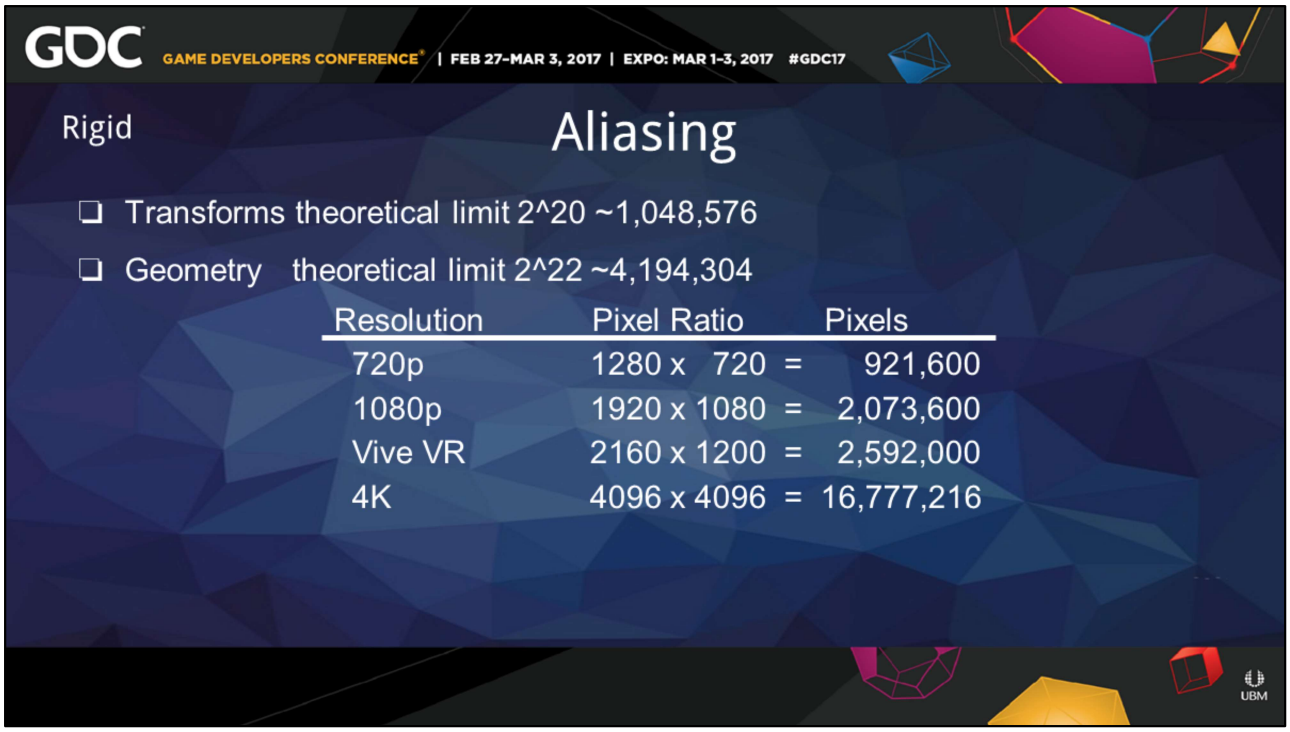
If those assets did **not** need to **interact** with the player, our system had a significantly **reduced overhead** in comparison.
This way the prepared assets would not get cut and breath more life into the level.

- We could reduce birds to a few polygons, and send hundreds, or thousands flying through the air.

It's a **trade off**, but it allowed us to retain **cheap animation** in the scene,
and use those interactive technologies **where it mattered**, on the ground or in your face.

- It's important to remember this is **not** a **particle** or a crowd system, nothing runtime interactive is happening,

However, if you **componentize this code** you could use it based on **event triggers**
In essence what we are doing is trading memory for runtime cycles,



- 42:05, 00:50, 16:05
- This is a good time to explain about the **viability** of having over a million transforms at 60 frame per second.

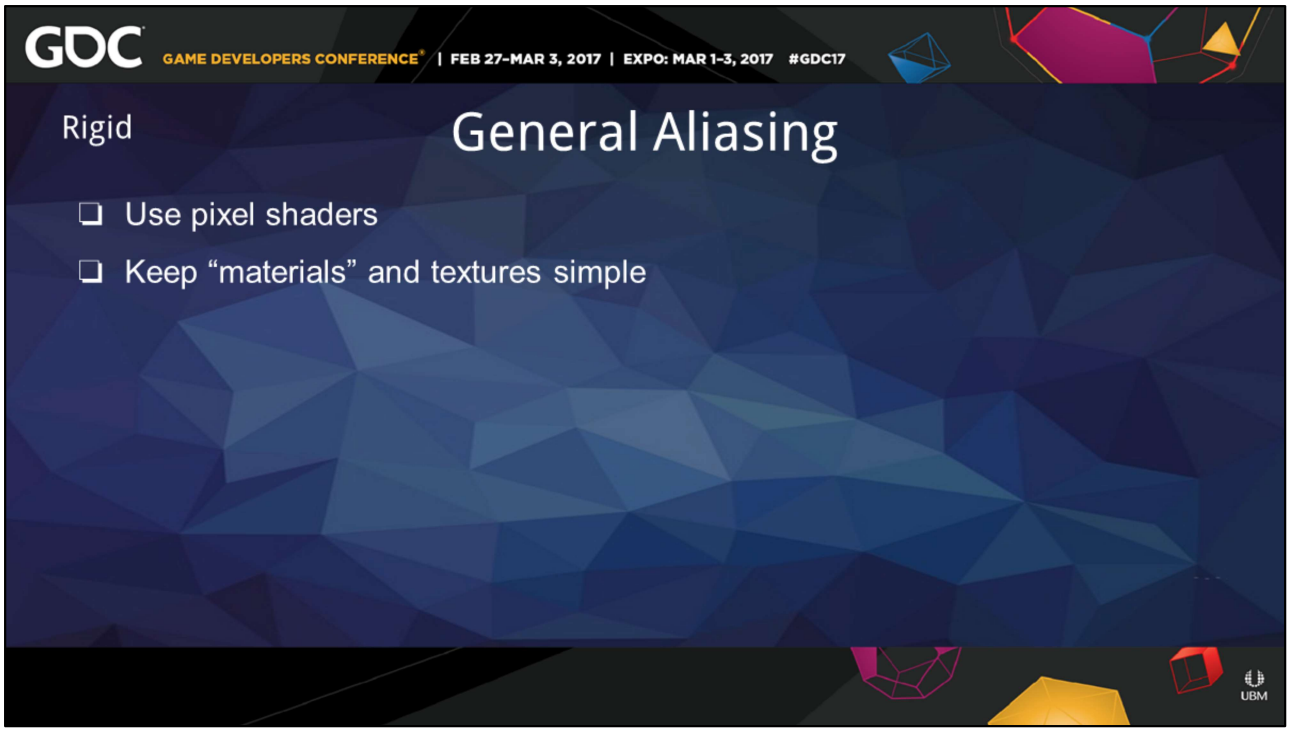
The million transforms was actually a **test** to see the **upper limits** of our system, with nothing else in the scene.

- The key thing we found out is, there are **only so many pixels** on the screen at 1080p

Unless you are at 4k can you really comprehend the different scale factors of increased transforms beyond a hundred thousand
We are not trying to compete with **Krakatoa**, this is kinematic baked motion in a AAA game.

- In production we never came close to **needing** this.

However what this does mean is that when we show a percentage of this many parts
Those pieces are moving extremely **efficiently, fast,** and **cost** us very little



- 42:05, 00:50, 16:05

- For **general aliasing** issues.

Keep your materials and textures simple if you are pushing a lot of geo the beauty is in the complex motion.
A pixel shader will suffice with over a hundred thousand transforms on screen.
Plus you can just use a ribbon.

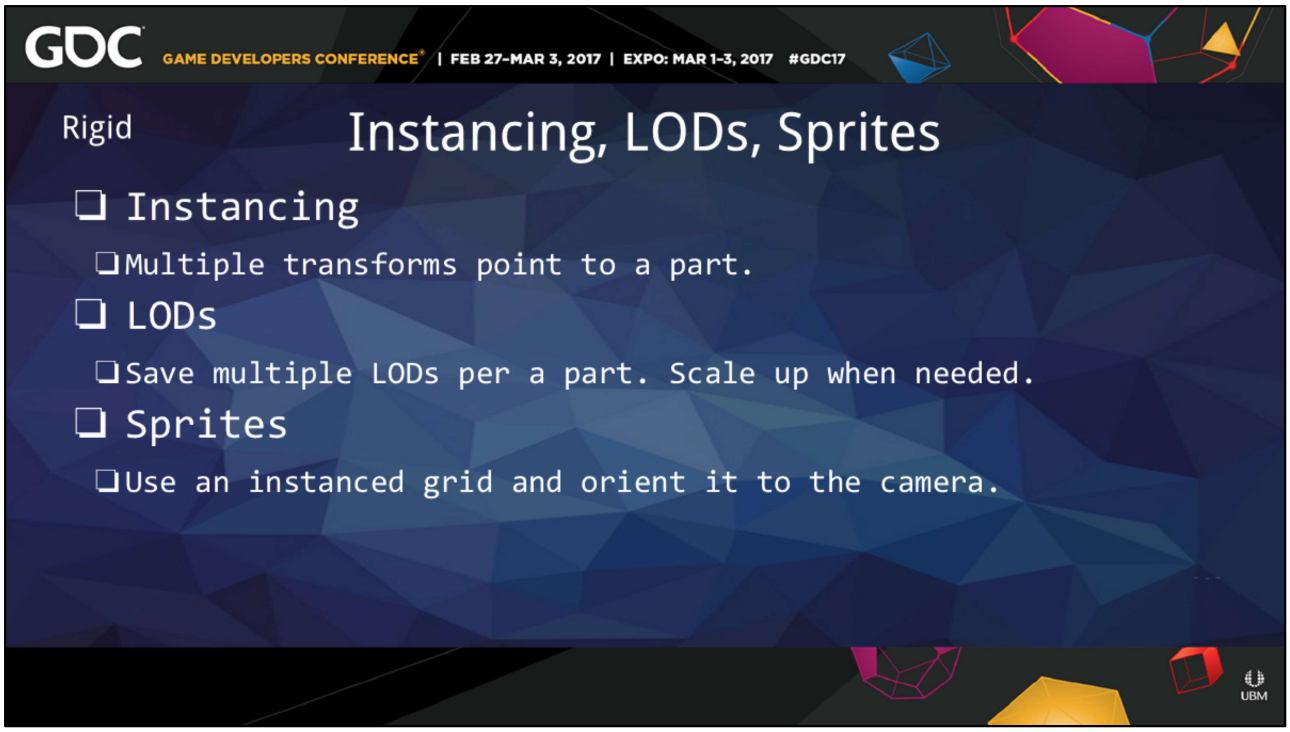
- 42:55, 00:30, 15:35

- **Subsitizing** was one of the key methods we used to reduce complexity with a significant number of transforms on screen

- Subsitizing is the reason why phone numbers are broken into patterns, Humans have a hard time comprehending around **7 digits**,

- Try and quickly **count the stars** on the screen for instance.
::pause::
It is 10.

- If you have ever built **crowds**, you learn 3 is a base for variation and 8 is more than enough.
So with a handful of unique models we had more than enough.



- 43:25, 00:50, 14:45

- Which leads us into our next topic, **instancing**, **LODs**, and **Sprites**

With Rigid we really just have a **keyframed** based **transform cloud**.
So you can put anything on those transforms.

- You can **instance** from a common GPU memory buffer across all those transforms

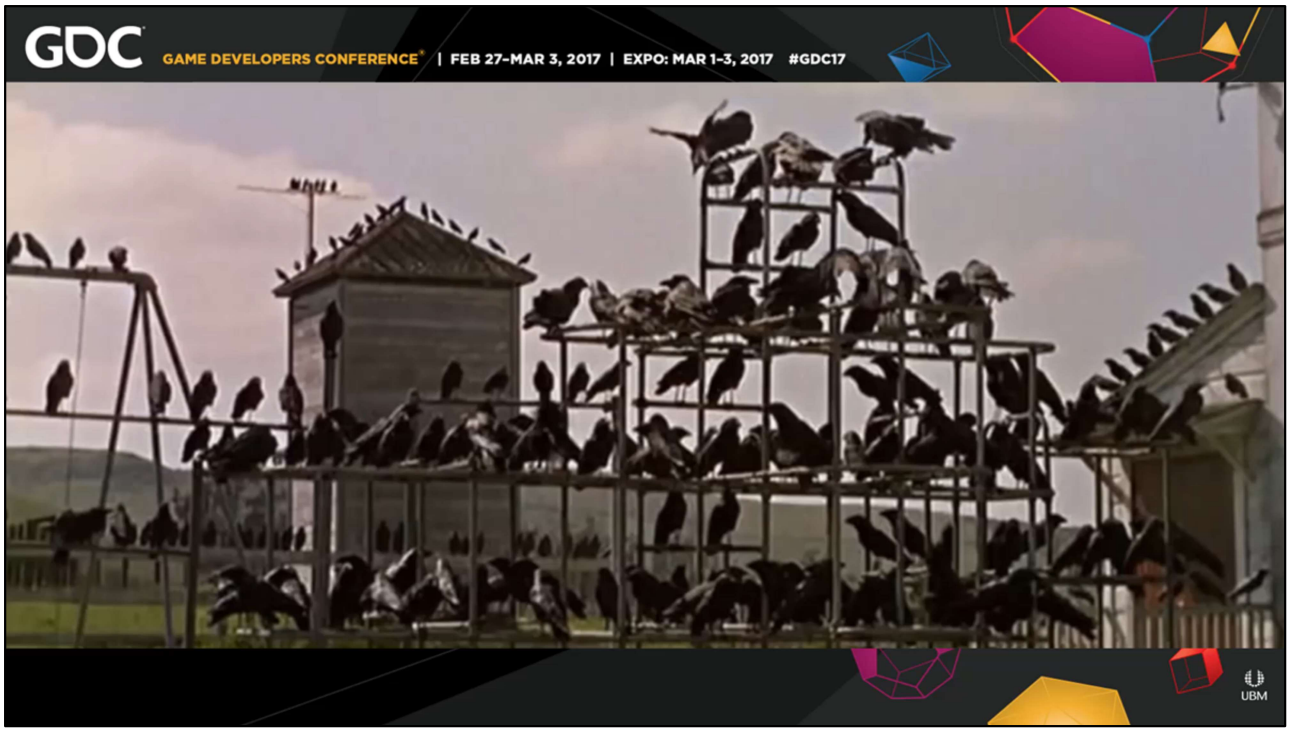
to reduce your disk foot print, and the amount you stream to the GPU
Render time does not go down, but it's cheaper on disk than 30,000 of the same model.
Plus it reduces the memory bottle neck coming from the hard drive.
Unfortunately we did not have this setup for those swarms in fact, and it cost us alot.

- You can even do **level of detail**, by rendering only select models from the common memory buffer, or

You can use the scale to zero method we used for deleting based on the zdepth.

- You can even use **sprites**, or billboards, with this method, By unwrapping an instanced grid, towards the camera.

These all require a more **complex runtime shader**, but the IO and streaming benefits are heavily worth the effort.



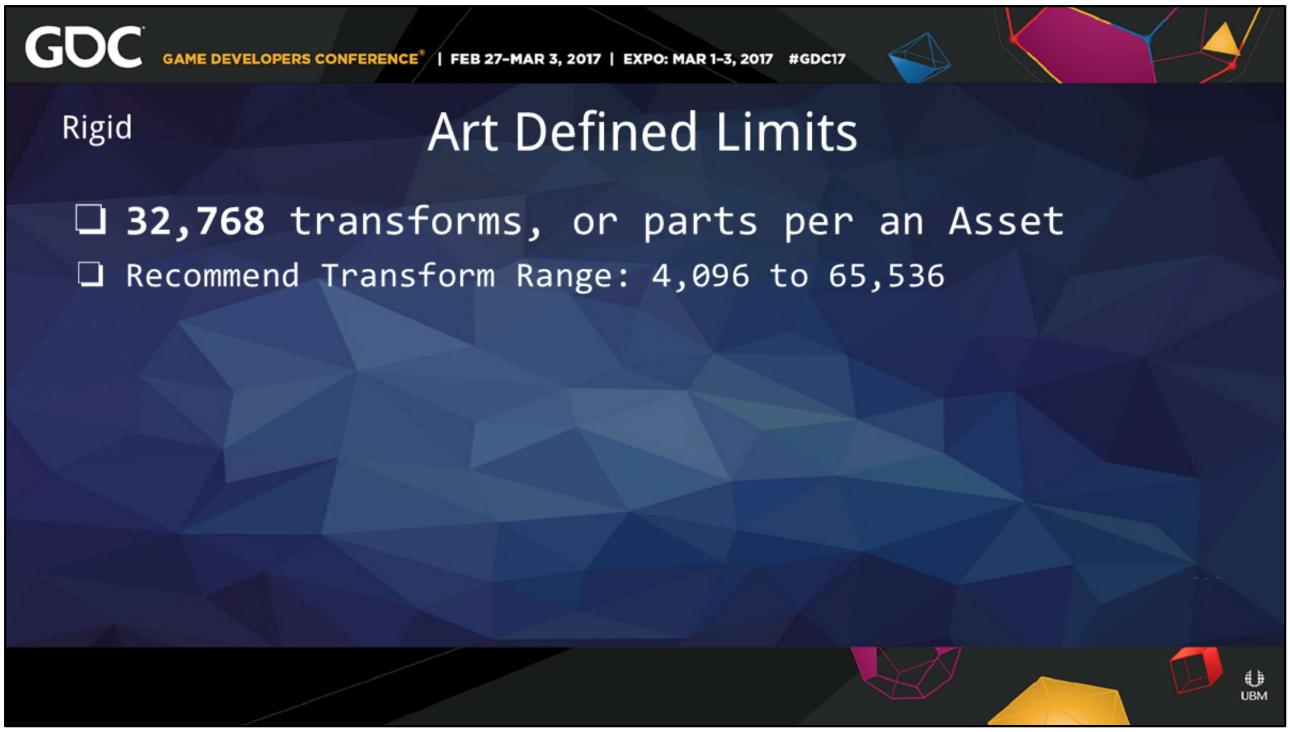
- 44:15, 00:25, 14:20

- To wrap up on the shear scale factor,

I'm going to draw reference from Alfred Hitchcock's "**The Birds**" filmed not too far from here.

- This was filmed in **1963**, way before computer graphics in FX

Only a handful of those birds are alive, the rest are just static props
But playing with human perception we can use that to our advantage.



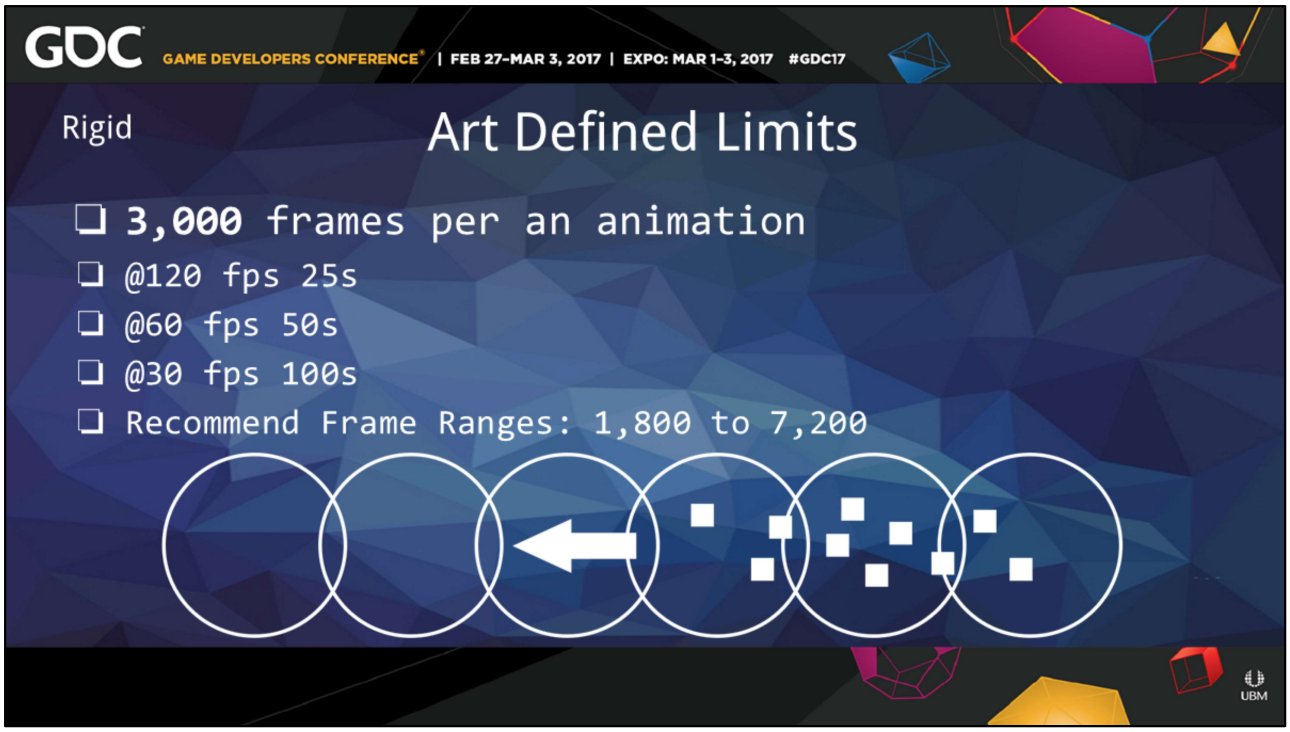
- 44:40, 00:30, 13:20

- Zabir mentioned our limits already, so I'll be succinct.

Each **asset** has it **limits**, and the artist should be aware of this,
as they are in the best position to know what transforms or pieces of geometry to cut.

- **32 thousand transforms** was good for our larger building assets we destroyed.

However a majority of our assets used half as many transforms
So for each studio and game type you may want to limit
at between **4k** to **65k** transforms



- 45:10, 00:30, 12:50

- Additionally we limited the transforms **duration** based on screen time.

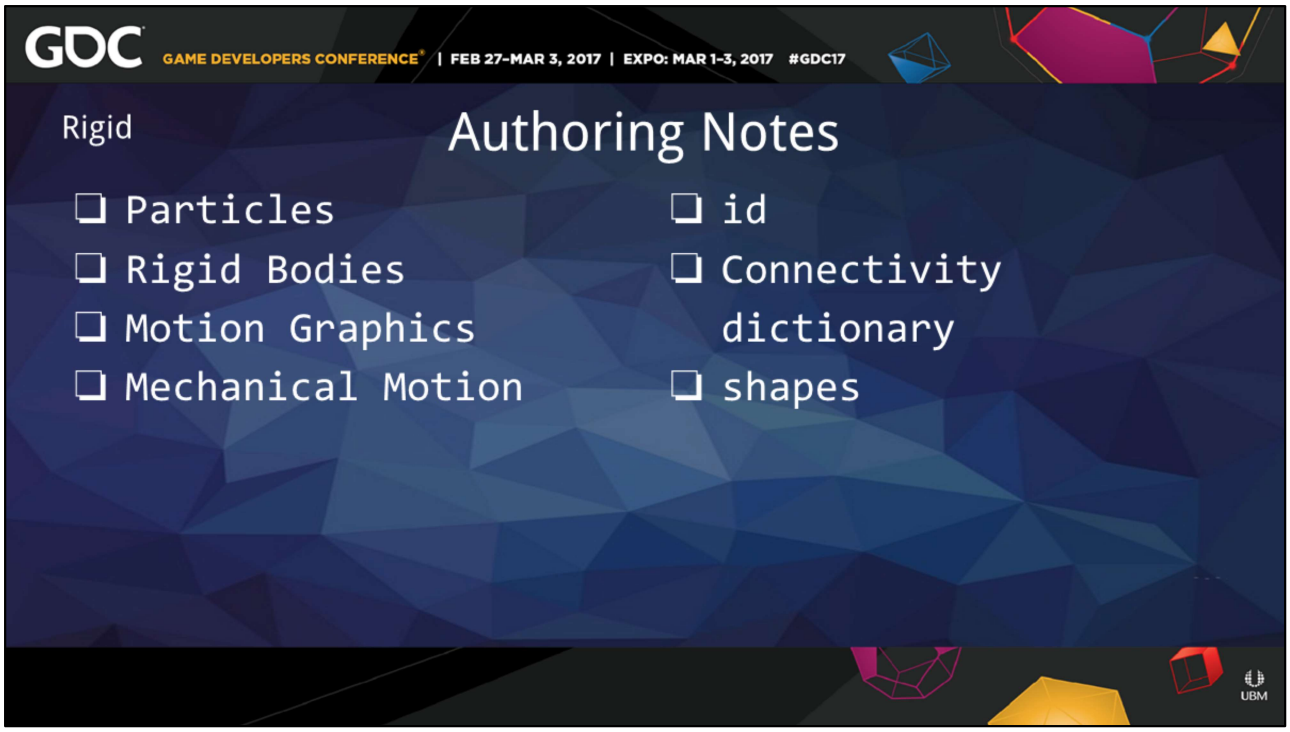
However, when a flock of birds has to cross a whole level we did run into these limits.

- However a single asset across a level makes **culling** an **issue,** but you can actually centipede these animations,

by splitting it into multiple assets, qued in a sequencer to delete and create from one section to the next.

- So these time limits weren't truely the issue it was distance.

You can handle this under the hood, but we would at least recommend to keep the frame range from 1,800 to 7,200
to best compress your data.



- 45:40, 01:00, 12:20

- A few last authoring notes for rigid. Rigid has **numerous** ways to author, Including **particles** we used to authored the swarms, **Rigid Bodies** were used in the Kraken Reveal and a lot of E3 trailer.

-The conversion of high quality **animated mechanical** motion, like Shon Mitchell did on our team.

- An important note is that **maintaining** over time a **part id**, per a transform, in an artist based workflow can be complex.

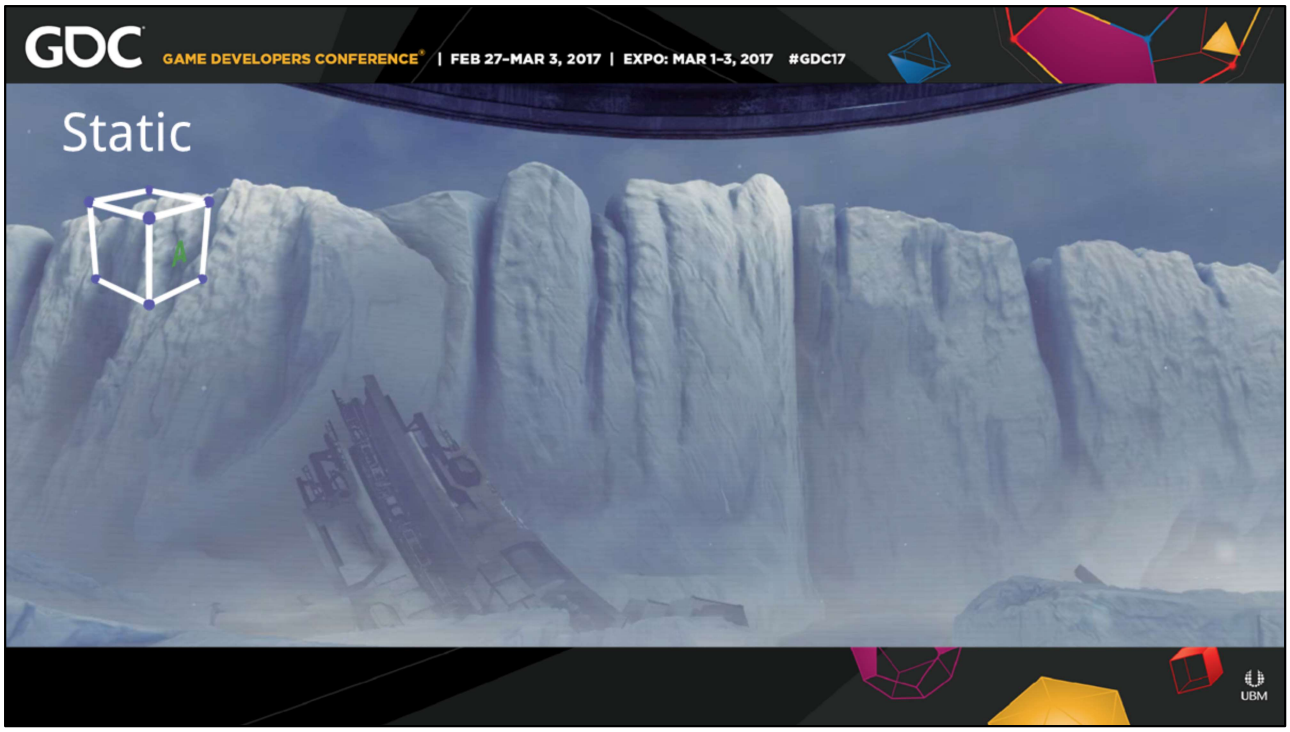
For RBDs we need to pre designate the ids on input and fracture, and make sure they didn't overlap from multiple inputs.

- We also needed to use **connectivity** based **dictionaries** to make sure each id is assigned correctly.

You can't have multiple transforms on one part. This should not be post processed either,
as calculating this on the static pre-simed frame is more efficient than doing it over a sequence.
This does require an artist to maintain a **part id** through their workflow,
However unlike a rig it is more like a **material**, but think how often art goes without materials into the game.

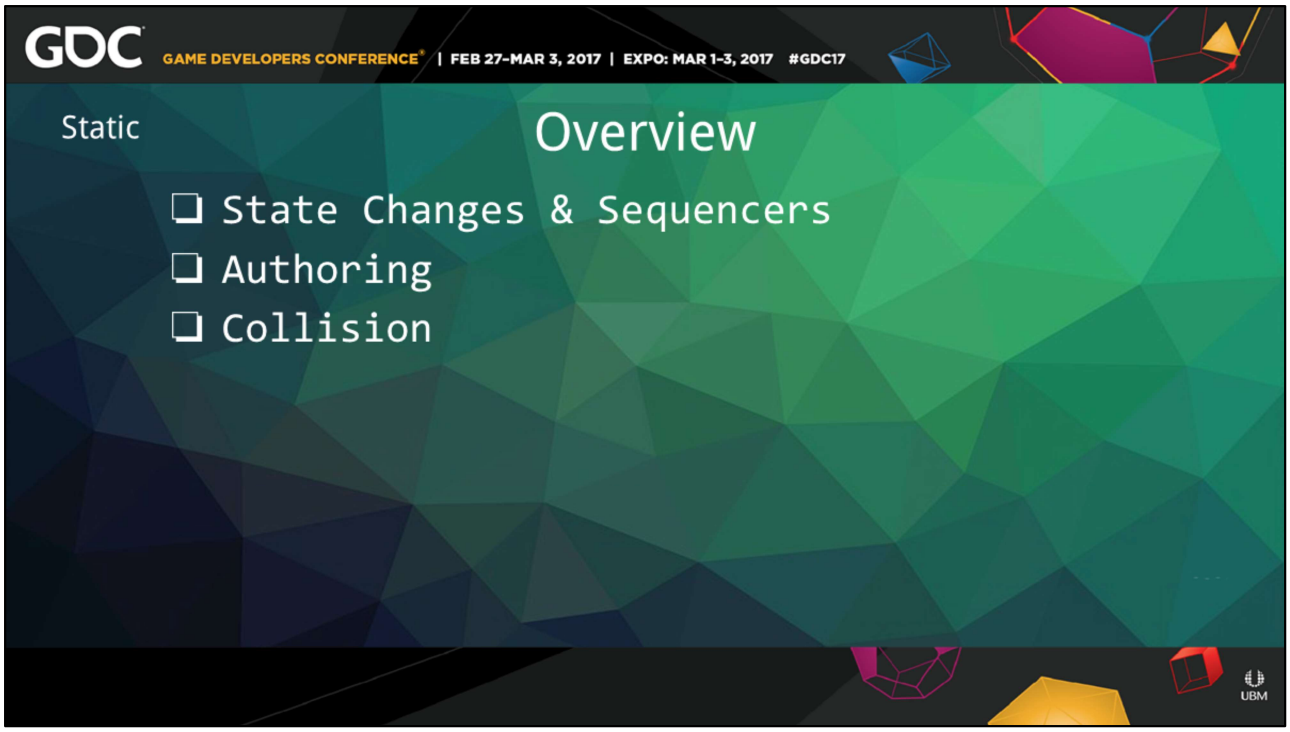
- Another item is to make sure to **combine** multiple parts traversing the same motion into a single part id.

You don’t want to pay for the same transforms over and over again.



- 46:40, 00:05, 12:15
- Moving on to static, Unfortunately **no videos**

...I leave you with a stoic start state instead.



- 46:45, 00:10, 12:05

- At the very beginning we didn't focus on the **end states** of our geometry caches
We got caught up in making it move.

- So this is a section born out of our **mistaken assumptions**

    - 46:55, 01:15, 10:50
- The majority of our **unaccounted** for **time** at the end of project fixing the static geometry

You don't want costly moving geometry pretending to be static,
Nor should you hide it out of sight, it cost additional cycles, You need to kill it.
When you use **state changes** and **sequencers**,
you can efficiently slice up any moving geometry to just when it moves
You should build this strictly GPU based, but use your traditional authoring methods.

- One of the biggest culprits in our E3 Trailer was triggering all the **towers at once**,

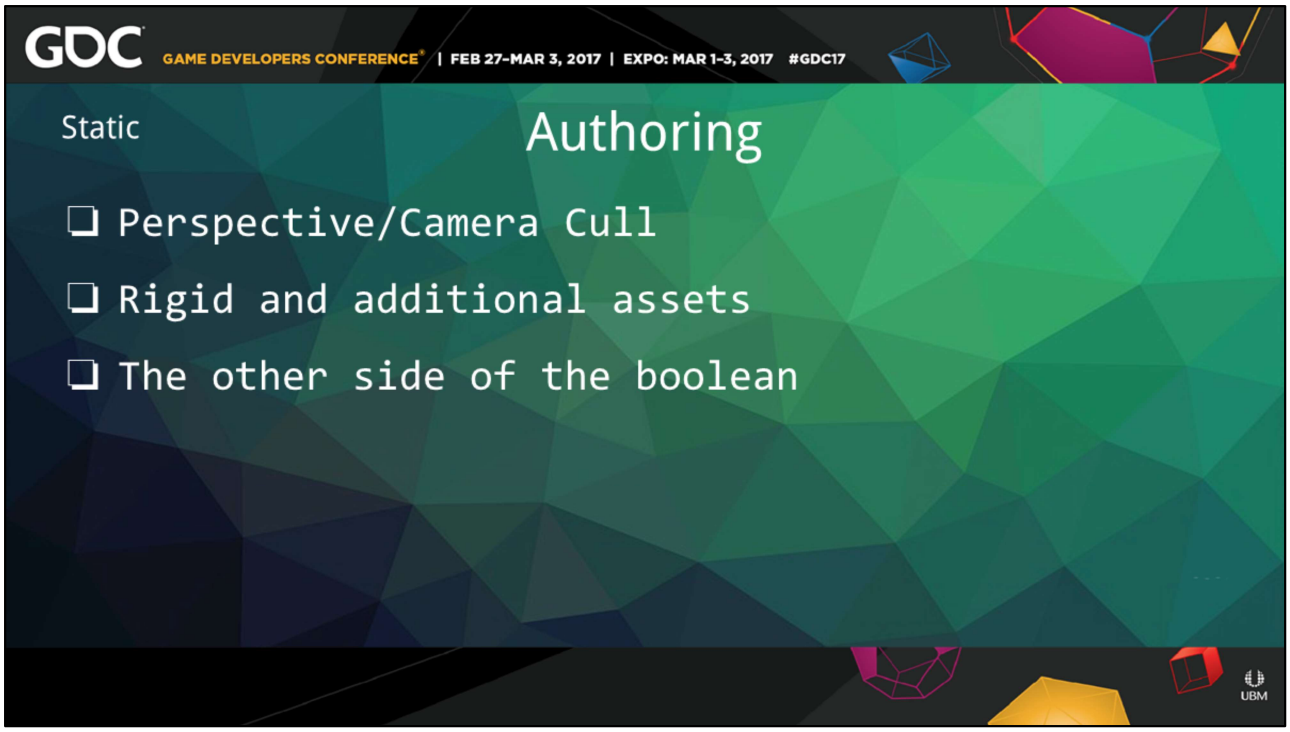
Our performance spiked for a few frames, All we needed to do was to stagger the start times, to the **start of the beats**. This removed the spike, and reduced holding frames,
saving us disk space, too

- Also we made the assumption if you destroy something, there is **nothing left** in the end.

In fact, there is often something left at the end,
You can't have everything sink through the ground, especially giant towers that collapse
So you want to clear your GPU buffer of this whole sequence in favor of a single end state

- Additionally it's important to **LOD** these end states, It seems simple, but we screwed that up.
- Another oddity we came across was how often we used **looping cycles**. Flags,

lava, and flocks.
The geocache should able to maintain a loop without the end states being called, so that
there is not a spike every cycle.



- 48:10, 00:45, 10:05

- As far as authoring, this may seem very ungame like, but **perspective cull** your end states

and the parts of geometry that move out of sight.

- Take for instance a **simple explosion**,

When it explodes, each part needs to be completely **enclosed** as it freely rotates
If these parts go **off a cliff** they need to be culled
When they **land**, the side touching the surface needs to be culled
When they **start**, you don't want to include the inside of the surface.

- However when you **model**, you model with the **intention** of those parts exploding,

so you need to model a **shell** as well as the **interior** pieces

- Additionally if a section of a building explodes you need to include

the **undamaged**, unmoving pieces in your regular **environment** geometry
This becomes a team **hand off issues** as the environment modeler,
And the fx artist need to hand geometry back and forth to each other and maintain those puzzle pieces.

- 48:55, 00:30, 09:35
- **Zabir** mentioned a solution for **runtime collision**, so this is more an alternative.

- If you need additional collision. You can include it in your **environment** geometry,
As a **state machine**, in your **sequencer**, or in a **character rig**
We used a whole range of these for the E3 sequence

- We also had this happen in the **kraken Reveal** when we were told,
nobody will ever be able to run up and touch, the **central pillar**,
but it was proven to be extremely easy to do
so we had to patch in a collision mesh in a state machine for this.

- 49:35, 00:10, 09:15

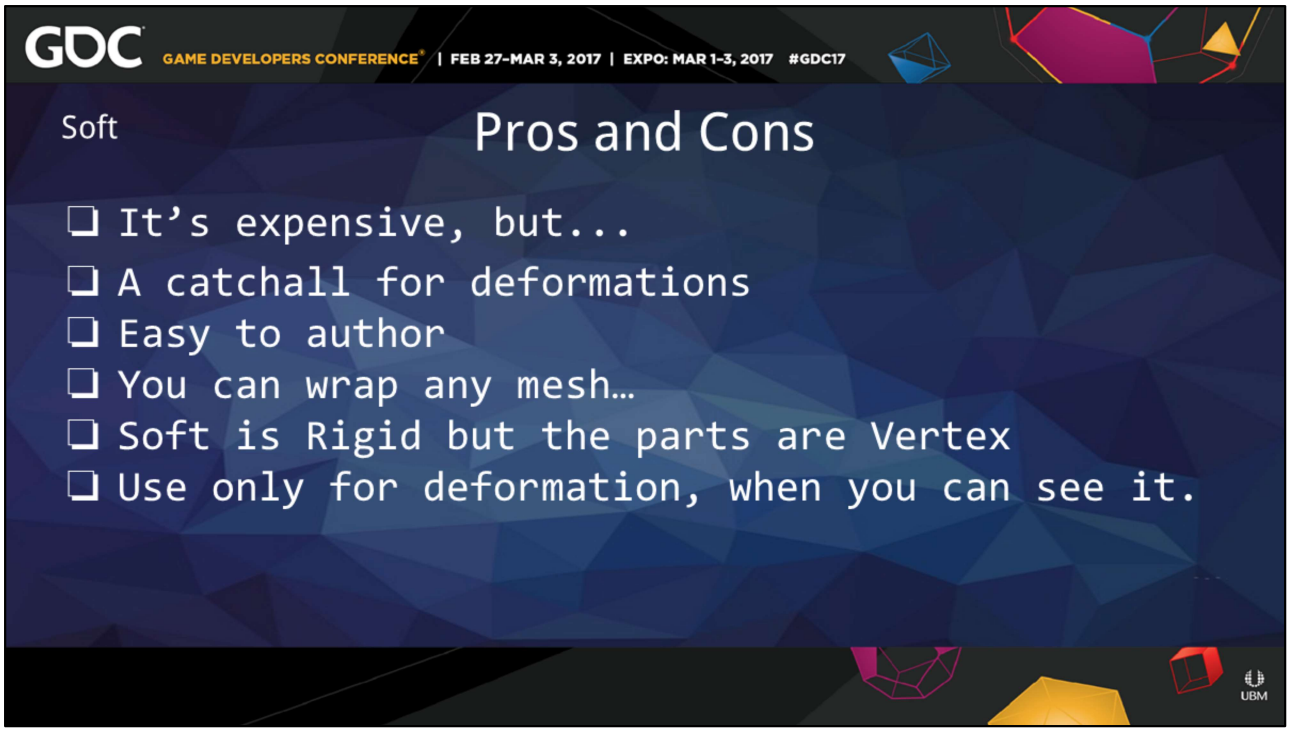
- Soft is the cool invention by Zabir to optimize deforming geometry.

- We tried cranking this up to 11 in this D L C multiplayer level, Molten. learning a few valuable lessons in doing so.

- 49:45, 00:10, 09:05

- So we are going to hit up the **pros** and **cons** for soft,

Plus a lesson learned in **extra vertex attributes**, and two **topology** examples.



- 49:55, 00:45, 08:20

• Out of the box I'm going to say Soft is relatively **expensive** in memory compared to skinning and using runtime noises.

• However Soft is a beautifully simple catch all for all deformations.
Bending Metals, Shape Morphing, Fluids, Flags, Cables, Vines, and Lava
It's extremely **easy** to author. You can use any deformer from any package.

• It's also extremely simple to add to your system after rigid,
and the surface is **implicit**, so there is no geometry to store.

• Plus we know we can push unreasonable amounts of points
The lava assets represent over **83 thousand animated** vertices, across **7 assets**
That carry color and use a custom shader,

• As we mentioned with static, only use soft when you can **see it**. It is relatively expensive.

Make sure you **culling spheres** are tight, and you **stitch** large assets like this together.

- 50:40, 00:45, 07:35

- I did want to call out one large **assumption** and **shortfall** of this methods especially in regards to the Lava,

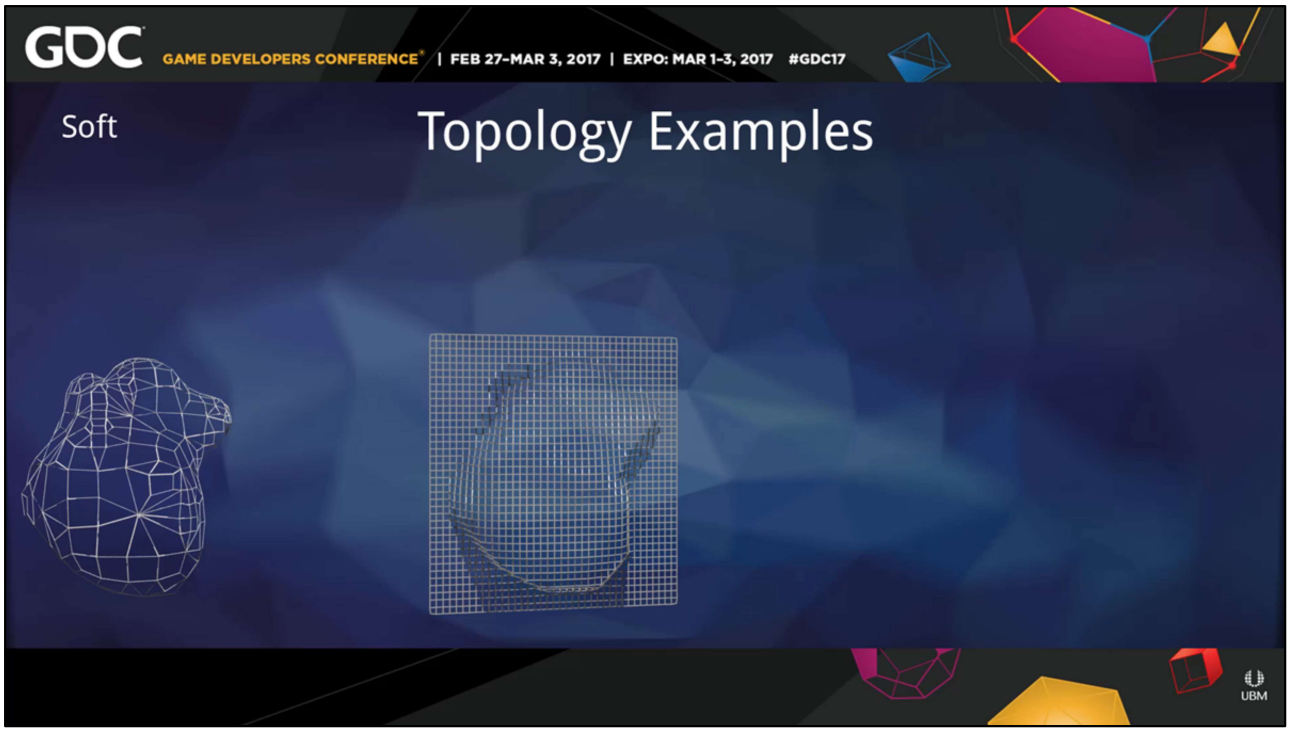
Texture density is still far above **vertex** density by a general magnitude across a surface area.
A rough equivalent is we used about **5**, **128x128** textures arrays at 180 frames
At roughly 175mb of “total” memory.

- We could have used the same sim as a nicely rendered texture **array** and the **tessellator** for higher fidelity assets

The main falls, just barely worked using a remapped 8 bit float channel
Thus a call out to the nice work of **Matt Sutherlin** to compensate for the flat areas.

- Any FX artist will tell you why you need additional attributes for FX as an option

But I would say focus on the transforms use by all these assets,
And then add **additional attributes** as second tier **options**.



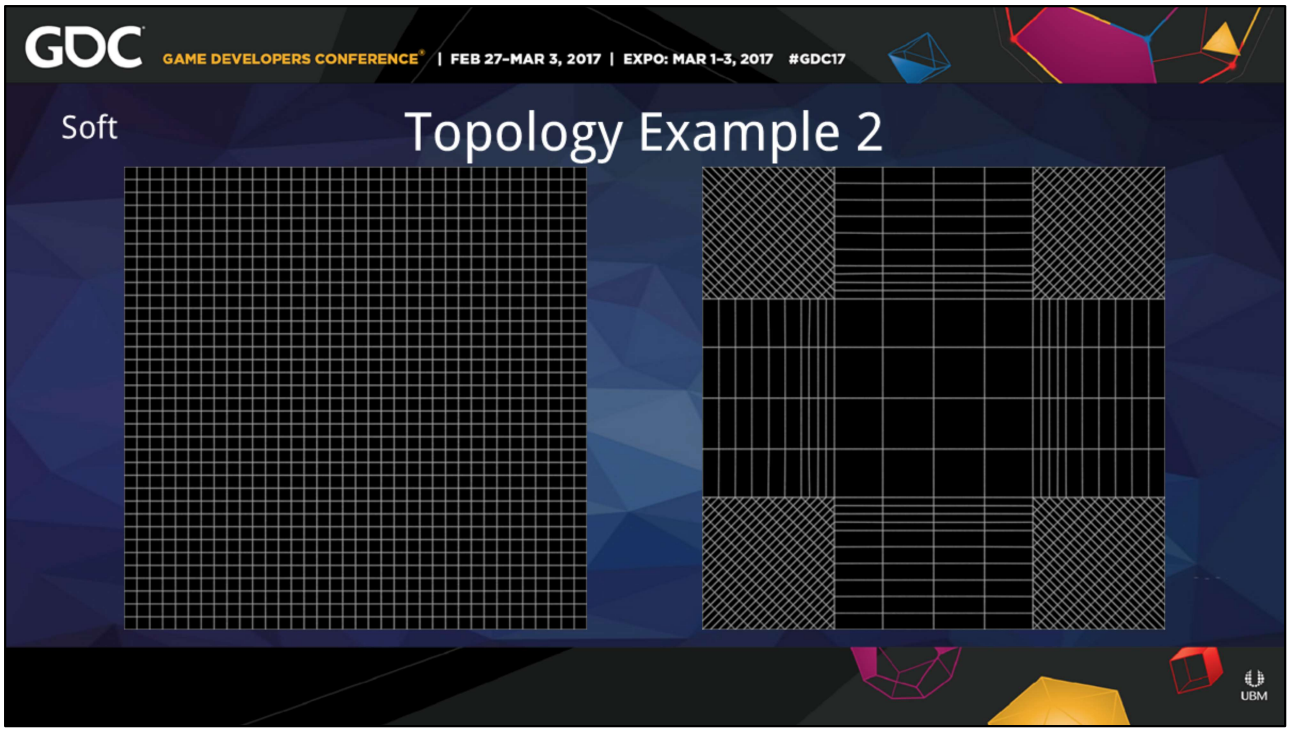
- 51:25, 00:30, 07:05

- If you can remember the mesh on the left from our fluid topology

On the right we have a **projected mesh** on that surface that is game usable.
By doing this we can convert a majority of all of our changing topology meshes,
Like the lava fluid simulation into a cheaper soft method.

- If you are just using **deformers** than straight away you can use this method.

The only key is to make a convincing looping geometry,
Which if you are used to tileable textures, you can use the same method
but with time to blend the shape with it's beginning loop.



- 51:55, 00:30, 07:35

- In the previous method I clearly did not optimize the mesh to that surface,

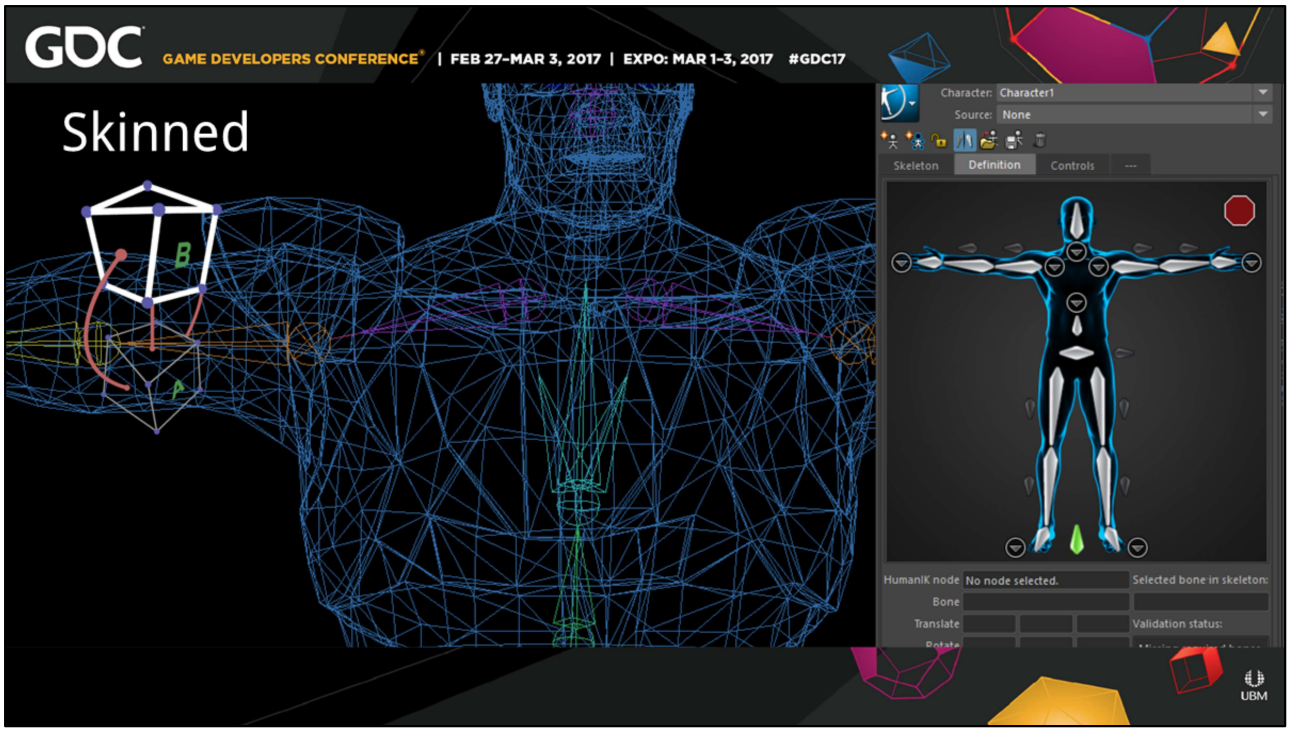
but just like creature **topology** where you can add edge loops to where it matters,
You can do the same for soft.

- Here is an example for a cloth on a **square** table,

If you drop the cloth on the right you'll get weird fold patterns at the edge, and waste vertices in the middle and the sides.
The one on the left has **diagonal** edges so that the cloth can accordian nicely at the corner of the table.

- You can pretty much contort to any shape, or rag with a **hole** in them like some of our ripped tarps,

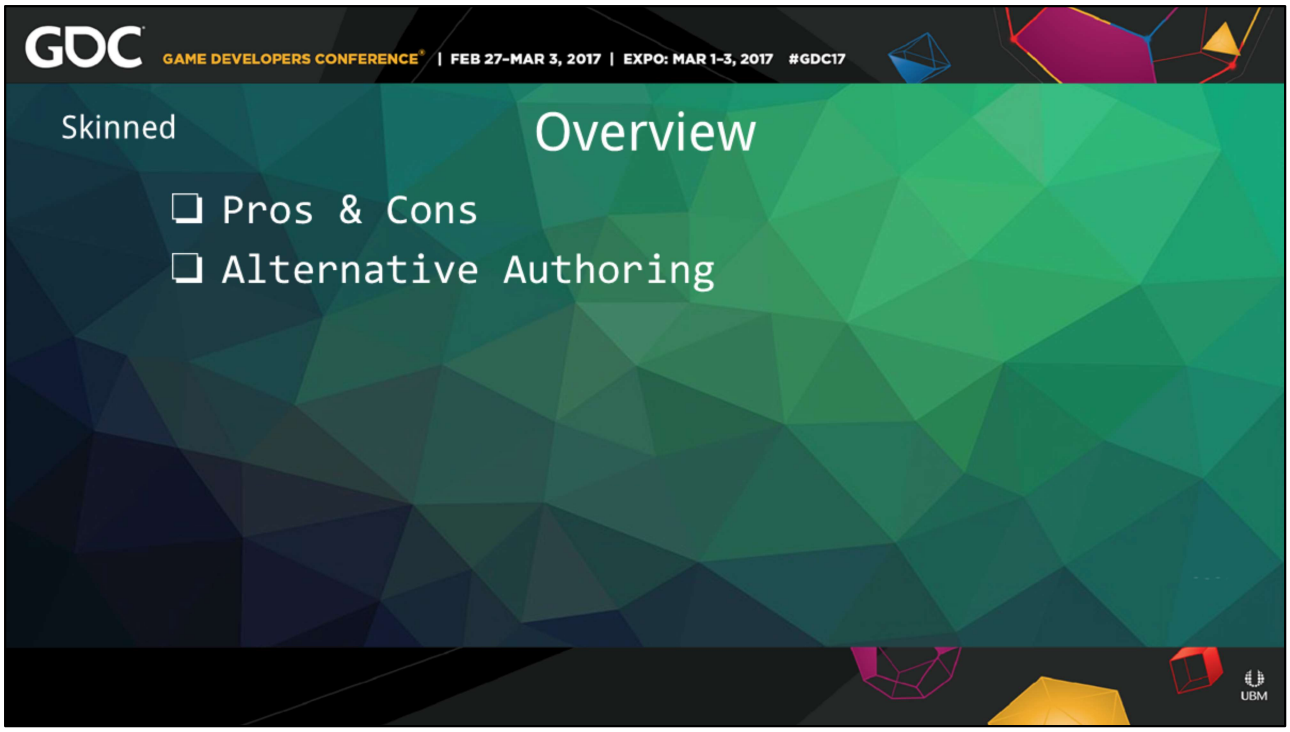
The only rule of thumb for us, was one connected shape per an asset



- 52:25, 00:10, 07:25

- The last component on our agenda is **skinned**.

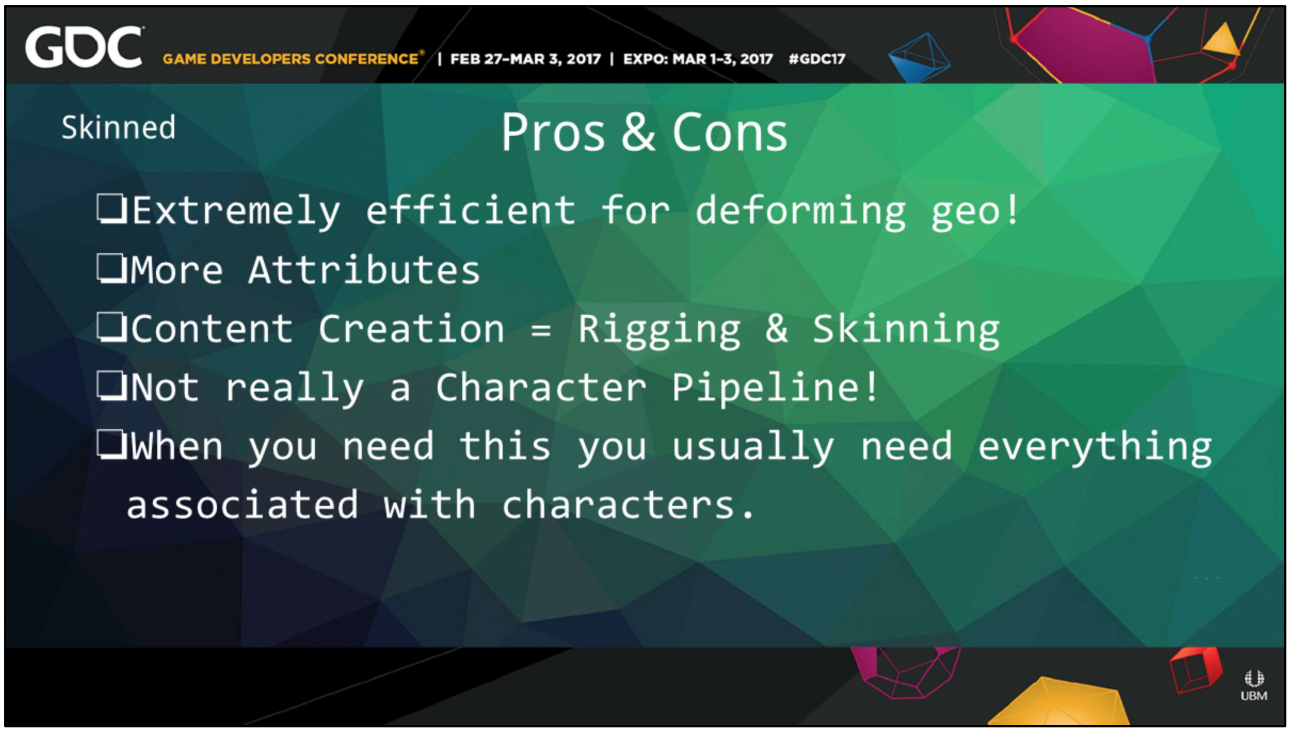
I imagine everyone is familiar with **HIK** systems and motion builder
these represent a vast tool set in animation that speaks to the complexity of this pipeline.



- 52:35, 00:10, 07:15

- First we are going to hit up the **pros** and **cons**, of using this within a geometry caching system

and an **alternative authoring** story.



- 52:45, 00:50, 06:25

- This is hands down the most **efficient** with best **compression** method of storing deforming geometry.

It does require an additional attributes such as weighting for skinning that does adds more byte depth to the GPU struct,
and it requires you to have the most complex runtime shader.

- This also requires you to do **rigging** and **skinning** in your authoring package

which we have successfully avoided to this point.
Overall the cost of implementing this type of saving for us is not worth the investment and authoring hassle for the type of content our team was creating.
We can just as easily use our very robust **character pipeline**, instead of re-inventing it.
As was the case if we needed this types of savings we needed everything else associated with it.

- However you can use this technology and componentize the system to a very effective means.

I'm going to defer to another GDC talk from this year by Mario and Norman called the "**Illusion of Motion**"
Where at their studio in complete unknown parallel development has done this work and it's an amazing savings
Where you can get a character's worth of motion of 166 minutes, in two 4k textures.

- 53:35, 00:45, 05:40

- So I'm going to leave **skinned** out on his note,

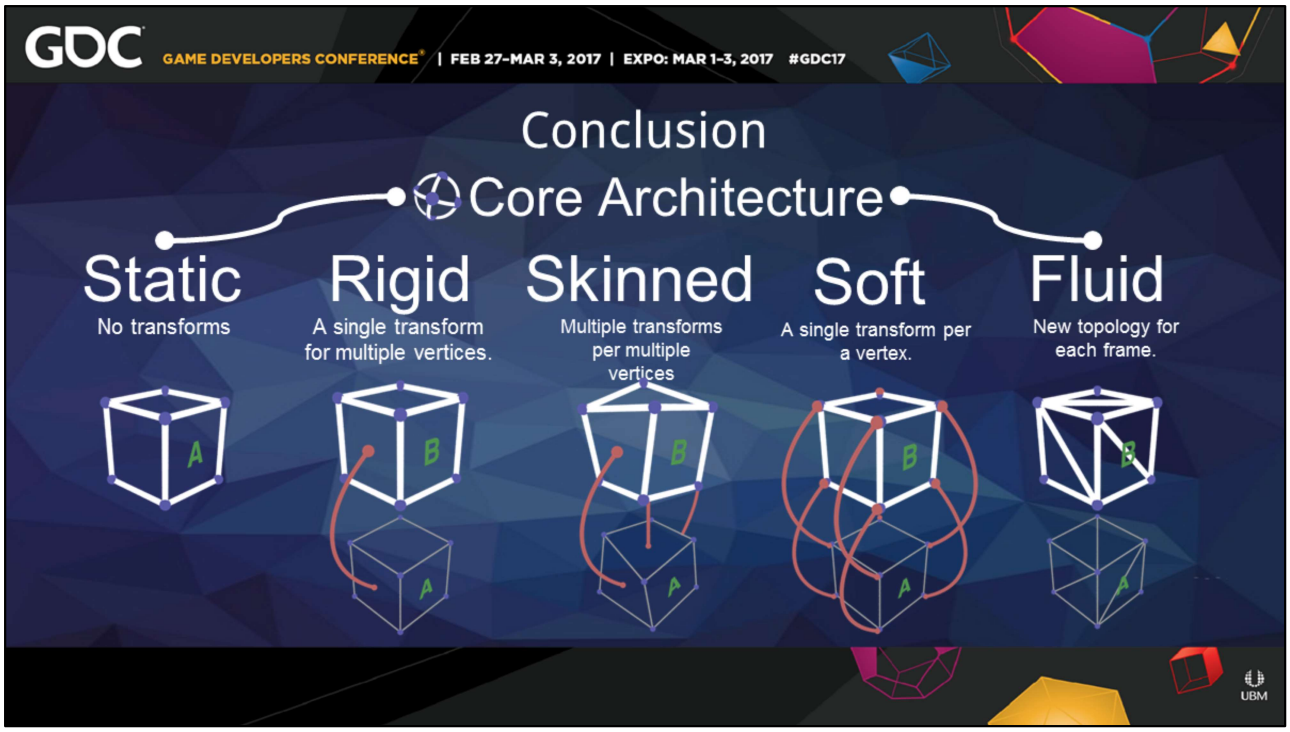
For deforming and complex objects as buildings and machinery
there is a wide array of **procedural rigging** and **animation** techniques that could be leverage,
that I have not had the reason to explore.

- Think of a **snake rigs**, something relatively simple, but this could be automated to have say hundreds of worms

and creepy crawlies wriggle over something, in a flocking pattern that would be amazing.

- At the **beginning** of this talk I defined Geometry Caching as the storage of **Vertex Motion**

However we primarily focused on **destruction** and designed our system based on the components required to build a **crowded stadium**
So there are probably dozen of authoring stories that simplify the traditional Skinning and Rigging pipeline
Such as Mario and Norman did in parallel.



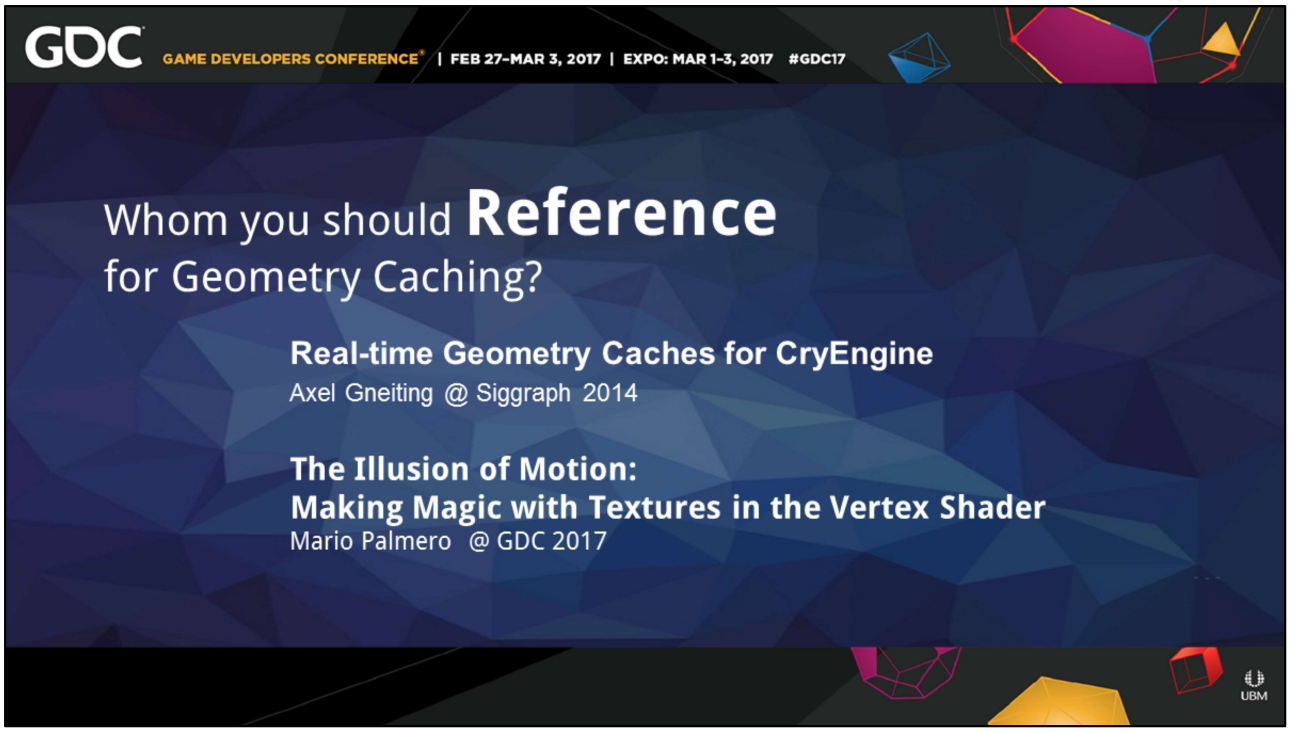
- 54:20, 00:30, 04:00

- So in conclusion we have **five different components** for our geometry caching system,

Static, Rigid, Skinned, Soft, and Fluid plus the core Architecture.

- For the **far future**, You could have an export process that **mitigates** the **switching** between these different methods.

For each individual method you can find further heuristic to process these faster on the GPU.
DX12 does allow for us to balance further than we could do before.
Our **State machine** and sequencer authoring for this method can be further improved.



    - 54:50, 00:50, 55:45
- We think our system was pretty good, we shared our implementation with Coalition team

and they leveraged it in **Gears of War 4**, and now aspects of this tech are in **Unreal**.

- For those that have yet to implement a Geometry Caching pipeline in your engine

I **strongly urge** you to look at these talks and papers.

- The first talk we consider our granddaddy, that started it all for us.

Real-time Geometry caching for **CryEngine** by **Axel Gneiting** at Sigraph in 2014.

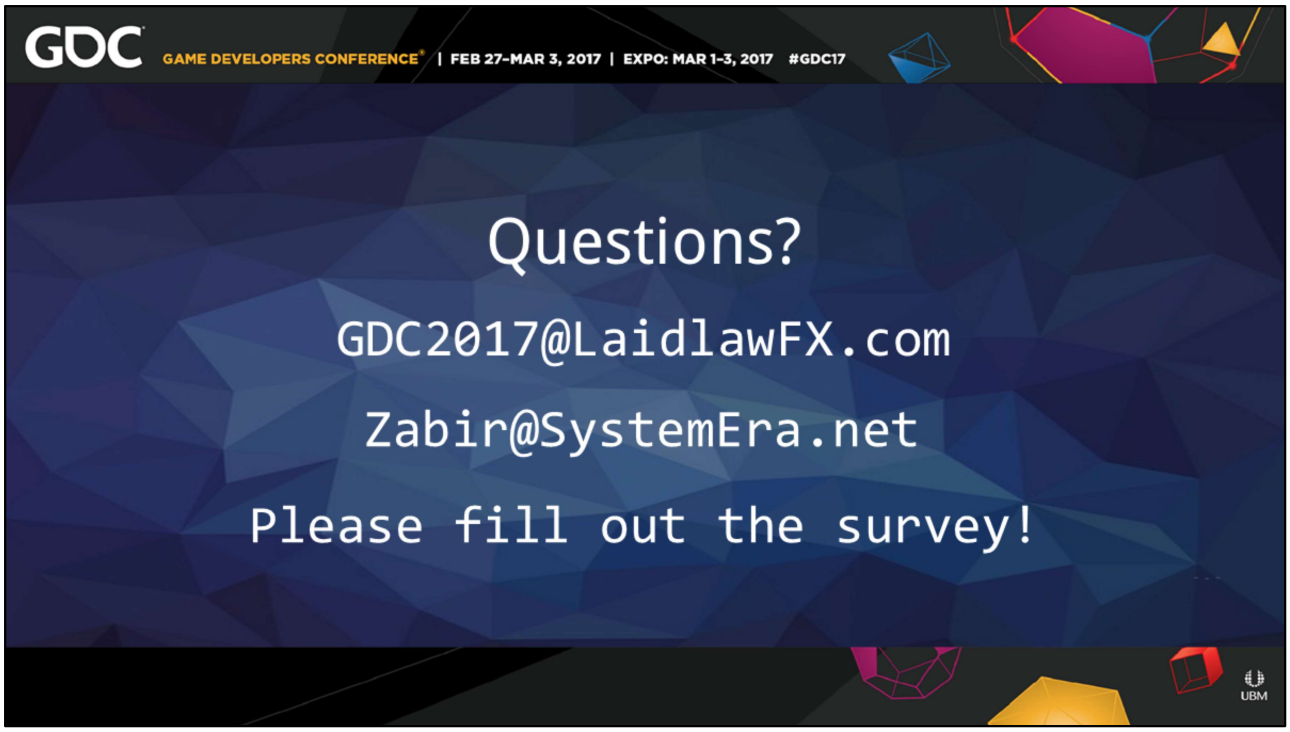
- In essence we are using the lessons learned in that Ryse Paper

and with our lessons from our E3 Launch Trailer from 2015 to develop these optimizations.

- I also want to mention the work of **Mario Palmero** & **Norman Shaar** from their work at Tequila Works.

with How to Take Advantage of Textures in Vertex Shaders
They did work in completely unknown parallel that complements our work,

proving it’s usefulness.



    - 55:30, 03:30, 00:00
- That was a whirlwind for us, and we both **hope you** can leverage aspects of this talk in the future.

- Feel free to ask **questions**, now.

Or you can reach out Zabir and I via **e-mail**.

- And please we would greatly appreciate if you enjoyed this talk,

That you fill out the **survey** so we can come back again.
Thank you