Bridging Pixels & Code

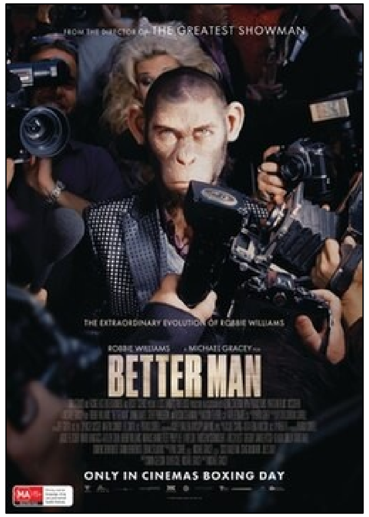
-- Teaching Computer Graphics to Technical Artists --

MATTHIEU DELAERE

Who am I?

Matthieu Delaere

- DAE Howest and BUAS MGT alumnus
- Senior Lecturer (C++ Graphics Programming) @ DAE Howest
- Senior Research Engineer @ Wētā FX
- Previously Rendering Researcher @ Unity


FROM THE DIRECTOR OF THE GREATEST SHOWMAN
THE EXTRAORDINARY EVOLUTION OF ROBBIE WILLIAMS
BETTER MAN
ONLY IN CINEMAS BOXING DAY

ACT I

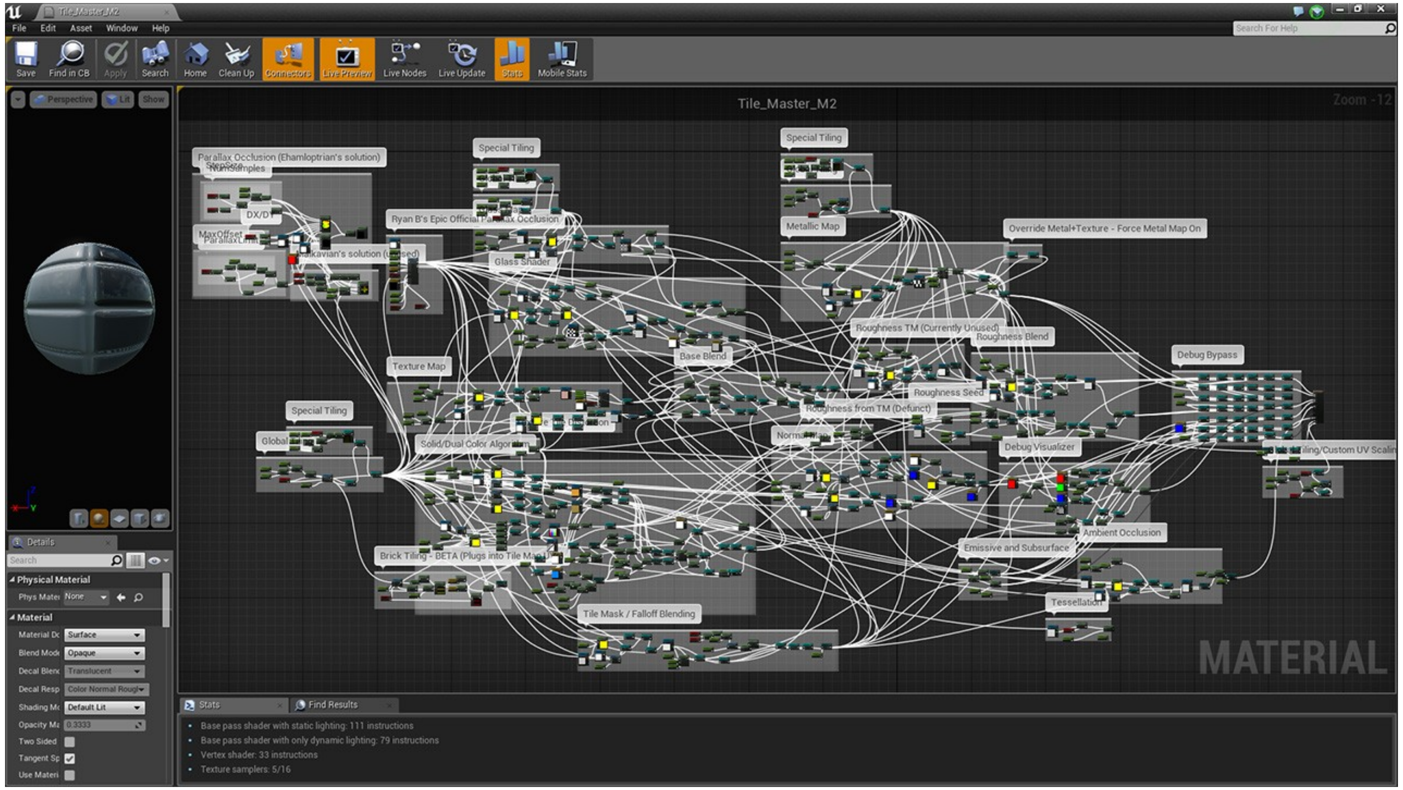
Seeing Beyond the Pixels

Boundless Creation

- When we look at this, we see **art**. But behind every pixel, there is a sequence of instructions, data transformations, performance considerations, and much more.
- For most students early in their (educational) career, these are often **invisible** and considered **not high priority**, even though they are crucial for game development!
- Most students concentrate on **perfecting their craft** rather than focusing on the product and overarching structure.
- This often resulted in:
    - artists not being able to spot bottlenecks, optimize their scenes and explaining why art was made in the was it was.
    - programmers not being able to discuss more advanced concepts or produce proper alternatives based on constraints.

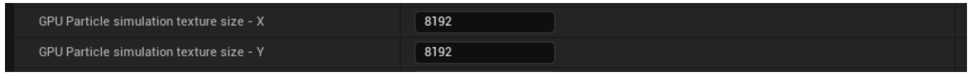
https://vimeo.com/1065465468?fl=pl&fe=vl

Boundless Creation

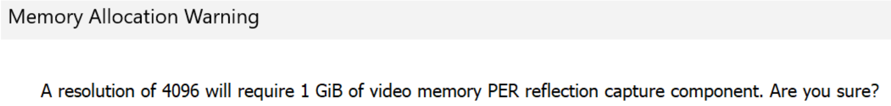


- Base pass shader with static lighting: 111 instructions
- Base pass shader with only dynamic lighting: 79 instructions
- Vertex shader: 33 instructions
- Texture samplers: 5/16

https://forums.unrealengine.com/t/material-optimization/59111/6

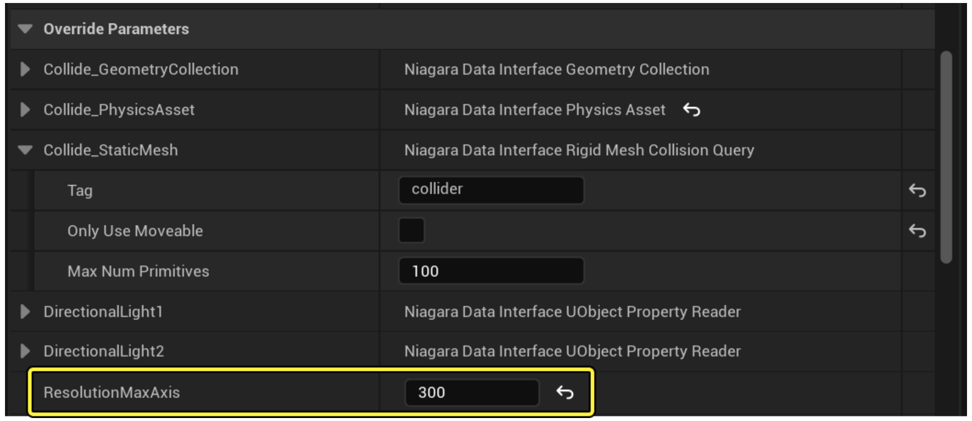


GPU Particle simulation texture size - X	8192
GPU Particle simulation texture size - Y	8192



Memory Allocation Warning

A resolution of 4096 will require 1 GiB of video memory PER reflection capture component. Are you sure?



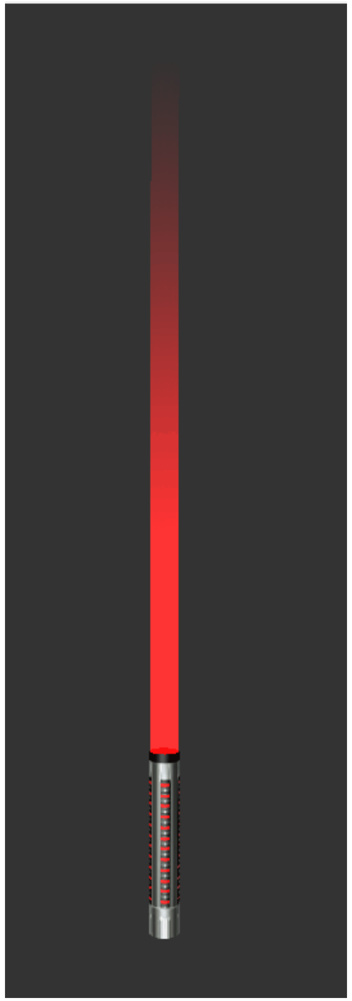
Override Parameters

Collide_GeometryCollection	Niagara Data Interface Geometry Collection
Collide_PhysicsAsset	Niagara Data Interface Physics Asset
Collide_StaticMesh	Niagara Data Interface Rigid Mesh Collision Query
Tag	collider
Only Use Moveable	□
Max Num Primitives	100
DirectionalLight1	Niagara Data Interface UObject Property Reader
DirectionalLight2	Niagara Data Interface UObject Property Reader
ResolutionMaxAxis	300

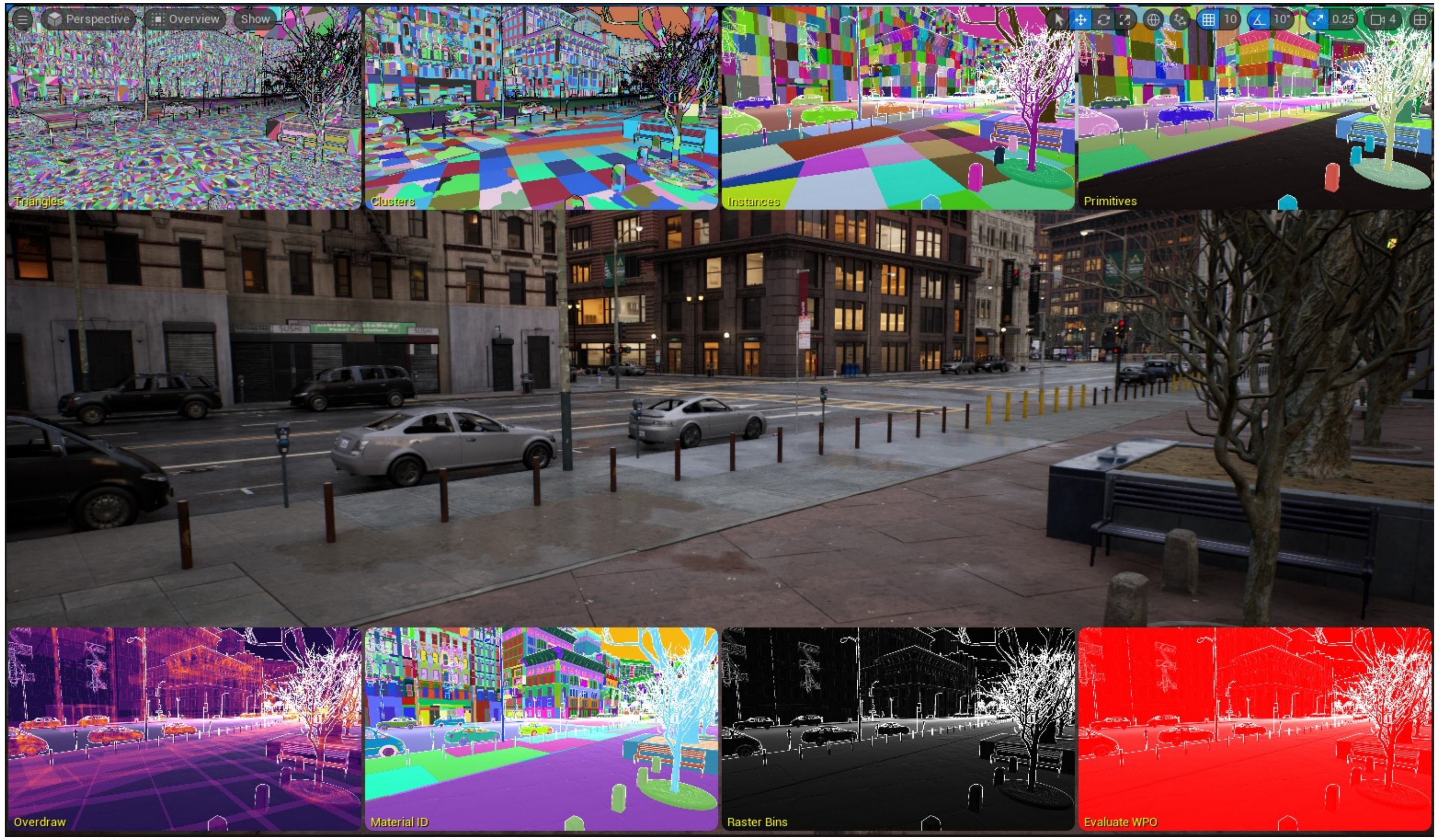
https://dev.epicgames.com/documentation/en-us/unreal-engine/niagara-fluids-quick-start-guide-for-unreal-engine

Boundless Creation

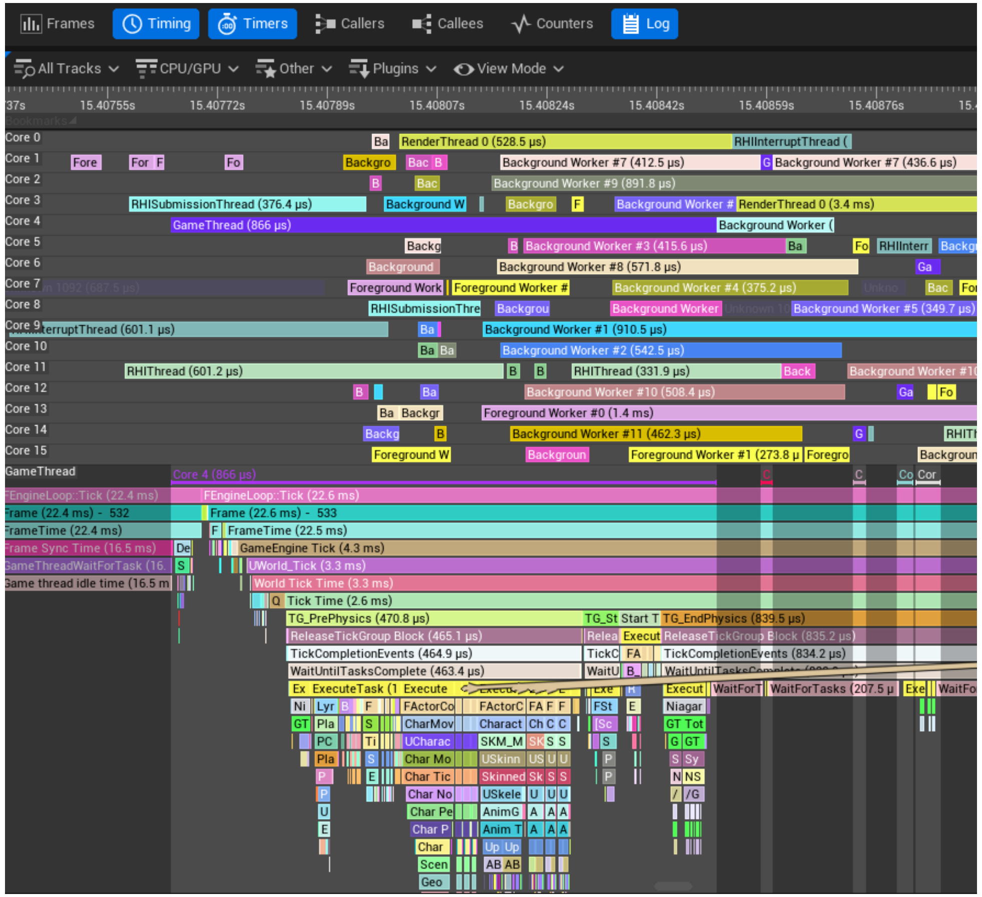
- The prominent effects are **freezing** when encountering "real" problems, hitting a **skill ceiling**, not speaking the **technical language**, lots of **retraining** needed, etc.
- How did we get there, so what does **not** work:
    - **passive tutorial consumption** → memorizing or copying is not understanding!
    - **theory without implementation** → reading about swimming does not teach you swimming!
    - **tool-dependent knowledge** → you learned where buttons are, not what buttons do!
    - **API learning without foundation** → APIs abstract away important mechanisms!
    - **isolated skill development** → games depend on bridging art and technology. Learning each side separately does not build a bridge!
    - **learning in siloed comfort zones** → growth happens at the edge of capability and not in the comfort zone!
- Many traditional approaches focus on **results**... We have "optimized" for **completion over understanding**!
- Finishing a tutorial **feels like progress**, while often you are just following steps **without deep understanding**.
- **Real learning is messy, frustrating and slow, but it works!**



Boundless Creation



https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine


Frames
Timing
Timers
Callers
Callees
Counters
Log
All Tracks
CPU/GPU
Other
Plugins
View Mode
15.40755s
15.40772s
15.40789s
15.40807s
15.40824s
15.40842s
15.40859s
15.40876s
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10
Core 11
Core 12
Core 13
Core 14
Core 15
RenderThread 0 (528.5 µs)
Background Worker #7 (412.5 µs)
Background Worker #7 (436.6 µs)
Background Worker #9 (891.8 µs)
RHISubmissionThread (376.4 µs)
RenderThread 0 (3.4 ms)
GameThread (866 µs)
Background Worker #3 (415.6 µs)
Background Worker #8 (571.8 µs)
Background Worker #4 (375.2 µs)
Background Worker #5 (349.7 µs)
Background Worker #1 (910.5 µs)
Background Worker #2 (542.5 µs)
RHIThread (601.2 µs)
RHIThread (331.9 µs)
Background Worker #10 (508.4 µs)
Foreground Worker #0 (1.4 ms)
Background Worker #11 (462.3 µs)
Foreground Worker #1 (273.8 µs)
GameThread
FEngineLoop::Tick (22.4 ms)
FEngineLoop::Tick (22.6 ms)
Frame (22.4 ms) - 532
Frame (22.6 ms) - 533
FrameTime (22.4 ms)
FrameTime (22.5 ms)
Frame Sync Time (16.5 ms)
GameEngine Tick (4.3 ms)
UWorld_Tick (3.3 ms)
World Tick Time (3.3 ms)
Tick Time (2.6 ms)
TG_PrePhysics (470.8 µs)
TG_EndPhysics (839.5 µs)
ReleaseTickGroup Block (465.1 µs)
ReleaseTickGroup Block (835.2 µs)
TickCompletionEvents (464.9 µs)
TickCompletionEvents (834.2 µs)
WaitUntilTasksComplete (463.4 µs)
ExecuteTask
WaitForTasks (207.5 µs)

https://www.intel.com/content/www/us/en/developer/articles/technical/unreal-engine-optimization-profiling-fundamentals.html

Boundless Creation

- The obvious solution is to subject yourself to **productive struggle**!
  - Attempt something slight beyond your current capabilities.
  - Fail at it. This is **required** and not optional!
  - Debug, investigate and understand **why** you failed.
  - **Succeed through understanding**!

- The “framework” is simple: **Build → Analyze → Explain**
  - **Build** – implement something from scratch. It forces to fill knowledge gaps immediately.
  - **Analyze** – break it deliberately, debug it and understand why it works or does not work.
  - **Explain** – teach it to someone else (the community is your friend) via blogpost, video, presentation, etc. You cannot bullsh*t when teaching.

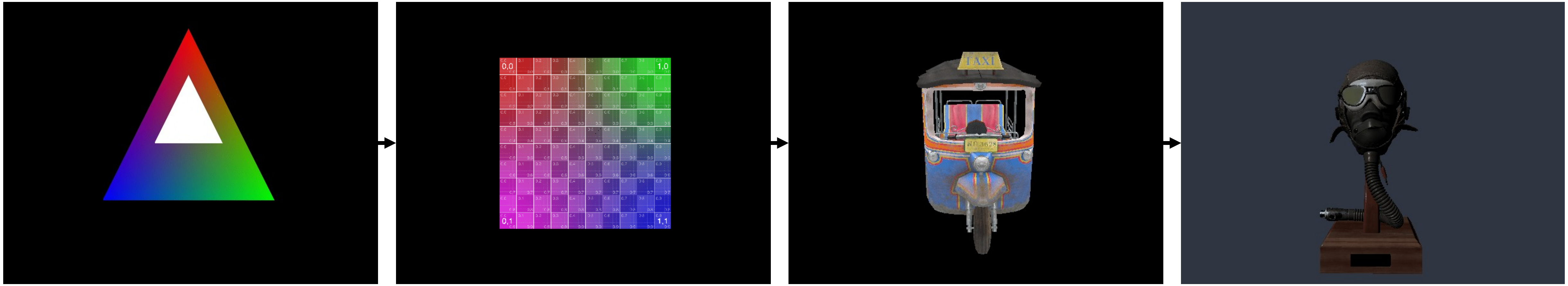
https://medium.com/skillenza/what-does-it-take-to-become-a-great-software-engineer-22be56649c6d

ACT II

Making the Invisible Visible

What’s in the Box?!

- For **both** programmers and artists, build a **software rasterizer** using your preferred language (C++, Python, etc.).
- Remove all APIs and boilerplate. Focus on “**raw pixels**”, **memory** and **mathematics**! In other words, **remove the black box** and **gradually learn core concepts**. You cannot fully understand what you have not built.
- The key concept is that you **reinvent the pipeline from scratch**, not because we need a new renderer but because **deep understanding requires reconstruction**!

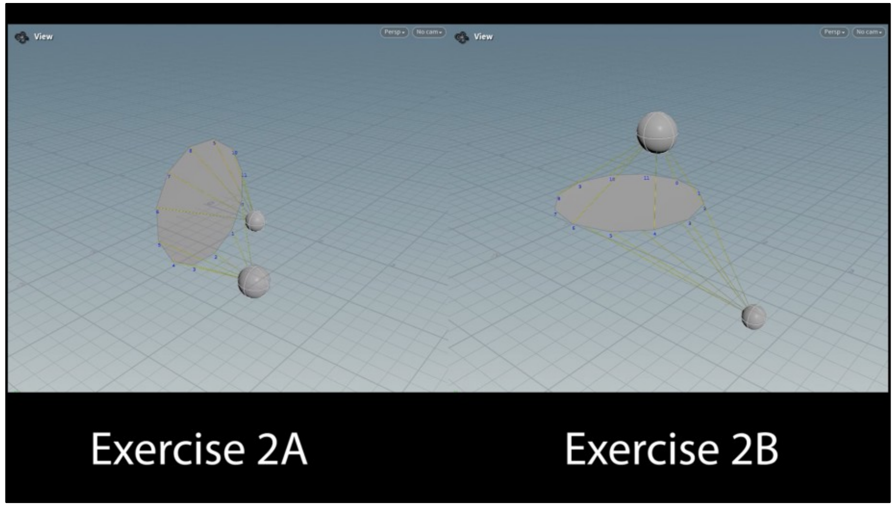


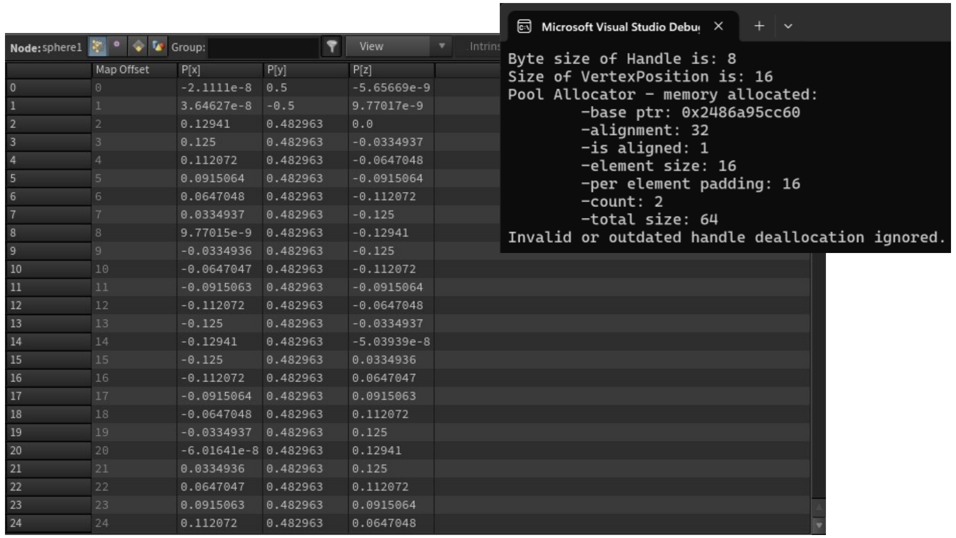
What’s in the Box?!

- **Removing all unnecessary overhead** (GPU and engine architecture, synchronization, etc.) allows for **very focused examples** and **easy-to-showcase-and-test concepts** or problems.
- What this **achieves** is:
  - **Data Literacy** → you learn how vertex data gets transformed throughout the pipeline and how it affects pixels.
  - **Conceptual Insight** → you learn why GPU stages exist without explicitly referencing the GPU just yet.
  - **Performance Intuition** → you learn the impact of decisions (small triangles, data and lighting complexity, etc.).
- This approach does have **challenges**:
  - **Accessibility Barrier** → a full coding assignment might still be too intimidating.
  - **Visualization Difficulty** → some concepts are not always easy to visualize out-of-the-box.
  - **Assumed Understanding Risk** → small conceptual gaps cannot be overlooked. Assuming you will get it later is dangerous.
- How to potentially deal with these challenges?

What’s in the Box?!

- While it might be **counter intuitive** at times, mix the visual coding part (writing a rasterizer) with **standalone console applications** or **standalone demos using external tools**, either static or with **animated examples**. **The journey, not the result, matters!**
- This lowers the **accessibility barrier** (contained examples), allows for **tailored visual learning** (e.g. using external tools the artist knows) and makes sure there are **clear milestones** for testing assumptions and understanding.
- Continuously use the **lens of being a teacher**. **You deepen your understanding by teaching what you learn to others**.
- **BUILD → ANALYZE → TEACH**





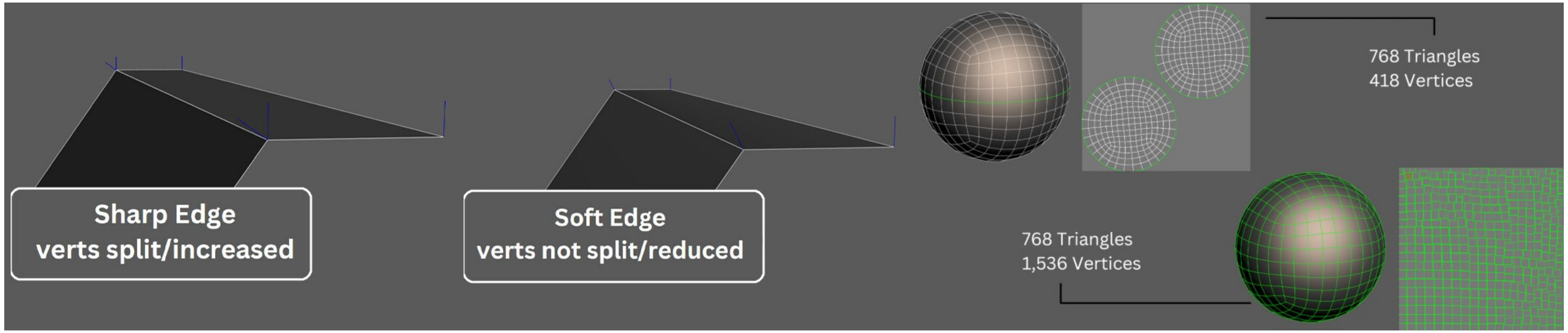
	Map Offset	P[x]	P[y]	P[z]
0	0	-2.1111e-8	0.5	-5.65669e-9
1	1	3.64627e-8	-0.5	9.77017e-9
2	2	0.12941	0.482963	0.0
3	3	0.125	0.482963	-0.0334937
4	4	0.112072	0.482963	-0.0647048
5	5	0.0915064	0.482963	-0.0915064
6	6	0.0647048	0.482963	-0.112072
7	7	0.0334937	0.482963	-0.125
8	8	9.77015e-9	0.482963	-0.12941
9	9	-0.0334936	0.482963	-0.125
10	10	-0.0647047	0.482963	-0.112072
11	11	-0.0915063	0.482963	-0.0915064
12	12	-0.112072	0.482963	-0.0647048
13	13	-0.125	0.482963	-0.0334937
14	14	-0.12941	0.482963	-5.03939e-8
15	15	-0.125	0.482963	0.0334936
16	16	-0.112072	0.482963	0.0647047
17	17	-0.0915064	0.482963	0.0915063
18	18	-0.0647048	0.482963	0.112072
19	19	-0.0334937	0.482963	0.125
20	20	-6.01641e-8	0.482963	0.12941
21	21	0.0334936	0.482963	0.125
22	22	0.0647047	0.482963	0.112072
23	23	0.0915063	0.482963	0.0915064
24	24	0.112072	0.482963	0.0647048

```
Byte size of Handle is: 8
Size of VertexPosition is: 16
Pool Allocator - memory allocated:
        -base ptr: 0x2486a95cc60
        -alignment: 32
        -is aligned: 1
        -element size: 16
        -per element padding: 16
        -count: 2
        -total size: 64
Invalid or outdated handle deallocation ignored.
```

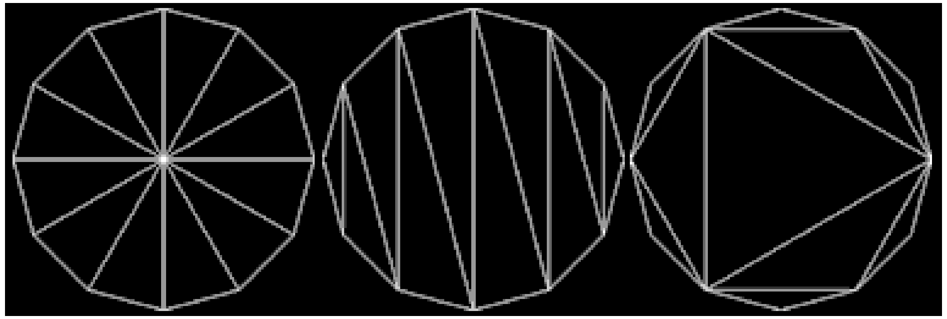
https://www.sidefx.com/docs/houdini/ref/panes/geosheet.html

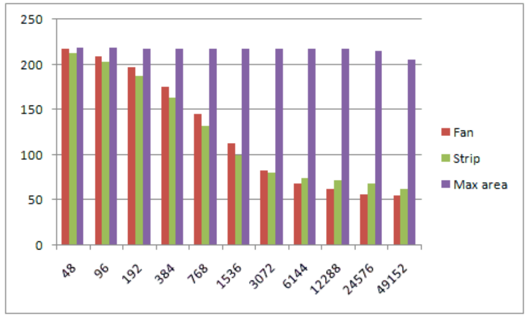
What’s in the Box?!

- When implementing and testing features, **follow the data** and **profile early** on to understand the impact of decisions both in code and data. Playing around with different setups will help with your comprehension.



https://www.artstation.com/blogs/ericcorreia/2AM0l

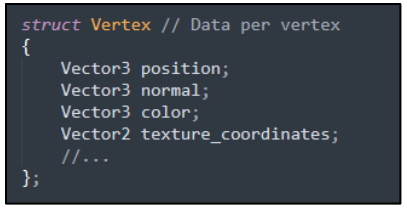




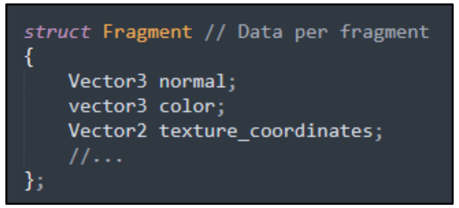
https://www.humus.name/index.php?page=News&ID=228

What's in the Box?!

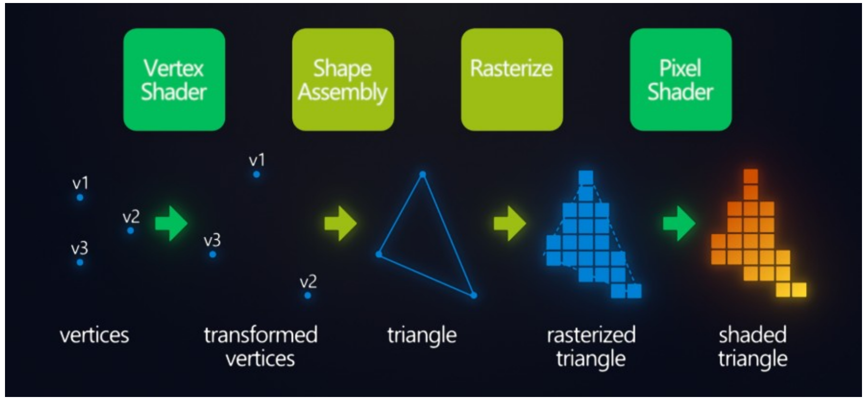
- Focus on the following concepts:
    - **Data flow** and **data transformation**.
        - Why is the **position of a fragment a vec4**, and in which space is each component?
        - How to perform **correct depth interpolation** of attributes?
    - Why the **depth buffer** is not linear?
    - What are the different ways to **sample a texture** (related to filtering)?
    - How can your **data influence the renderer** (vertex attributes and count, texture resolutions and compression, etc.)?
        - Even as an artist, learn the different **types of memory** and their properties (cache memory, bandwidth, etc.).
    - What data is needed for **per-pixel lighting**, and which data can be shared among pixels?
        - Focus on "input->function->output" as this will help you later!



```
struct Vertex // Data per vertex
{
    Vector3 position;
    Vector3 normal;
    Vector3 color;
    Vector2 texture_coordinates;
    //...
};
```



```
struct Fragment // Data per fragment
{
    Vector3 normal;
    vector3 color;
    Vector2 texture_coordinates;
    //...
};
```



https://mini.gmshaders.com/p/vertex

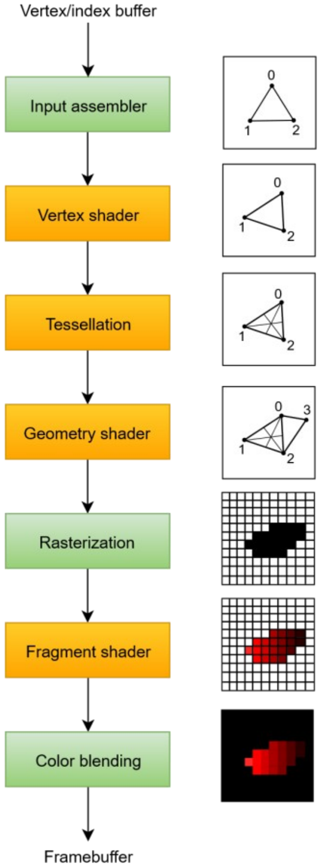
- Before continuing, make sure your thoroughly understand **why things work** and how **you have influence** over it! You do not need the best implementation, but you need to be able to "**connect the dots**".
- For **mathematics** or **CS topics**, **learn Just-In-Time and not Just-In-Case**!

ACT III

The Machine Scales Up

The same, but faster!

- Once you understand how pixels are "born", you can scale up and see how GPUs actually make things faster through parallelization.
- Start with a simple API such as OpenGL or DirectX11. Key here is to **map** your learnings to this predefined **graphics pipeline**.
- **Revisit your CPU-based rasterizer while doing this**. When learning a simple API, **the focus should be to learn the API**, **not core concepts**. If you feel something is missing, go back to your rasterizer as it should be the easier environment to test things. Make sure there is a **clear distinction between their purposes**.
- At this point, while learning the API, there are two new things you should focus on as well:
  - Learn **shader coding** (link this to the "input->function->output" of per-pixel lighting). It is a new language but make sure to understand the architecture and do not loose yourself in making fancy shaders from the get-go.
  - Learn about the **GPU architecture**!



https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

The same, but faster!

- Do not be intimidated by the GPU. There are great **visual resources** out there such as **Render Hell** by **Simon Schreibt**. These are equally valuable for artists and programmers!
- Once you understand what the GPU is doing on a high level, you realize that **every artistic choice affects performance** but also **image quality**.
- For **shader programming**, use existing frameworks or engines as the sandbox, though be careful and initially **avoid** purely node-based systems.
- During this stage of learning **avoid**:
  - wrapping or **hiding all API specific code**. The purpose is **not** to write an agnostic renderer!
  - writing a full-fledged (production) **game engine**.
- Instead, realize the focus is on understanding:
  - a graphics API is just an extra **indirection** to communicate with the GPU.
  - that using a graphics API is all about **data and state management**.
  - that this management requires **data transfer**.

https://simonschreibt.de/gat/renderhell/

The same, but faster!

- Once the core concepts of OpenGL or DirectX 11 are well understood, one can switch to **modern APIs** such as Vulkan and DirectX 12. [**optional** for art-focused students]
- While these can be intimidating at first (again), there are only a "few" **key differences**, which if you understand these, you will understand it is not that hard:
  - **synchronization** is explicit and must be handled with care.
  - **resource management** is more **verbose** which allows for more control.
  - **state tracking and setup** is more **explicit** and sometimes cumbersome.
- When learning another new API, **one must separate the new concepts from the API again**!
- Some insights for learning Vulkan:
  - **Avoid the graphics pipeline** when learning about **synchronization** and **resources**. Instead use the compute pipeline (another new topic you can explore with OpenGL first).
  - **Do not skip on synchronization details**! Take your time to understand **execution** and **memory barriers**, and the related **stage** and **access masks**, thoroughly.
  - Avoid render passes and use **dynamic rendering** first!
  - Always **question why** you do something when using tutorials! Some things are wrong or not best practices (best practices lists do not help you in the beginning).

ACT IV

Bridging The Gap

Be your bridge keeper!

- While learning modern APIs are **optional** for artists, knowing computer graphics and the GPU are **not**!
- When using existing engines, be the bridge keeper and ask yourself **questions**. When you do not understand a term or parameter, use the documentation and go down the rabbit hole until you **understand the implications**.

https://montypython.fandom.com/wiki/Bridge_of_Death

https://produitabulles.wordpress.com/2016/05/01/the-tales-of-the-killer-rabbit/

Be your bridge keeper!

- While learning modern APIs are **optional** for artists, knowing computer graphics and the GPU are **not**!
- When using existing engines, be the bridge keeper and ask yourself **questions**. When you do not understand a term or parameter, use the documentation and go down the rabbit hole until you **understand the implications**.
- Overall takeaways:
    - **Start from core principles and not APIs or engines** → don't rush into engine features or graphics APIs and instead understand what the pipeline does. Understanding beats memorizing!
    - **Find the origin** → learning why techniques evolved to what they are now will give you a deeper understanding, and a bigger toolbox to make informed decisions.
    - **Build something small to see how it works** → this can be a software rasterizer, but also a simple memory allocator, texture loader, toy shader, etc.
    - **Think in terms of data flow and cost** → every vertex, texture, draw call, and more has a cost. Learn to trace how data moves and where time is spent. Understand that performance is the natural consequence of understanding flow.
    - **Stay curious** → understanding grows by tinkering and doing things. When you find something new, take time to understand it.
    - **Stay interactive** → while framerate and performance is important, test your art and code in motion!
    - **Learn from artists/programmers** → always learn from your "counterpart"! Take an additional class if needed.

The strongest bridges are built on deep foundations.

Not by learning every feature but by understanding the **core principles** that make everything else make sense.

The concepts that intimidate you are the ones worth learning. Just take it one step at a time.

Every expert was once a beginner who refused to give up!

Thank You!

matthieu.delaere@howest.be