Hello everyone, I'm Erfan, a Lead Software Engineer at DevSH Graphics Programming

Welcome to this talk about porting n4ce's old gdi renderer to vulkan, we have a lot of ground to cover but let's first talk about n4ce

n4ce

- Survey Processing and Modelling Tool

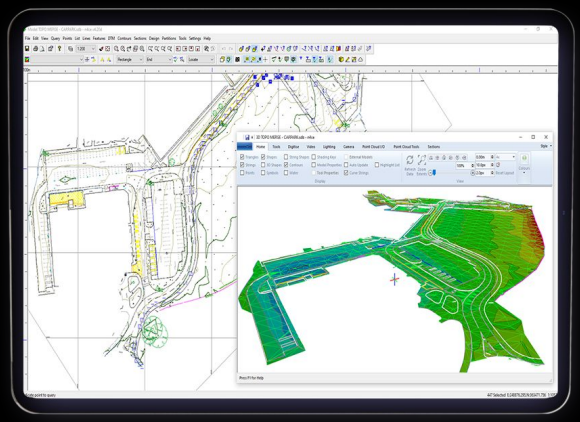


Image source: https://housing.com/news/total-station/

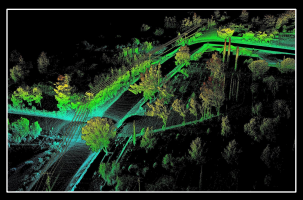
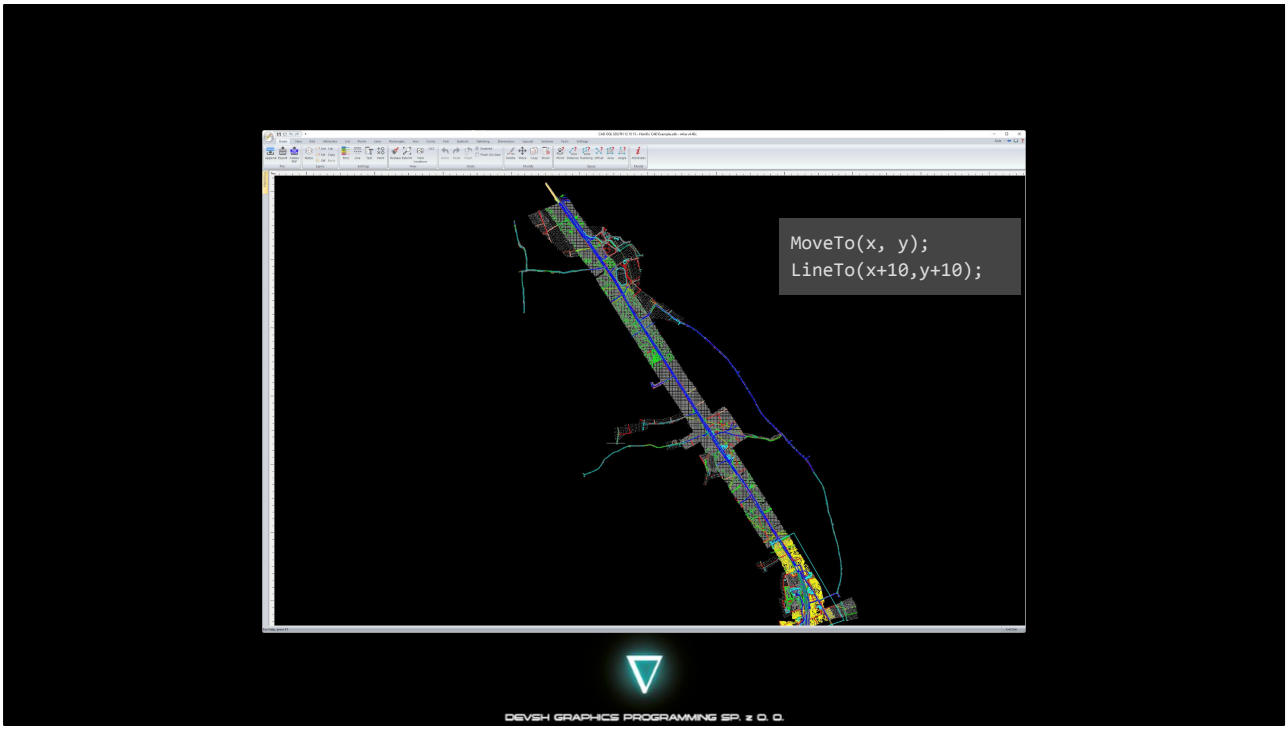


Image source: https://resources.trimble.com/resources/surveying

DEVSH GRAPHICS PROGRAMMING SP. z O. O.

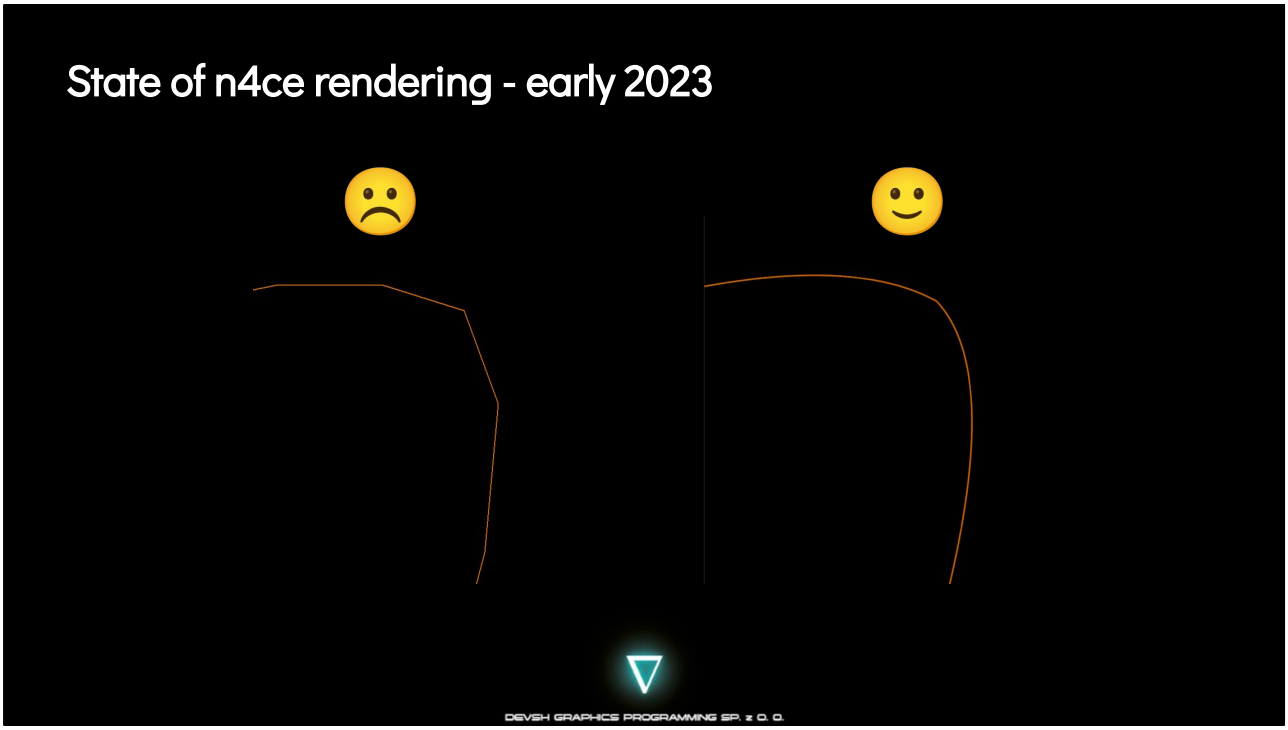
n4ce is a survey processing and modelling tool.
Survey mapping is an essential process in the development and management of any piece of land and
Software such as n4ce have been around since the 1980s and their capabilities have evolved to take measurements from all survey instruments such as total stations, GNSS, LiDAR point clouds



And so the size of the survey data kept on getting larger and larger, reaching millions of curves and points.

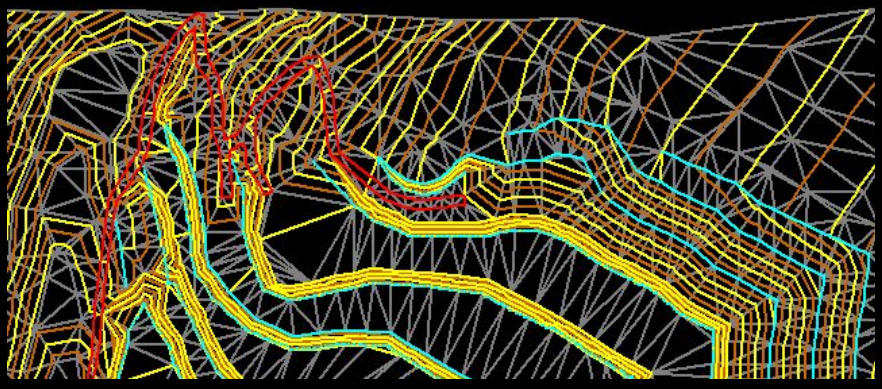
While n4ce kept up with the state of the art feature extraction using Machine Learning on CUDA, the 2D rendering retained a legacy implementation in the Windows Graphics Device Interface

GDI is basically an old immediate mode renderer for windows, you draw lines by moving markers around.



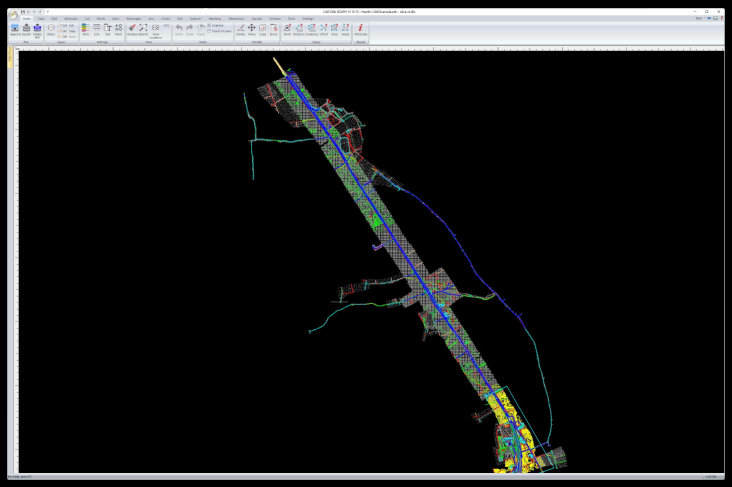
We noticed two visual deficiencies that we wanted to fix:
Curves had to be polygonized before feeding to GDI, this results in considerable error with the original definition of a curve and we have no smoothness
And the biggest problem is that you have to have a lot of straight lines to reasonably approximate the original curve

State of n4ce rendering - early 2023



And there is aliasing.

State of n4ce rendering - early 2023



Did I mention this survey takes 22 seconds to render a single frame with GDI. It contains more than half a million lines and curves combined.

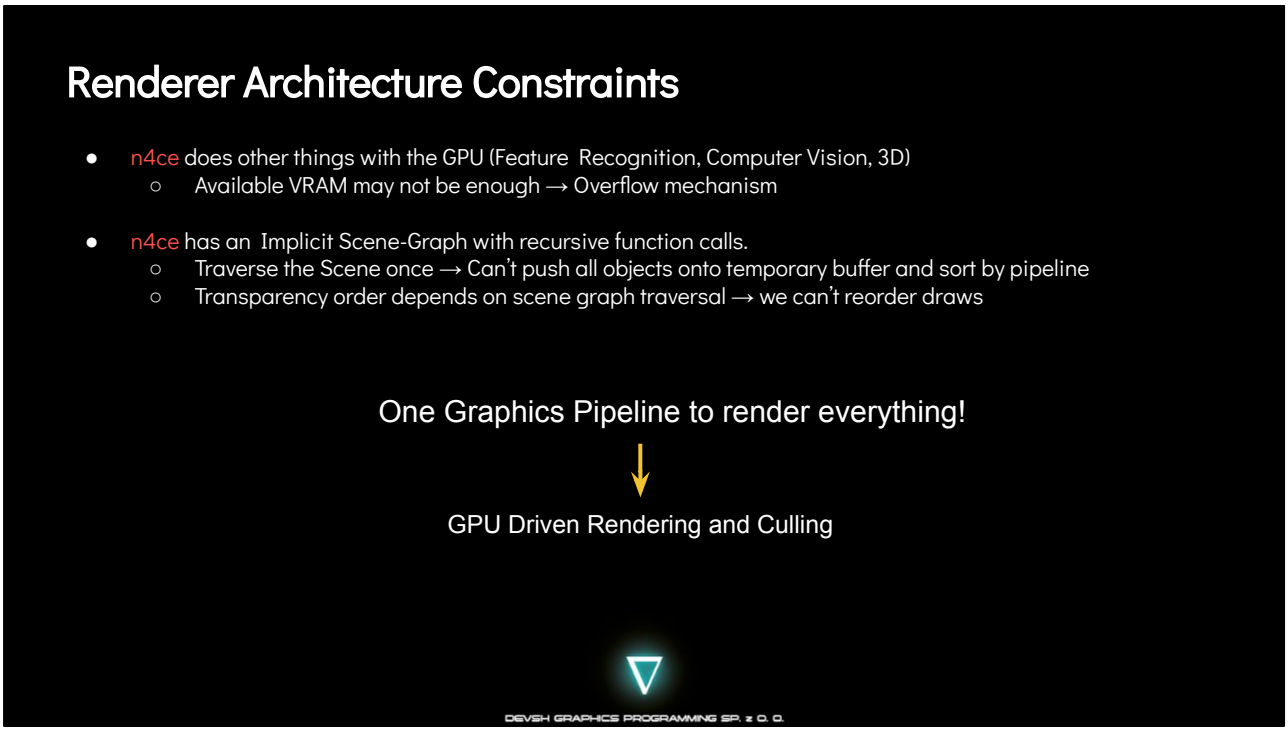
Other than GDI there were still a lot of performance pitfalls such as large temporary memory allocations and not caching polygonized data

Target with Vulkan on GPU

- **Faster**
  - heavy scenes interactive/real-time
- **Higher Quality**
  - Anti-Aliasing, Smooth Curves
- **Less Error** === Less Difference from Actual Shape

So We want the new Renderer to be “Faster” obviously, for our heavy scenes to be interactive, higher quality with anti-aliasing, transparency and smoother curves

And we want it to have less error



N4ce is a 20 year old codebase coupled with immediate mode rendering, there is going to be limitations on our rendering architecture,

1. n4ce does other things with the GPU so Available VRAM is not enough for the data we need access to on the GPU; there should be an overflow mechanism
2. More importantly n4ce has an implicit scene graph with recursive function calls and we cannot change this and re-engineer the entire application.

- we want to traverse the scene only once, but there are too many draws to push into temp buffer and sort by pipeline
- we can't reorder draws because transparency order depends on the scene graph traversal

So the best solution we came up with is to have 1 graphics pipeline to render every object!

This choice of having 1 pipeline also makes GPU Driven Rendering and Culling achievable because we no longer have an unknown number of pipeline switches

Target with Vulkan on GPU

- Windows Desktop with still supported GPUs:
  - Required
    - shaderClipDistance → customized clipping
    - scalarBlockLayout → HLSL and C++ struct Compatibility
  - Optional
    - shaderFloat64 → double precision coordinates needed in CAD
      - We need to emulate it for Intel GPUs
    - bufferDeviceAddress → single buffer multiple types
      - Can be emulated using different descriptors on the same buffer binding
    - fragmentShaderInterlock → our transparency algorithm

And we are targeting windows desktop with GPUs supporting following vulkan features

shaderClipDistance for customized clipping
scalarBlockLayout for HLSL/C++ struct Compatibility

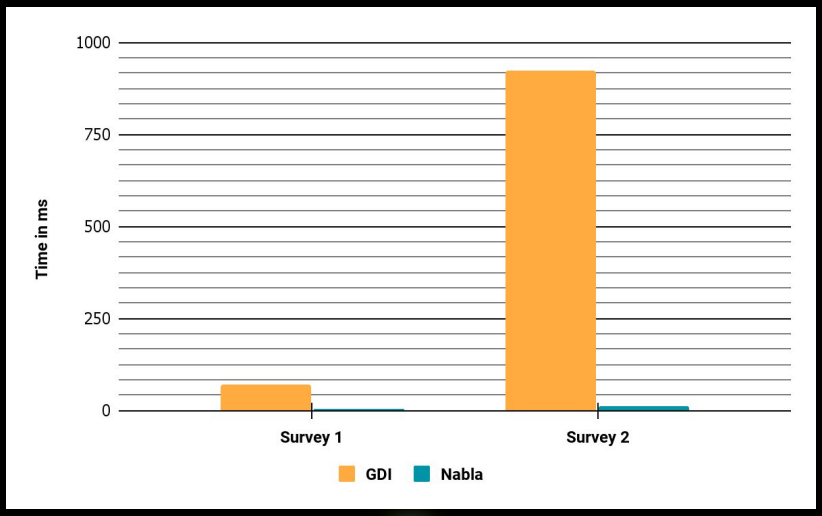
shaderFloat64 because we need to work with double precision coordinates in CAD, but this feature is not available on IntelGPUs so we need to emulate when this feature is not present

bufferDeviceAddress so we can have single buffer to store data with multiple types, it can be emulated using different descriptors on the same binding

fragmentShaderInterlock for our transparency algorithm explained later in this talk

Here is what we were able to achieve so far, quality-wise, as you can see the aliasing is gone

Performance



2560 × 1600 Resolution
AMD Ryzen 7 6800HS 3201 Mhz, 8 Core(s), 16 Logical Processor(s)
NVIDIA GeForce RTX 3070 Ti Laptop GPU

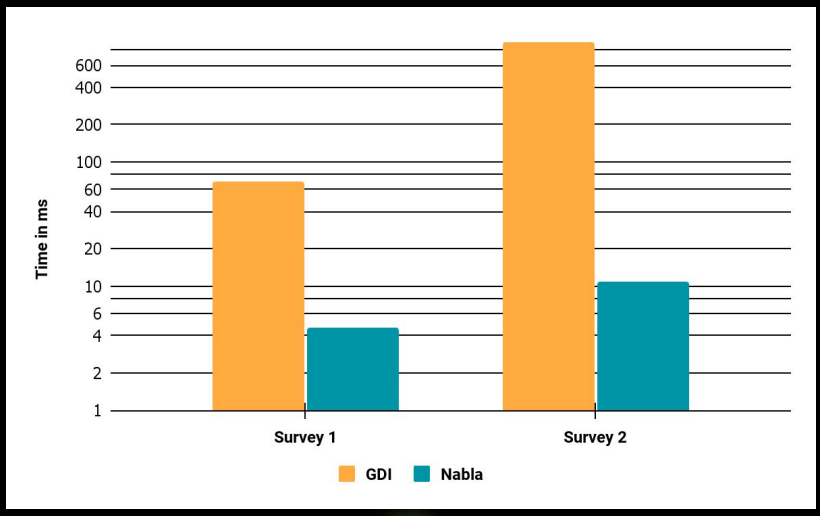
DEVSH GRAPHICS PROGRAMMING SP. z O. O.

And here is our performance comparisons for two simple surveys with a few thousands of lines and curve

Orange is before, Green is after

Timing is in milliseconds to render a single frame, so the less is better

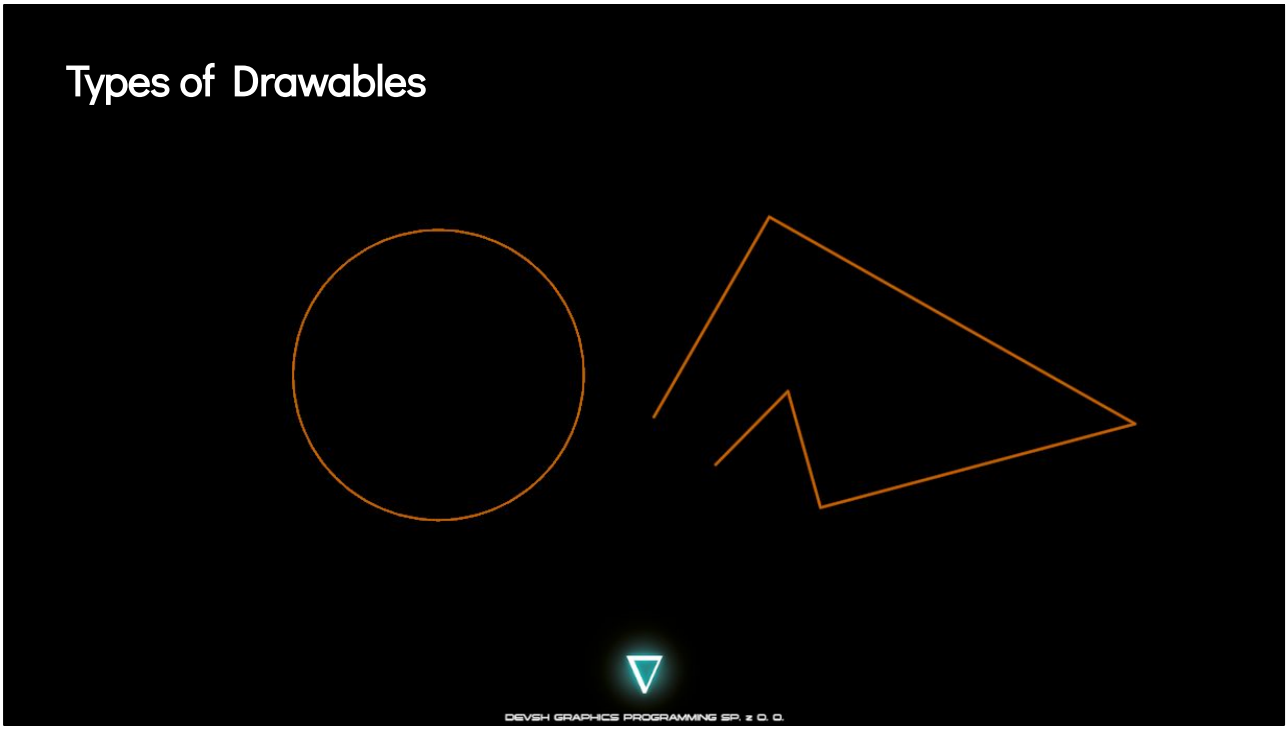
Performance



2560 × 1600 Resolution
AMD Ryzen 7 6800HS 3201 Mhz, 8 Core(s), 16 Logical Processor(s)
NVIDIA GeForce RTX 3070 Ti Laptop GPU

DEVSH GRAPHICS PROGRAMMING SP. z O. O.

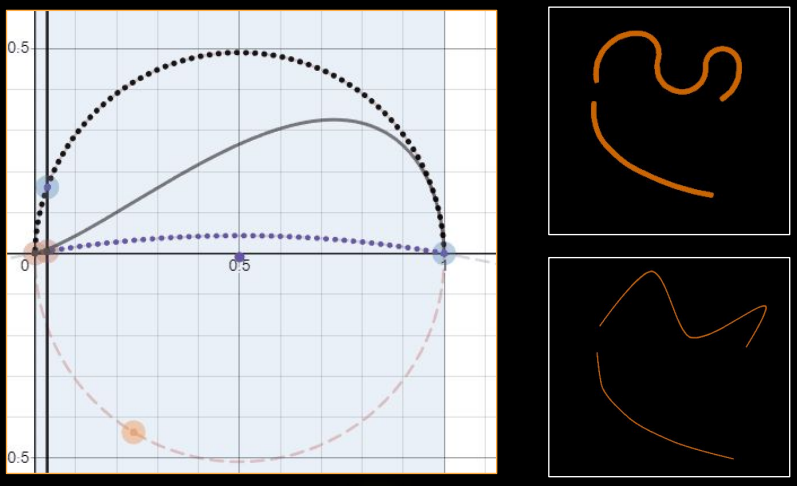
Ok I need to bring it to logarithmic scale, what takes n4ce almost a second, only takes 11 milliseconds on GPU, both include the time spent on the CPU to construct the geometries from scene data without caching them.



Before getting into how our rendering pipeline is structured we should first define what's the problem we're trying to solve, what kinds of objects are we trying to render.

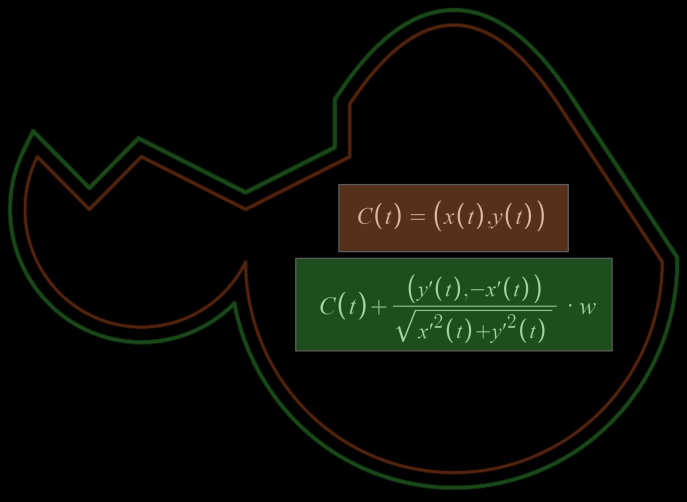
We have straight lines and elliptical arcs

Types of Drawables



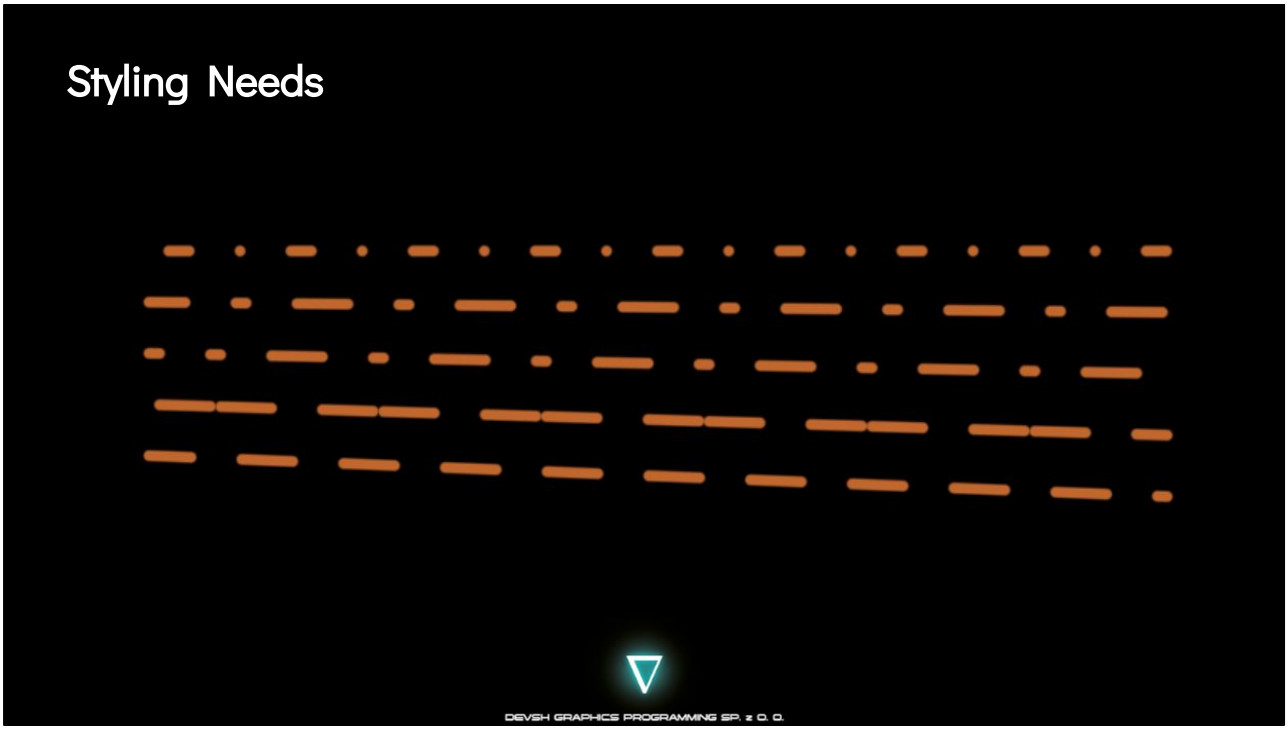
And then we have some complex curves, they are a blend of two parabola or two elliptical curves along a chord

Types of Drawables



And we have offsets to each of the curve types

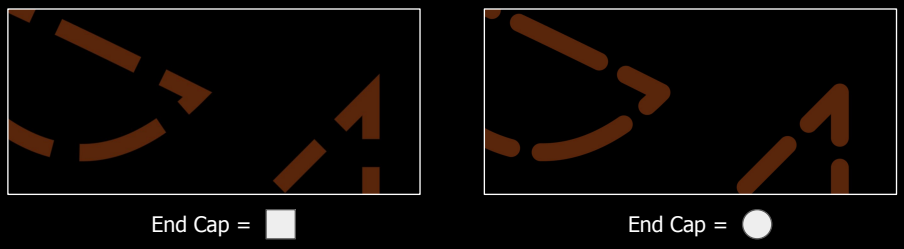
An offset curve is the original curve added to it's normal, so it's not a simple transformation, it changes the whole mathematical definition, so it's an added complexity for each new curve type we introduce



It doesn’t end there, each of those curves need custom stylization, they need dot-dash patterns applied to them, also known as stipple patterns

Styling Needs

- Cannot use GL-style 16bit stipple patterns
  - styles are vectorized and have infinite resolution (0.0 in pattern)
  - Custom end caps and mitering
  - Constant width in world-space



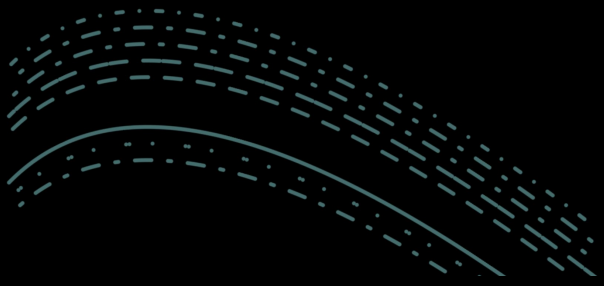
We cannot use Hardware line primitives with Gl-Style 16 bit stipple patterns for several reasons,

- The styles are vectorized and have infinite resolution
- lines have custom end caps,
- Line width can be in worldspace

Styling Needs - Arc Length

- Do Not Want to polygonize complex curves like the initial stage of legacy GDI backend

$$L = \int_a^b \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt$$



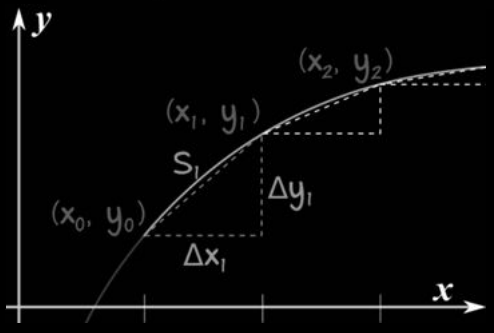
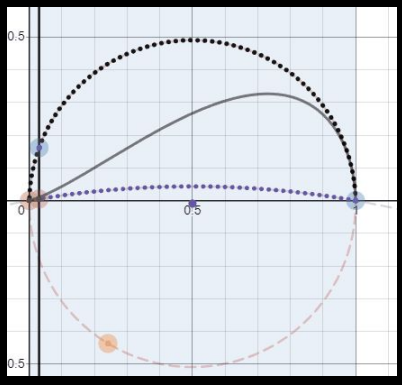


Image from mathisfun.com

So in order to achieve that without breaking our curves into multiple smaller geometries we need to compute arc length and its inverse for our curves, because we need a mapping between the position on the curve and the position on the pattern.

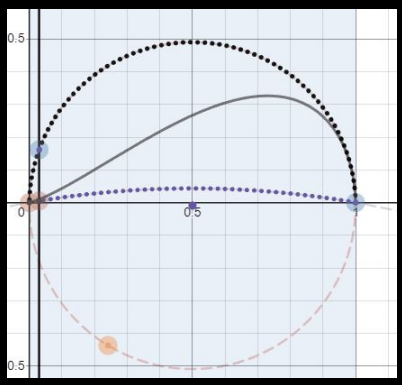
Remember calculus?

Styling Needs



Let’s look at our blended elliptical curves again.

Styling Needs



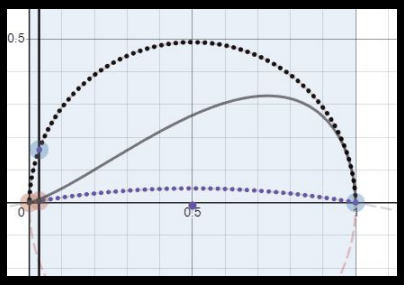
$$m(x) = h(x) \cdot \big(y_2(x) - y_1(x)\big) + y_1(x)$$

Interpolation value function [0,1]

Curves we want to blend

Here is their general equation

Styling Needs

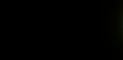


$$m(x) = h(x) \cdot \big(y_2(x) - y_1(x)\big) + y_1(x)$$

Interpolation value function [0,1]

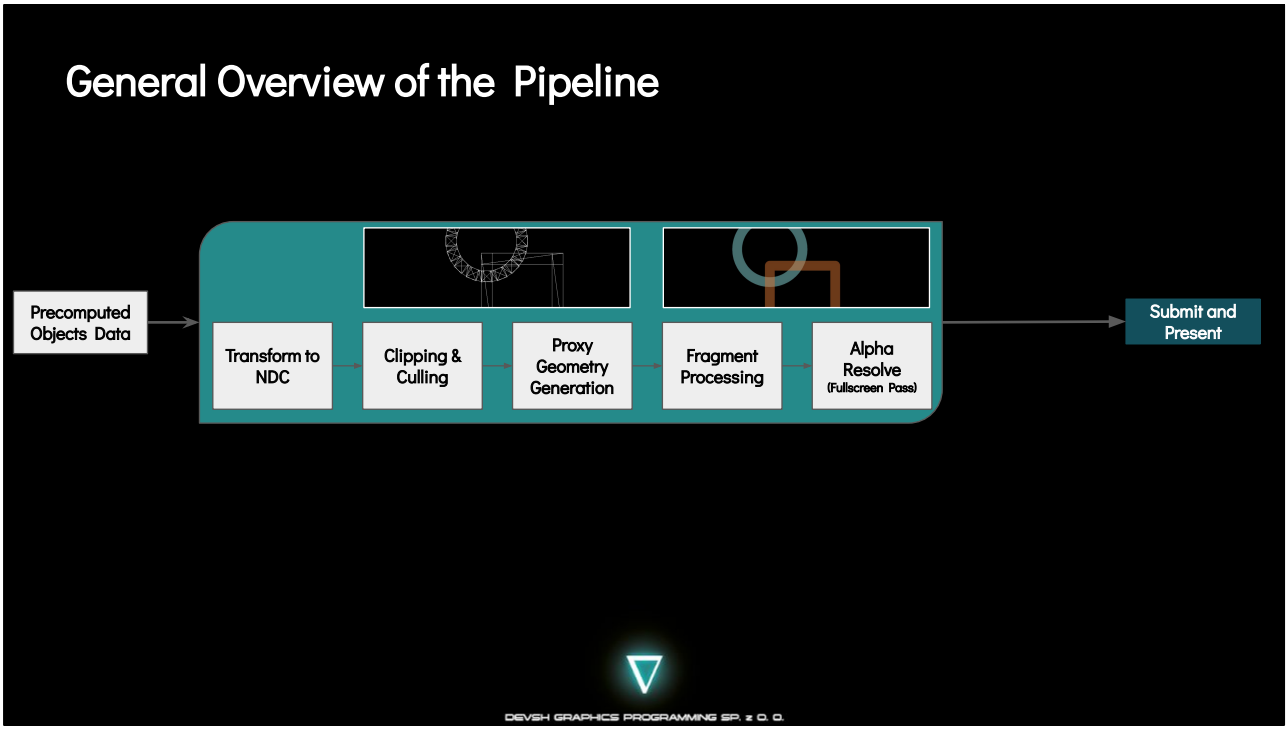
Curves we want to blend

$$\int_{t_1}^{t_2} \frac{2\left(\frac{s_1 \cdot x}{\sqrt{r_1^2 - x^2}} - \frac{s_2 \cdot x}{\sqrt{r_2^2 - x^2}}\right)}{l} + \left(\frac{x}{l} + \frac{1}{2}\right)\left(s_2\left(-\frac{x^2}{\left(r_2^2 - x^2\right)^{\frac{3}{2}}} - \frac{1}{\sqrt{r_2^2 - x^2}}\right) - s_1\left(-\frac{x^2}{\left(r_1^2 - x^2\right)^{\frac{3}{2}}} - \frac{1}{\sqrt{r_1^2 - x^2}}\right)\right) + s_1\left(-\frac{x^2}{\left(r_1^2 - x^2\right)^{\frac{3}{2}}} - \frac{1}{\sqrt{r_1^2 - x^2}}\right) dx$$



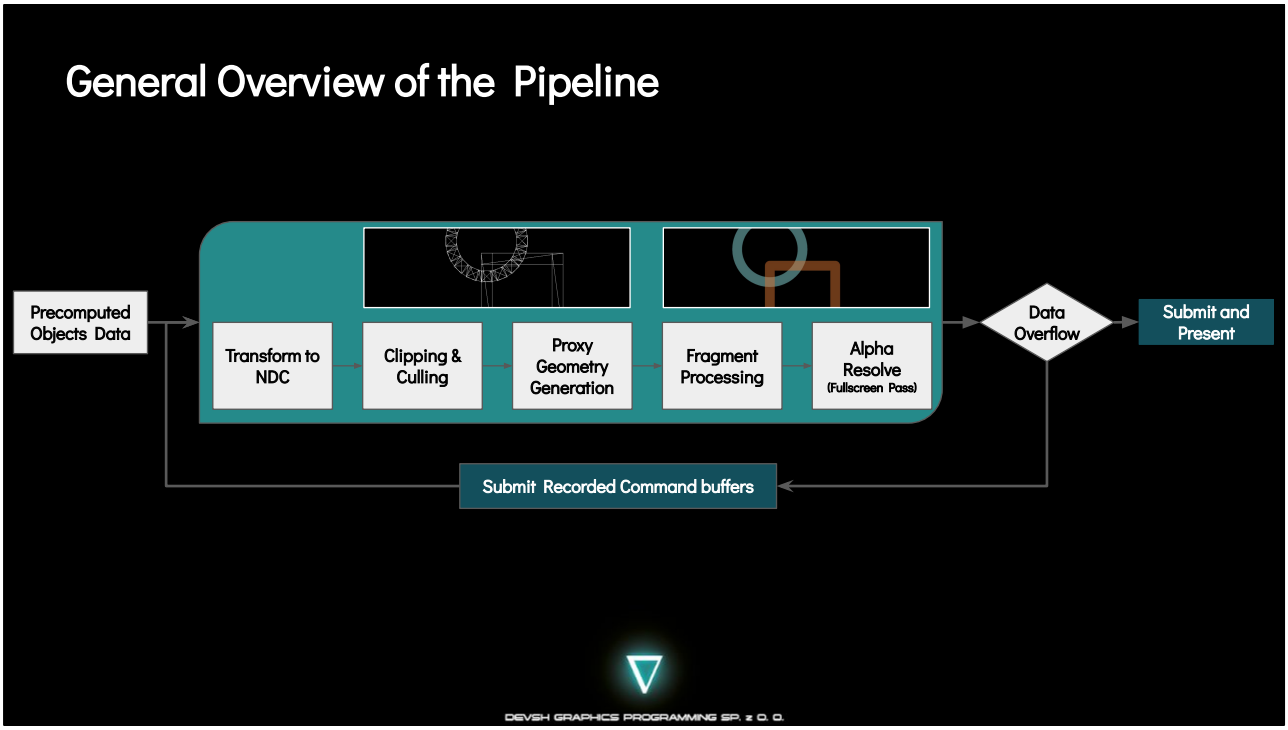
And here is their arc length. I only used this example because the equation would fit in the screen, but it is enough to make the point that these are complex curves we're dealing with.

We have to go through this process for every new curve type we introduce, and the associated offset curve counts as a new type by itself.

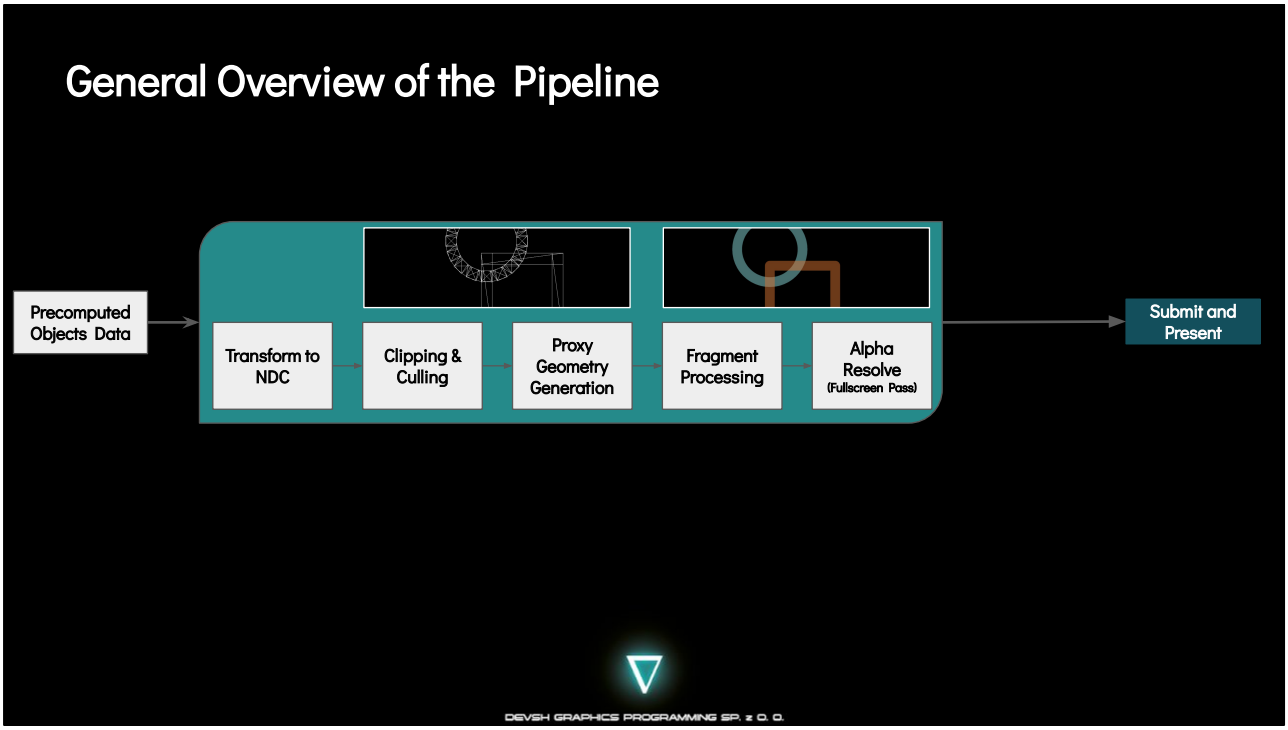


Now we're ready to begin walking through out the rendering pipeline:

- We start by transforming our precomputed curve data to normalized device coordinates and clip them
- Then we generate proxy geometry which tightly bounds our curves and we feed them to fragment shader to render the curves
- There is also a fullscreen alpha resolve pass we will get into when we get to explaining transparency, and finally we submit and incrementally present everything

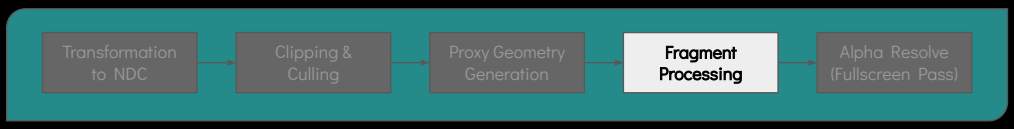


Of course there is an overflow mechanism because sometimes not all of the precomputed object data can fit into VRAM so we will submit and start processing the next chunk.
If you are curious, take a look at our Vulkanised 2023 talk “Keeping your staging buffer fixed size’



but let's ignore that for simplicity

Fragment Processing

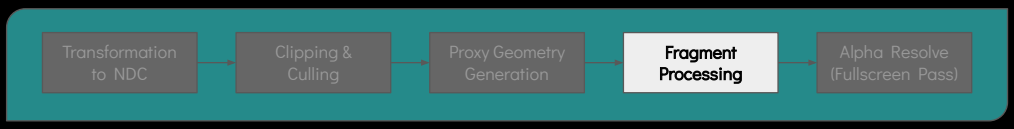


1. Applying styles in the fragment-shader, to reduce:
   - Polycount
   - Cycles spent by CPU on serial geometry generation
   - Staging buffer bandwidth

Let's focus on the fragment processing part of the pipeline:

- Firstly, we want to render our curves and apply styling in the fragment shader which essentially reduces the cycles spent by cpu on serial geometry generation and reduces the staging buffer bandwidth

Fragment Processing



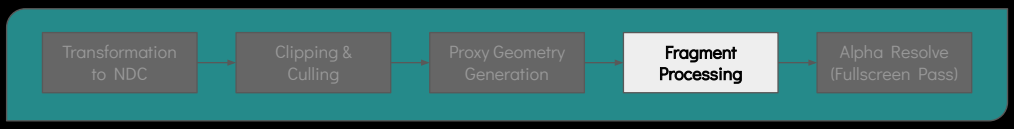
1. Applying styles in the fragment-shader, to reduce:
   - Polycount
   - Cycles spent by CPU on serial geometry generation
   - Staging buffer bandwidth
2. But want Anti-Aliasing:
   - Super-Sampling is Expensive (duh :/)
   - MSAA is useless without actual geometry
   - AlphaToCoverage useless because of sample correlation →
   - Single Graphics Pipeline for all draws → blending always ON
     - So let's compute the Coverage analytically

Wyman et al 2017

We also want to do anti-aliasing

- Super sampling is expensive, we're doing most of the rendering work in fragment shader
- MSAA is useless without actual geometry since it's for geometric anti-aliasing and not interior
- AlphaToCoverage useless here too, it's used for screen door transparency or for anti-aliasing alpha test such as in grass rendering in games
- Remember that we have 1 graphics pipeline, and we have blending always enabled for that pipeline because of transparency
  - so let's compute the coverage analytically and set the pixel's alpha

Signed Distance Field Rendering



- Signed Distance Field
  - Signed → negative means inside, positive means outside of boundary
  - Distance → to the boundary
  - Field → defined for every point in space

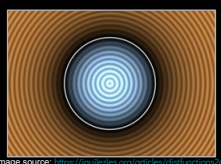


Image source: https://iquilezles.org/articles/distfunctions2d/

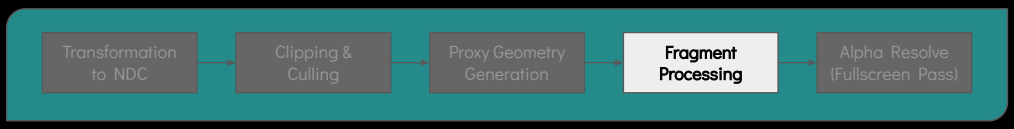
DEVSH GRAPHICS PROGRAMMING SP. z O. O.

We decided to do that using Signed Distance Fields to render our curves and lines.

SDFs are:

- Signed because negative is inside of a shape, and positive outside
- And they tell you the distance to the boundary
- Field, because its defined in every point in space, 2D or 3D

Signed Distance Field Rendering



- Signed Distance Field
  - Signed → negative means inside, positive means outside of boundary
  - Distance → to the boundary
  - Field → defined for every point in space
- SDFs allow for a good approximation of coverage

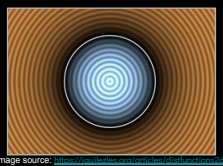
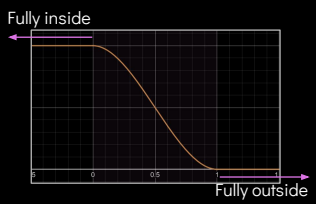


Image source: https://iquilezles.org/articles/distfunctions2d/

```
float alpha = 1.0f - smoothstep(-1.0, +1.0, sdf);
```



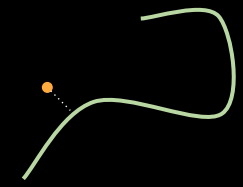
Sdfs allow for a good approximation of coverage, we deduce pixel coverage from sdf value using smoothstep because it has nice properties.

Proxy Geometry Generation

- Computing SDFs are computationally heavy

D(t) = Distance(point, curve(t))

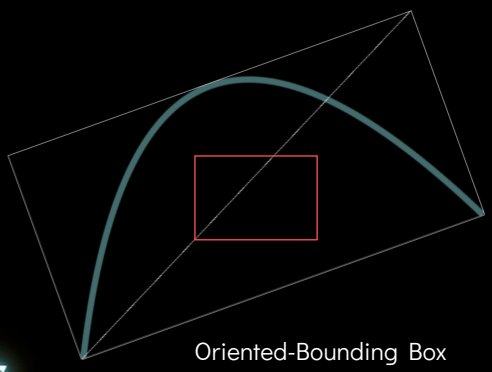
Solve D'(t) = 0 to find minimums



But computing SDFs is expensive, as it requires finding the local minima of the function that tell you the distance to the curve from the query point.
And basic calculus tells us to find local minima of a function we need to differentiate it
The more complex the curve the more complex the root finding on function D'.

Proxy Geometry Generation

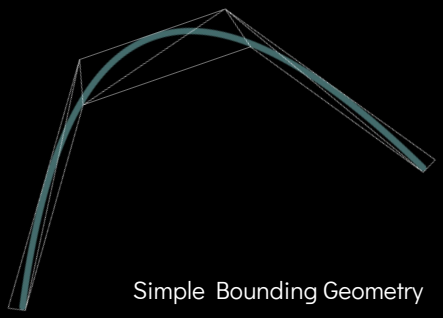
- Precomputed Bounding Geometries are not good enough
  - Clip Curves before
  - Generate Proxy Geom afterwards



That's why we want to tightly bound the curve to reduce the number of fragment shader invocations that do not contribute to anything and redundantly perform this SDF computation.
Precomputed Bounding Geometry won't work, imagine you zoom in on a small part of this parabola's that's off the curve, now it covers the entire screen; and the sdf is evaluated for every pixel on the screen.
Even Oriented Bounding Boxes have a poor overdraw ratio.

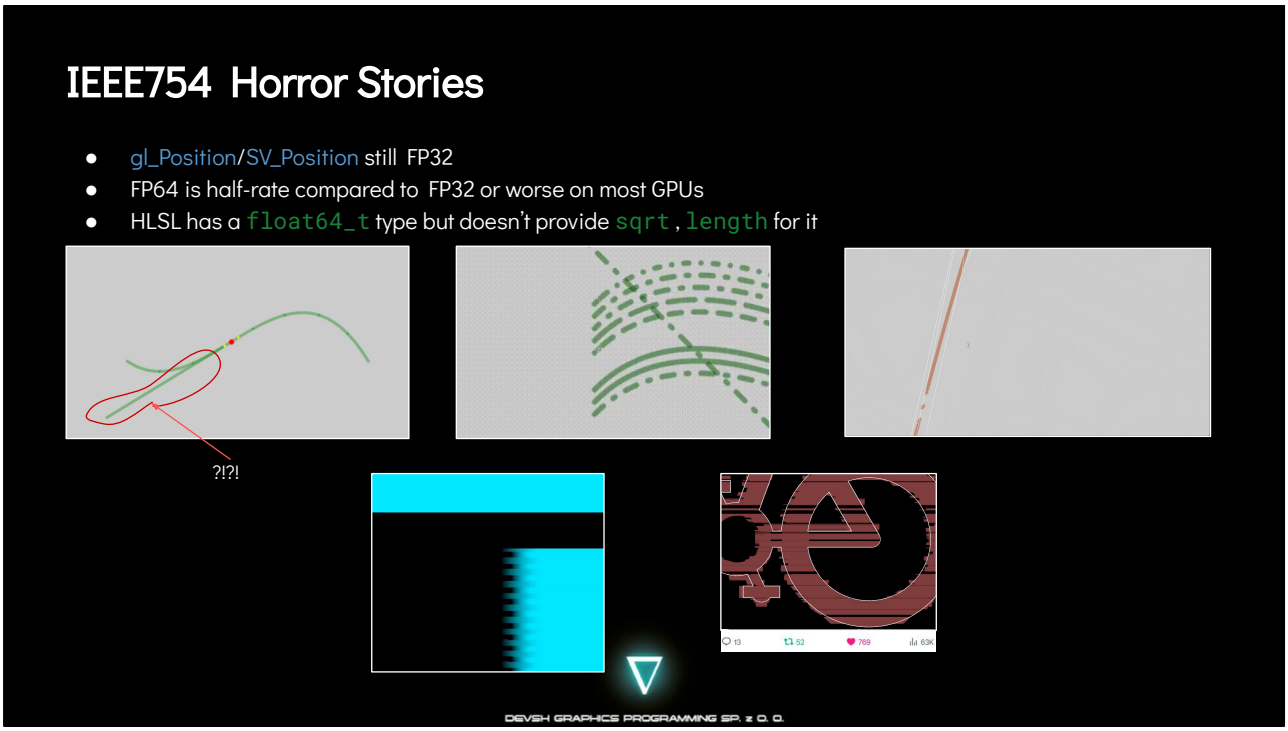
Proxy Geometry Generation

- Precomputed Bounding Geometries are not good enough
  - Clip Curves before
  - Generate Proxy Geom afterwards
- Tighter Bounds
  - requires finding extremum points



Simple Bounding Geometry

That's why we decided to bound our curves with custom bounds, like the image you see on the screen.
But it's not as simple as you think and requires finding extremum points of a curve



So far, we've seen that each curve type has different equations for SDF, proxy geometry gen and arc length which require individual analysis of singularities where precision breaks down.

Here is a photo album of what we were dealing with regarding precision issues

Keep in mind that we want to avoid doubles in the fragment shader at all costs. because:
- `gl_Position` still 32-bit floating point
- 64-bit floating point is half-rate compared to 32 or even worse on most GPUs
- HLSL has a float64_t type but doesn't provide important functions such as sqrt, length we need, and implicitly casts to 32-bit floating point

All that for a single Curve Type!

1. Generate a tight bound polygonal mesh (Proxy Geometry)
2. Calculate the Signed Distance Field function
3. Compute Arc Length (to map curve position to pattern)
    - Requires finding Derivative of the curve
4. Compute Inverse Arc Length (to map pattern to curve position)
5. (even more: computing curve-curve intersections)

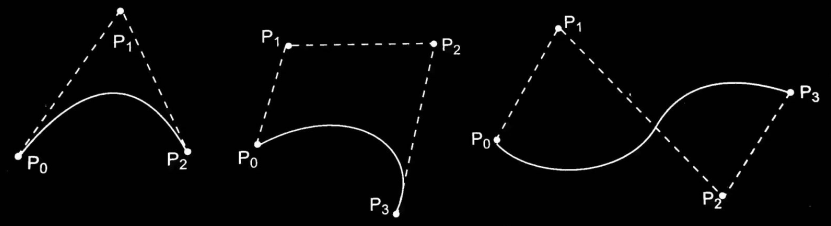
All of the above need to go through error analysis, tests and be made numerically stable.

So for each curve type we have to
1. Generate a tight bound polygonal mesh
2. Calculate the SDF function
3. Compute Arc Length which Requires finding Derivative of the curve
4. Compute Inverse Arc Length
(there is even more we won't get into)

And All of the above need to go through error analysis, tests and be made numerically stable.

Choosing Beziers as the main representation

1. The properties, and the precision of their computation are better studied
2. Numerical Stability achievable
3. Having a single curve type is heaven for Maintainability
    - Faster Development Time



www.allbca.com

Fig. Bezier curves.

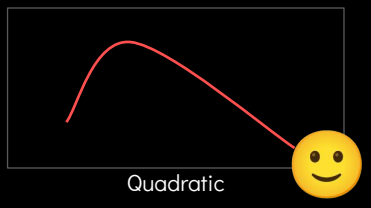
That's why we chose to turn any complex curve into piecewise beziers. That gives us HUGE benefits:

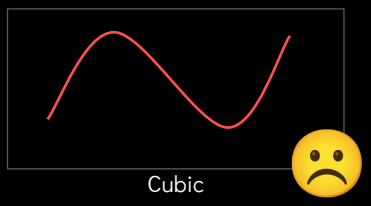
- The properties, and the precision of their computation are better studied as there are thousands of articles on bezier
- Numerical Stability is now achievable much easier
- Having a single curve type is heaven for Maintainability and results in a Faster Development Time in the long run

Choosing Quadratic Beziers

Quadratic vs Cubic

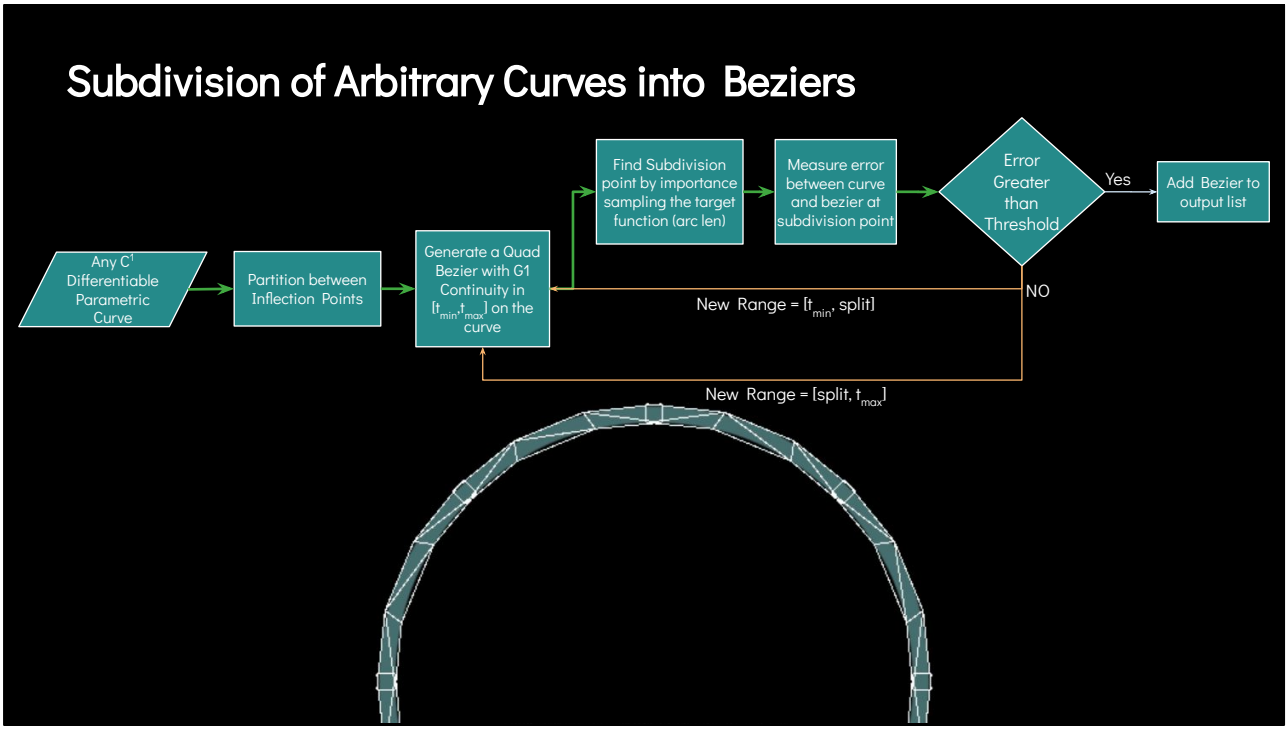
1. Quadratic SDF more numerically stable
    - Solving a lower degree polynomial
2. Arc Length of a Quadratic Bezier can be computed analytically
    - Makes numerical Inverse Arc-length faster
3. Intersection of
    - Two Quadratic Beziers is Quartic (Degree 4) equation which can be solved analytically
    - Two Cubic Beziers is Nonic (Degree 9) which need numerical methods
4. Rendering Quadratics 2x Faster than Cubics
    - SDF evaluation 4x, but 2x as many Quadratics required





We chose to use Quadratic beziers even though Cubic beziers have a higher degree of freedom and can approximate a curve more precisely but at a higher cost:

- The SDF of a Quadratic is more numerically stable than for a Cubic
- Arc Length of a Quadratic can be computed analytically and this makes numerical inversion faster
- Intersection of two quadratic beziers can be solved analytically, but cubics require numerical methods.
- Our benchmarks also showed the quadratic SDF is 4x faster to evaluate compared to a cubic

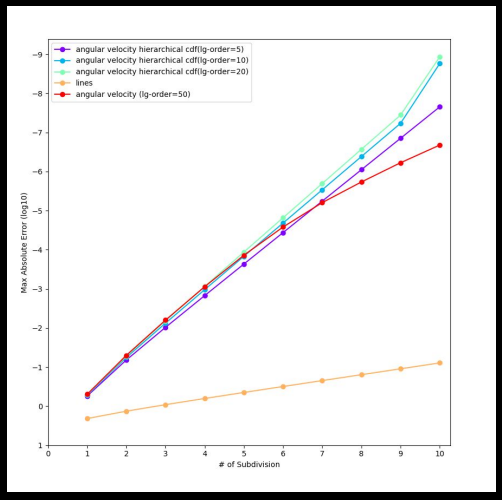


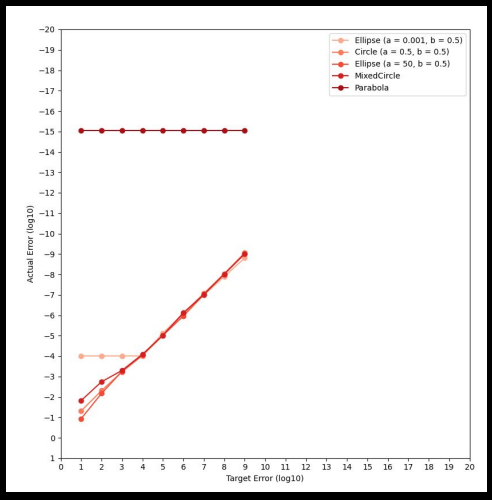
This is our pipeline which turns any C1 differentiable parametric curve into bunch of beziers until a certain accuracy is met, and this makes us future proof against new types of curves

This is just a fancy graph for a simple process which means subdivide until you don't need to anymore, the most important part is that we find the subdivision point by importance sampling the arc len, which really means with each subdivision we break the curve into two parts with equal arc length.

We also thought about importance sampling other functions such as curvature but those didn't work well.

Subdivision of Arbitrary Curves into Beziers



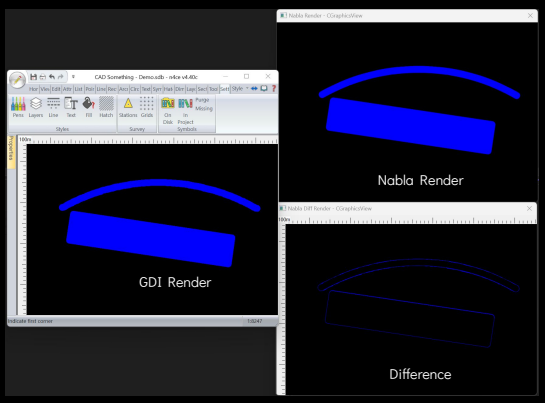


DEVSH GRAPHICS PROGRAMMING SP. z O. O.

In this case, 4 quadratic beziers result in higher accuracy than 1024 straight lines, and we can see we have almost a 45 degree slope in log scale which means with each subdivision level we cut down the error by a factor of 2

Vulkan Details - Built on Nabla

- Vulkan First Class Citizen API
- No Global State
  - Runs side by side other OpenGL/DirectX/Vulkan middlewares
    - Can interop resources with them
- Some Thread Safety
  - Functions can be called from any thread, most need external sync.
- No default/global Window
  - Spawn windows whenever you like
- All very useful for porting in-situ (side-by-side)



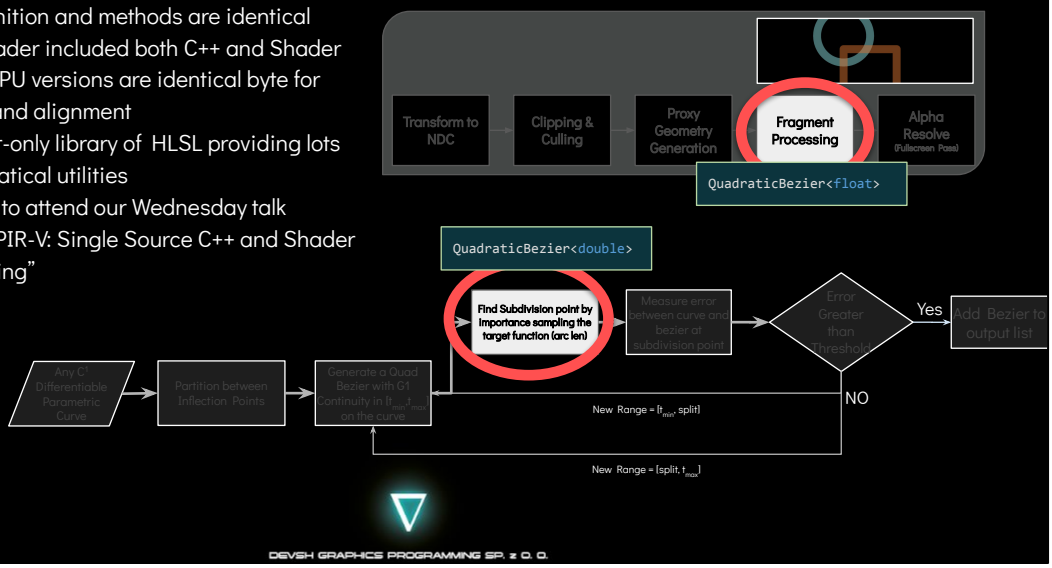
Now let's get in to more details on how everything is built on top of Nabla, our open-source Framework for working with Vulkan.

Nabla has no Global state, it can run side by side other APIs and even interop resources with them.
Functions can be called from any thread. most need external synchronization
There is no global window so you can spawn windows, surfaces and swapchains whenever you like.

All of this is very useful for porting in-situ where we can run them side by side even with their difference and notice any visual bugs during the development phase

Single Source HLSL/C++

- Struct definition and methods are identical
- Single Header included both C++ and Shader
- CPU and GPU versions are identical byte for byte, size and alignment
- Big header-only library of HLSL providing lots of mathematical utilities
- Make sure to attend our Wednesday talk "Beyond SPIR-V: Single Source C++ and Shader Programming"



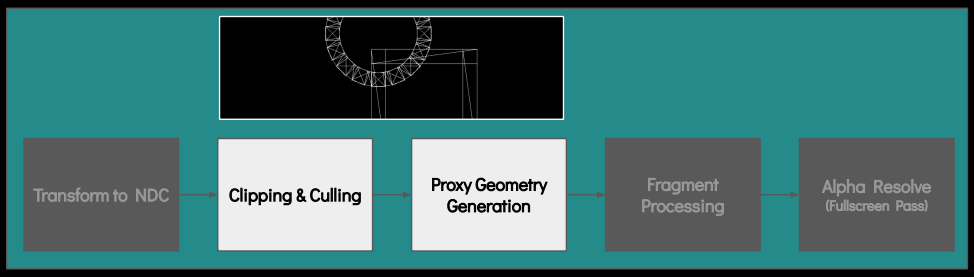
A standout feature for us in Nabla is sharing code with HLSL and C++;
helps a lot with maintaining a large project as there are a lot of similarities between C++ CPU code and the shaders on GPU

As an example we need to calculate the arc length of a bezier both in fragment shader for styling and in the subdivision pipeline for importance sampling subdivision points.

- One uses doubles, another uses float
- One is a preprocess step on CPU, one is runtime in the fragment shader

Single Source HLSL/C++

- Flip-flop between CPU and GPU Compute Culling and Proxy Geometry generation



Single source HLSL/C++ also allows us to be flip-flop between CPU and GPU for Culling or Proxy Geometry generation at a moments notice.

Single Source HLSL/C++

- Step-through and debug simple functions on x86
- Unit Test without a GPU !

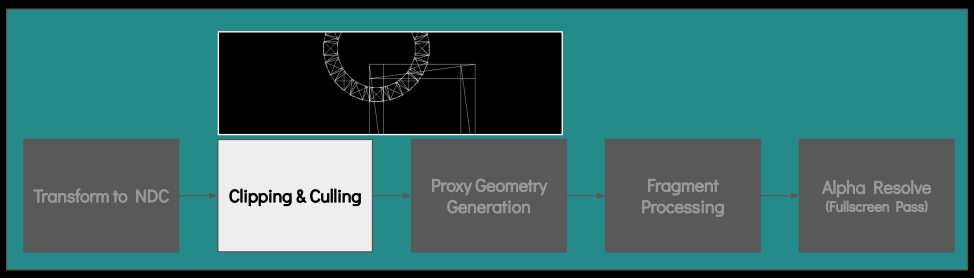
```
template<typename float_t>
struct QuadraticBezier
{
    using float_t2 = vector<float_t, 2>;
    float_t2 P0;
    float_t2 P1;
    float_t2 P2;

    float_t2 evaluate(float_t t)
    {
        float_t2 position =
            P0 * (1.0 - t) * (1.0 - t)
            + 2.0 * P1 * (1.0 - t) * t
            + P2 * t * t;
        return position;
    }
}
```

As a bonus, you can debug and prototype shaders on the CPU without expending effort to think about setting up descriptors and pipelines.

Global Compute and Workgroup reusable Scan & Sort Primitives

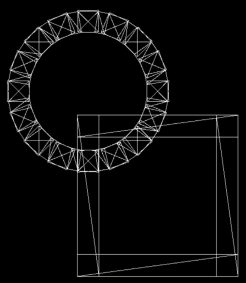
- Alpha Blending → Output of the compute culling stage to maintain the ordering of the input
  - GPU Radix Sort



Because of the alpha blending, we need the output of the culling stage to maintain the ordering of the input, this is easy for a serial CPU implementation but is tricky on the GPU.
A Parallel Prefix Sum a.k.a a Scan, performed on a global array, is an indispensable building block of a GPU Radix sort or a Compaction.
And nabla helps with that by providing Global Compute and Workgroup reusable Scan & Sort Primitives

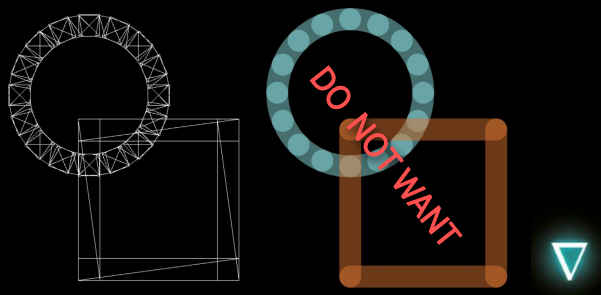
Transparency and Anti-aliasing

- Alpha depends on Coverage and overall desired transparency
- Proxy Geometries are Conservative so they overlap



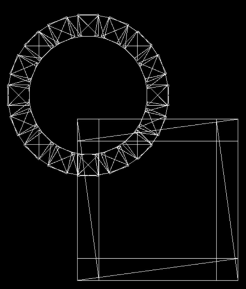
Transparency and Anti-aliasing

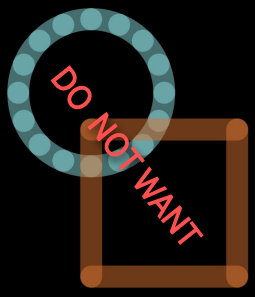
- Alpha depends on Coverage and overall desired transparency
- Proxy Geometries are Conservative so they overlap
- Beziers for the same Group ID should not overblend on their intersections
    - Could be solved by careful mitering where these beziers touch

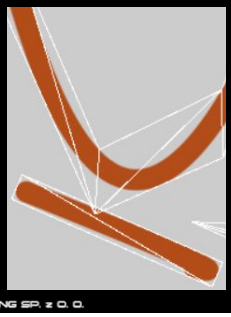


Transparency and Anti-aliasing

- Alpha depends on Coverage and overall desired transparency
- Proxy Geometries are Conservative so they overlap
- Beziers for the same Group ID should not overblend on their intersections
  - Could be solved by careful mitering where these beziers touch
  - But we fallback to OBB Proxy Geometry when `curvature*lineWidth > 4`
    - Reason: Hard to find where the offset curve gets too close to the original bezier

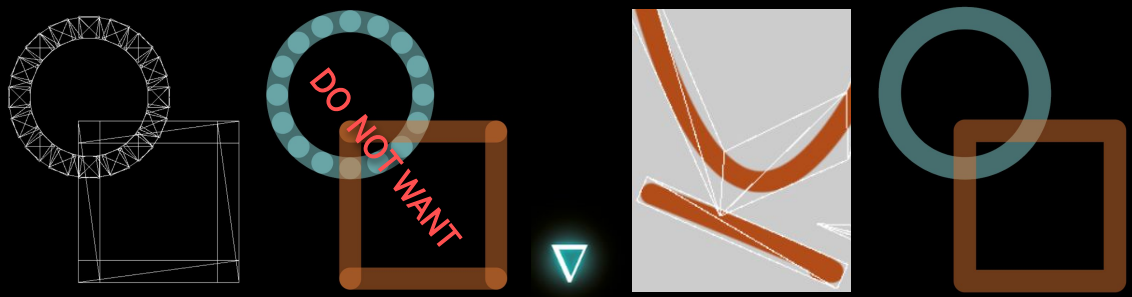






Transparency and Anti-aliasing

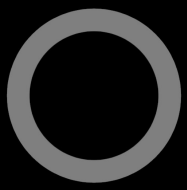
- Alpha depends on Coverage and overall desired transparency
- Proxy Geometries are Conservative so they overlap
- Beziers for the same Group ID should not overblend on their intersections
  - Could be solved by careful mitering where these beziers touch
  - But we fallback to OBB Proxy Geometry when `curvature*lineWidth > 4`
    - Reason: Hard to find where the offset curve gets too close to the original bezier
- Solution: Invent a new, never seen before algorithm!



Framebuffer-Space Constructive Solid Geometry with Signed Distance Fields

We begun with a “what-if” idea of drawing Proxy Geometry twice:

1. Color write **OFF**, Stencil Pass **ALWAYS** and Stencil Op **MAX** to accumulate coverage values
    - Would need a programmable Stencil Export value and a late Stencil Test



Framebuffer-Space Constructive Solid Geometry with Signed Distance Fields

We begun with a “what-if” idea of drawing Proxy Geometry twice:

1. Color write **OFF**, Stencil Pass **ALWAYS** and Stencil Op **MAX** to accumulate coverage values
   - Would need a programmable Stencil Export value and a late Stencil Test
2. Color write **ON**, Stencil Pass **GREATER** than 0 and Op **CLEAR**, use the Stencil Buffer to modulate alpha
   - Only the first drawn pixel “resolves” the colour, rest are dropped
   - But you cannot obtain the current value from a bound stencil attachment!

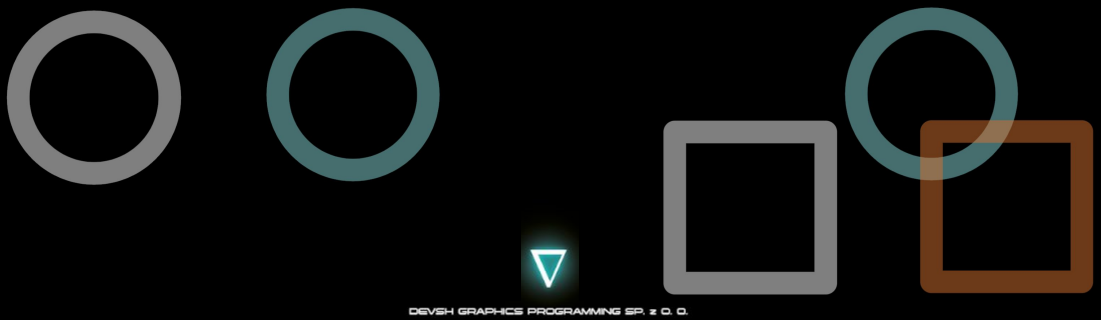
Framebuffer-Space Constructive Solid Geometry with Signed Distance Fields

We begun with a "what-if" idea of drawing Proxy Geometry twice:

1. Color write **OFF**, Stencil Pass **ALWAYS** and Stencil Op **MAX** to accumulate coverage values
   - Would need a programmable Stencil Export value and a late Stencil Test
2. Color write **ON**, Stencil Pass **GREATER** than 0 and Op **CLEAR**, use the Stencil Buffer to modulate alpha
   - Only the first drawn pixel "resolves" the colour, rest are dropped
   - But you cannot obtain the current value from a bound stencil attachment!

Also didn't we say one pipeline and one draw call for everything?

- Separate Backface and Frontface stencil state + permuted index buffers still doesn't solve all of the above



What if we had a Programmable Stencil Buffer? Like Programmable Blending?

Enter Fragment Shader Interlock!

What if we had a Programmable Stencil Buffer? Like Programmable Blending?

- Fragment Shader Interlock allows for a single Critical Section in the Fragment Shader which is entered into w.r.t. Rasterization Order allowing synchronized access to any resource and can be used to implement:

Enter Fragment Shader Interlock!

What if we had a Programmable Stencil Buffer? Like Programmable Blending?

- Fragment Shader Interlock allows for a single Critical Section in the Fragment Shader which is entered into w.r.t. Rasterization Order allowing synchronized access to any resource and can be used to implement:
  - DX12-style Raster Ordered Views
  - Programmable blending of non-HW-blendable formats like RGB9E5 (or your own)
  - Complex Per-Pixel Data Structures which need in-order atomic modifications
  - Single Pass Order Independent Transparency such as Multiple Layer Alpha Blending
    - MLAB can be implemented with a spinlock (not UB) but will be temporally unstable

Enter Fragment Shader Interlock!

What if we had a Programmable Stencil Buffer? Like Programmable Blending?

- Fragment Shader Interlock allows for a single Critical Section in the Fragment Shader which is entered into w.r.t. Rasterization Order allowing synchronized access to any resource and can be used to implement:
  - DX12-style Raster Ordered Views
  - Programmable blending of non-HW-blendable formats like RGB9E5 (or your own)
  - Complex Per-Pixel Data Structures which need in-order atomic modifications
  - Single Pass Order Independent Transparency such as Multiple Layer Alpha Blending
    - MLAB can be implemented with a spinlock (not UB) but will be temporally unstable
- Only one Desktop IHV doesn't implement it, despite ability to do so as demonstrated in FOSS Drivers

Enter Fragment Shader Interlock!

What if we had a Programmable Stencil Buffer? Like Programmable Blending?

- Fragment Shader Interlock allows for a single Critical Section in the Fragment Shader which is entered into w.r.t. Rasterization Order allowing synchronized access to any resource and can be used to implement:
  - DX12-style Raster Ordered Views
  - Programmable blending of non-HW-blendable formats like RGB9E5 (or your own)
  - Complex Per-Pixel Data Structures which need in-order atomic modifications
  - Single Pass Order Independent Transparency such as Multiple Layer Alpha Blending
    - MLAB can be implemented with a spinlock (not UB) but will be temporally unstable

- Only one Desktop IHV doesn't implement it, despite ability to do so as demonstrated in FOSS Drivers
  - That's okay, we will just show a "Buy a Real CAD GPU" pop-up with our Amazon Affiliate Link

Enter Fragment Shader Interlock!

What if we had a Programmable Stencil Buffer? Like Programmable Blending?

- Fragment Shader Interlock allows for a single Critical Section in the Fragment Shader which is entered into w.r.t. Rasterization Order allowing synchronized access to any resource and can be used to implement:
  - DX12-style Raster Ordered Views
  - Programmable blending of non-HW-blendable formats like RGB9E5 (or your own)
  - Complex Per-Pixel Data Structures which need in-order atomic modifications
  - Single Pass Order Independent Transparency such as Multiple Layer Alpha Blending
    - MLAB can be implemented with a spinlock (not UB) but will be temporally unstable
- Only one Desktop IHV doesn't implement it, despite ability to do so as demonstrated in FOSS Drivers
  - That's okay, we will just show a "Buy a Real CAD GPU" pop-up with our Amazon Affiliate Link
- Extension seems not compatible with Dynamic Rendering? So we use Renderpasses
  - Interlock scopes specified to be a Subpass, and Dynamic Rendering doesn't have subpasses...
  - Maybe should open a Vulkan spec issue about clarifying interactions of FSIL and Dynamic Render?

Why Fragment Shader Interlock > other extensions?

Currently all other extensions only allow Shader Access to a bound attachment, and they suffer from:

- ALL = Storage PerPixel mapped to location in an image and bounded by max attachment number
- EXT_attachment_feedback_loop_layout = Only one read-write loop per Subpass Dependency or Barrier
- The only two alternatives are rasterization_order_attachment_access or shader_tile_image, both:
  - Only provide read-only access, so need to export values to all attachments, which is bad for:
    - MLAB, often only modifications are the opacities of buckets after current fragment depth
    - Containers in general, especially Heaps, Trees and Linked Lists → whole container copied
  - **TL;DR:** Either discard NOOP or copy/overwrite all coherent storage
- rasterization_order_attachment_access explicitly stated to only work with Renderpasses
  - Works by allowing feedback loops between input and subpass attachments w/o explicit sync.
- shader_tile_image explicitly requires Dynamic Rendering

Our New Algorithm!

We actually achieve the original algorithm only drawing the geometry once:

1. If current Coverage<= 0, then discard, else enter Critical Section

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();
```

Our New Algorithm!

We actually achieve the original algorithm only drawing the geometry once:

1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
```

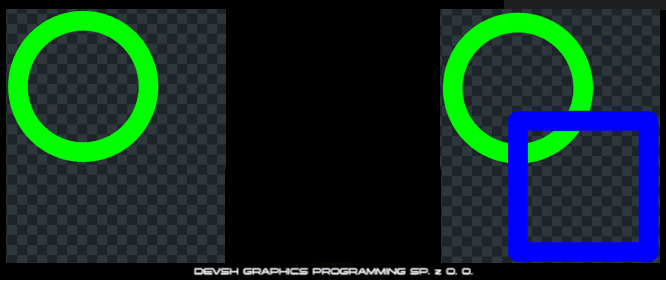
Our New Algorithm!

We actually achieve the original algorithm only drawing the geometry once:

1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S
3. If either Object ID changed or Coverage increased, overwrite entry in S

```
if (currAlpha<=THRESHOLD)
    discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
bool resolve = currObjectID!=prevObjectID;
if (resolve || currAlpha>prevAlpha)
    S[fragCoord] = (currObjectID,currAlpha);
```



Here you can see the Object-by-Object contents Pseudostencil Image S, normally this is all drawn with a single drawcall.

Our New Algorithm!

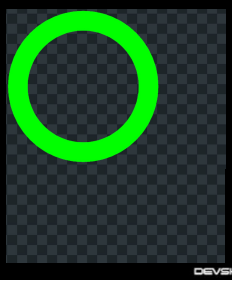
We actually achieve the original algorithm only drawing the geometry once:

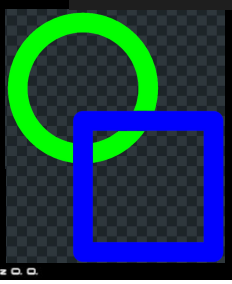
1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S
3. If either Object ID changed or Coverage increased, overwrite entry in S
4. Exit Critical Section

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
bool resolve = currObjectID!=prevObjectID;
if (resolve || currAlpha>prevAlpha)
      S[fragCoord] = (currObjectID,currAlpha);

endInvocationInterlockEXT();
```





Our New Algorithm!

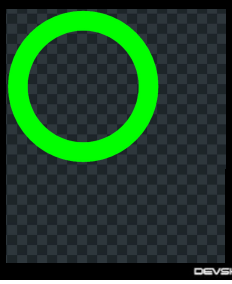
We actually achieve the original algorithm only drawing the geometry once:

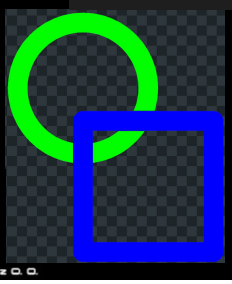
1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S
3. If either Object ID changed or Coverage increased, overwrite entry in S
4. Exit Critical Section
5. If Object ID didn't change then discard

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
bool resolve = currObjectID!=prevObjectID;
if (resolve || currAlpha>prevAlpha)
      S[fragCoord] = (currObjectID,currAlpha);

endInvocationInterlockEXT();
if (!resolve)
            discard;
```





Our New Algorithm!

We actually achieve the original algorithm only drawing the geometry once:

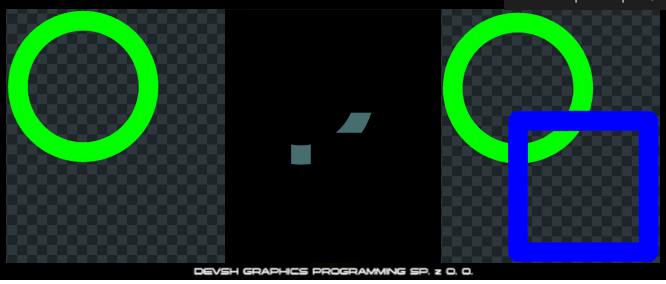
1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S
3. If either Object ID changed or Coverage increased, overwrite entry in S
4. Exit Critical Section
5. If Object ID didn't change then discard
6. Take RGBA colour from style associated with the PREVIOUS Object ID
7. Multiply the alpha by PREVIOUS coverage value in S and blend to output

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
bool resolve = currObjectID!=prevObjectID;
if (resolve || currAlpha>prevAlpha)
      S[fragCoord] = (currObjectID,currAlpha);

endInvocationInterlockEXT();
if (!resolve)
            discard;

col = styles[objects[prevObjectID].styleIdx].color;
col.a *= prevAlpha;
```



Now you can see the evolution of framebuffer contents between object draws.

Our New Algorithm!

We actually achieve the original algorithm only drawing the geometry once:

1. If current Coverage<= 0, then discard, else enter Critical Section
2. Read pixel's previous Coverage and Object ID from Image S
3. If either Object ID changed or Coverage increased, overwrite entry in S
4. Exit Critical Section
5. If Object ID didn't change then discard
6. Take RGBA colour from style associated with the PREVIOUS Object ID
7. Multiply the alpha by PREVIOUS coverage value in S and blend to output

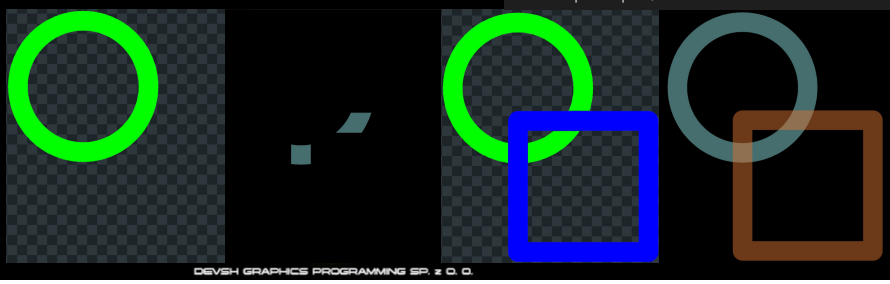
After everything is drawn, we draw a Full Screen Triangle with Alpha=0 and ObjectID=INVALID, to both resolve and clear the Image S.

```
if (currAlpha<=THRESHOLD)
      discard;
beginInvocationInterlockEXT();

(prevObjectID,prevAlpha) = S[fragCoord];
bool resolve = currObjectID!=prevObjectID;
if (resolve || currAlpha>prevAlpha)
      S[fragCoord] = (currObjectID,currAlpha);

endInvocationInterlockEXT();
if (!resolve)
            discard;

col = styles[objects[prevObjectID].styleIdx].color;
col.a *= prevAlpha;
```



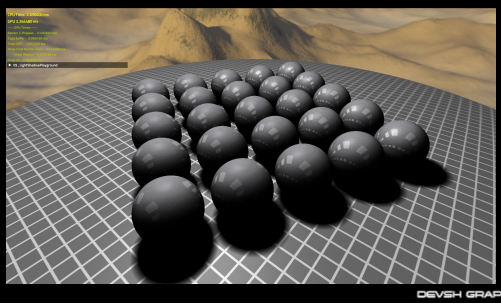
DEVSH GRAPHICS PROGRAMMING SP. z O. O.

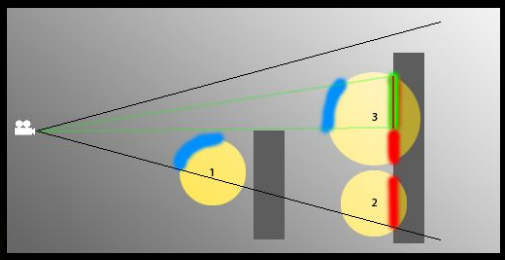
Note that you can operate on SDFs instead of coverage directly, which enables you to do CSG operations other than Union.

Remember Stencil Light Volumes?

So this was loosely inspired by a never published invention Matt contributed as an optimization to The Forge 1.23 in 2019:

- Draw SDF Shadow Volumes with Instancing using 2000s Deferred Stencil Light Volumes optimization
  - Depth Test **ON** but Depth Write **OFF**, also Stencil Test ON and Compare EQUAL against 0
  - **FAR** Face increments stencil upon Stencil Pass and Depth Fail but has an empty pixel shader
  - **NEAR** Face decrements stencil upon Stencil Pass and Depth Pass then does work in the pixel shader
- Specialized Vertex Shader that reorders the vertices of a Cuboid Frustum formed by extruding the SDF's OBB in direction of light such that:
  - triangles farthest w.r.t. the camera draw before closest
    - total abuse of Vulkan's implicit Rasterization Order Guarantees!
- Fragment Shader discards **FAR** Faces but declares Early Fragment Tests so HiZ and Early Z should be retained
- Can draw thousands of these **instanced**, meaning no Pipeline State changes or doing any image clears





Only the green pixels proceed to execute the Fragment Shader. **Image Credit: Yuriy O'Donnell**

https://github.com/ConfettiFX/The-Forge/blob/v1.23/Examples_3/Unit_Tests/src/09_LightShadowPlayground

Have similar Problems?

Mail us!
new.clients () devsh.eu

DEVSH GRAPHICS PROGRAMMING SP. z O. O.

Questions?