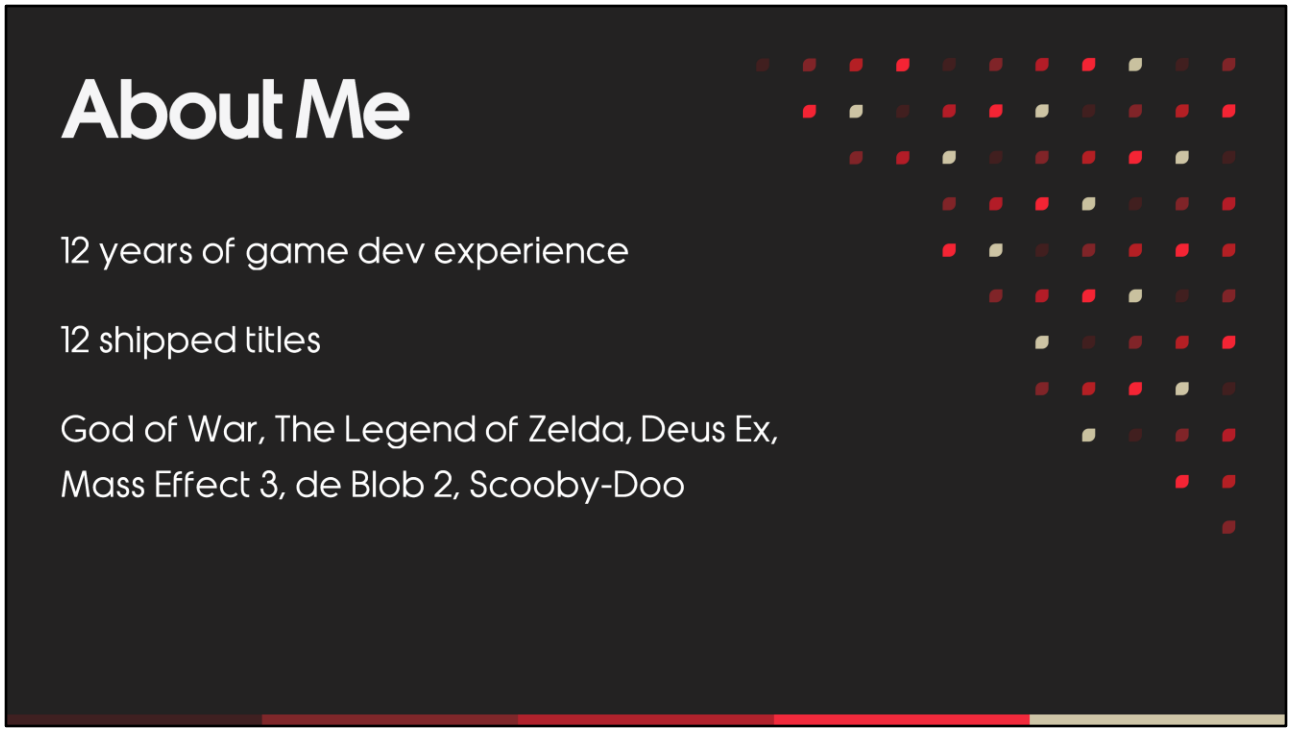

Santa
Monica
Studio™

Wind Simulation in God of War

Rupert Renard

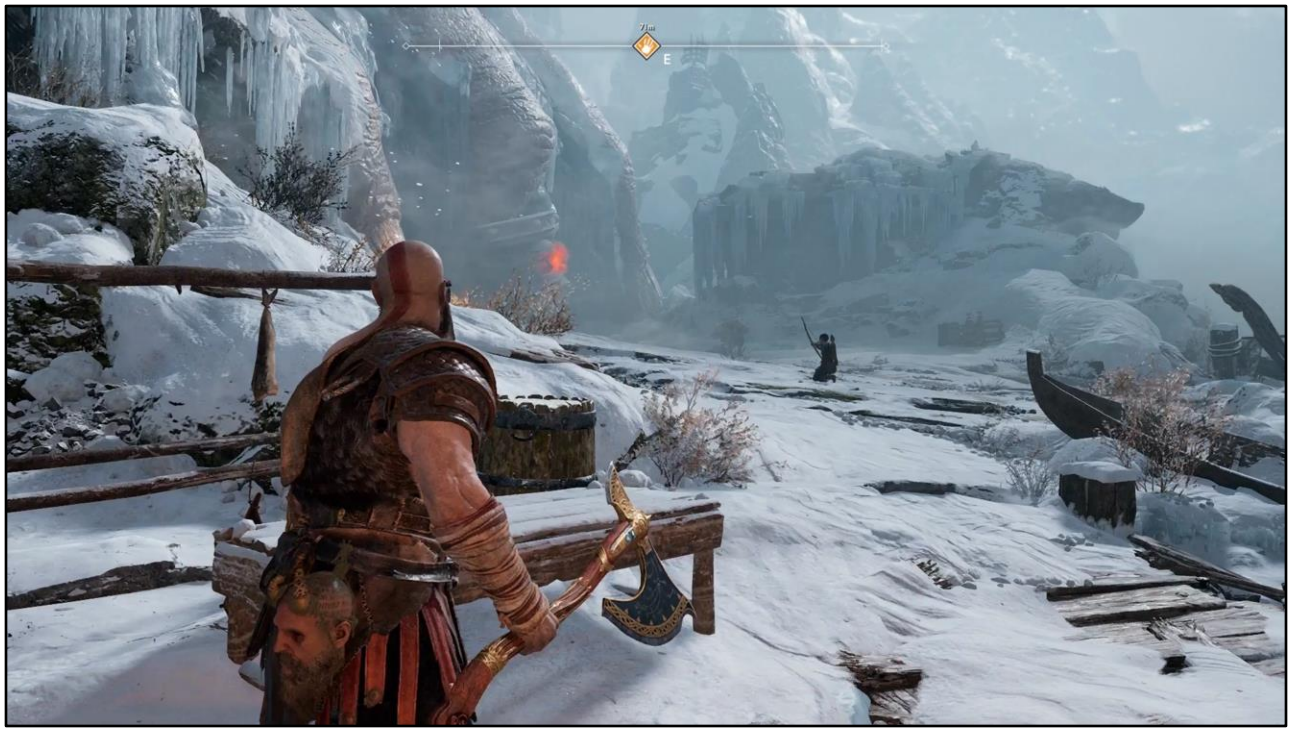


[00:00] [35 seconds]

Welcome to my talk.
My name is Rupert Renard.
I'm an Australian game developer.
I've been programming games for over 12 years now.
I've worked on 12 shipped titles, and half a dozen cancelled titles.
Some of the games I've worked on you may have heard about, such as: God of War, The Legend of Zelda, Deus Ex, Mass Effect 3, de Blob 2, and Scooby-Doo.
I've worked in a variety of programming positions.
I'm currently at Sony Santa Monica as a graphics and engine programmer, where we shipped God of War in April 2018, and it did pretty well.

[00:35] [55 seconds]

Wind is starting to become a standard feature in games these days.
Most times the wind is just a generic sine wave animating some bushes back and forth, we decided to go a little deeper.
While building our wind system, over time we started to have a reasonable decent amount of customers wanting to dynamically change their state based on the wind around their location.
Our particle system was one of the early adopters, particles getting pushed around the world.
Hair, leaves and fur were grouped together in another system, able to bend and sway in localised clusters.
Audio emitters changed the way they behaved based on the intensity of the wind.
And our cloth system also experimented with being affected by the wind.
The goal was to have a dynamic, living world, where nothing less than hard stone was susceptible from being affected by wind.



[01:30] [96 seconds]

I've recorded some footage from the final game to demonstrate some of the wind effects we have, as well as demonstrate that the game can get pretty windy at times, and it's a useful effect to have in order to help set mood.
Basically if it looks like it's moving in the wind, there's a really good chance it's being animated dynamically based on the wind simulation, otherwise it might be a baked animation.
We also wanted the wind to be SUBTLE for most cases, and to be obvious in others.

[03:06] [34 seconds]
click We prototyped the fluid simulation on the CPU, we based it off an old tech paper from GDC 2003 called Real-Time Fluid Dynamics for Games by Jos Stam.
click The tech paper however took a few shortcuts.
These shortcuts were beneficial at the time the paper was written, but created some very obvious quality issues.
We felt like these quality issues weren't acceptable anymore, especially since we have the performance available to avoid it.
So now we have the ability to do a proper fluid simulation without these shortcuts, and achieve high quality.
click But the paper itself was an amazing source of information as a starting point, and it still is today.

Wind Tiers

"Static" wind is a constant vector throughout scene. Can change as player moves through world

"Dynamic" wind is 3D volume that follows player

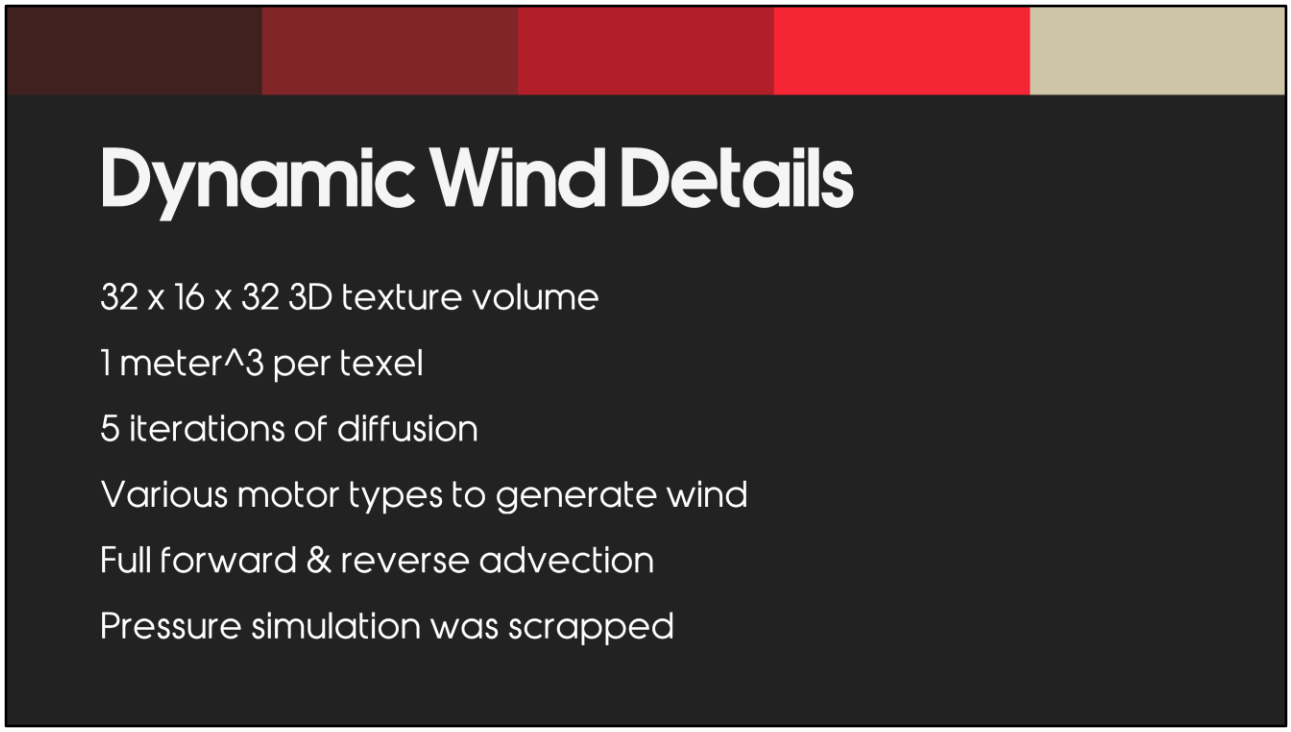
"Counter" wind is the negative velocity of an object

[03:40] [65 seconds]
We have a 3 tiered system for most of our wind sampling systems.
click Static wind is a global vector applied uniformly to everything in the scene.
It's capable of changing over time and also as the player moves around the world.
We feed static wind sometimes with a scrolling noise texture.
click Dynamic wind is the focus of this talk, it's a 3d volume that contains the detailed fluid simulation that surrounds the player.
It's always aligned to world coordinates, and shifts as the player moves.
click Counter wind is a simple mechanism used to fake wind application on things that are moving.
It's simply the negative velocity vector of the object that is moving.
If an object is moving roughly at the same speed and direction as the static or dynamic wind, this will counter-act the wind (hence the name), and give the appearance that the object isn't being affected by wind.
This is ideal, since the object is roughly in sync with the static or dynamic wind itself.

Wind Tiers

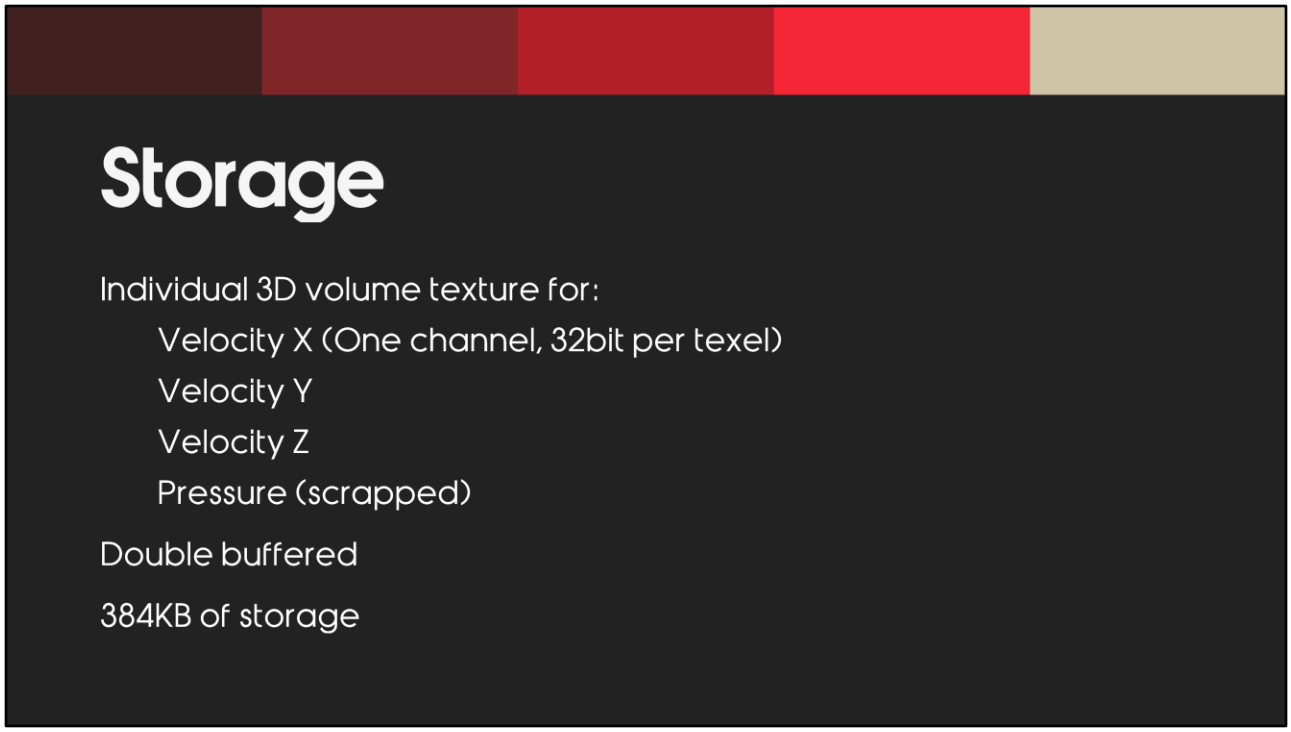
```
SampleWind(object) :=
  StaticWind +
  DynamicWind[object.position] +
  -object.velocity
```

[04:45] [16 seconds]
So we combine these three tiers like so for an object that is sampling:
The static wind vector is added.
The dynamic wind volume is sampled with the object's position.
The object's velocity is subtracted.



[05:01] [140 seconds]
click Our 3D volume is 32x16x32 texels.
click Which covers about 1 meter^3 per texel.
We had a very, very strict time budget for simulation and other wind processing on the GPU.
So our resolution is mostly tailored to fit this budget.
We opted for a uniform 3D volume out of simplicity, instead of a complicated hierarchical volume.
Our volume also needed to be large enough in world space to contain the player interactions such as throwing the axe.
Since our game takes place mostly on a horizontal game plane, we opted for more resolution horizontally than vertically.
But we still wanted enough vertical resolution to encompass trees that the player can walk past, or interact with via the axe throwing.
Try throwing the axe directly up, the leaves will be affected by it!
click We have 5 iterations of diffusion each frame.
Diffusion can be tightly packed and highly performant, so we went with 5 iterations.
There's no real reason why we picked 5 specifically, we just felt we got a good quality/performance ratio with it.
click We have several different types of motors, which are used to inject velocity in to the volume.
Some motor types were specifically crafted for certain scenarios in the game.

click We have full forward and reverse advection.
These really do compliment each other well, I highly recommend taking the time to have both, and avoid preferring one over the other.
This is one of the shortcuts I mentioned earlier.
click We also used to simulate pressure, which was eventually scrapped.
Some of the FX artists didn't like the way particles were moving, we deduced it to pressure.
We turned pressure off for the whole studio, nobody complained.
Simulating pressure was actually one of the more difficult simulations to implement, since it's a finite quantity and cannot go negative.
I was more than happy to remove it.
But for the sake of this presentation, I will include Pressure as a demonstration that you can add extra attributes fairly easily.

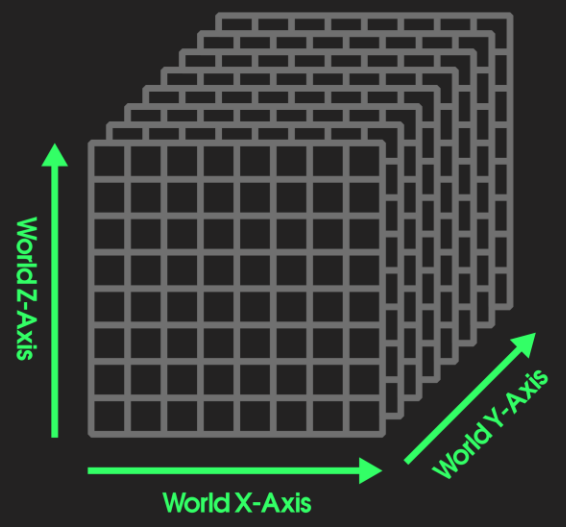


[07:21] [45 seconds]
click We have separate 3d volume textures for each attribute.
I might just point out velocity is considered a 3 dimensional attribute, so we have separate textures for each axis.
This actually proved to be incredibly beneficial for performance.
You can see we have a very slim number of attributes.
This was mostly enforced by our tight timing budget.
But we also weren't really willing to dive in to any of the more exotic fluid simulation attributes such as heat.
click In order to properly execute the simulation, we needed to double buffer each of the textures.
click Combining all the textures needed for simulation, we ended up with about 384KB worth of storage.

Storage

XZ slices along Y axis

```
float Sample(uniform Texture3D<float>
tex, in uint3 coord)
{
  return tex[coord.xzy];
}
```



[08:06] [60 seconds]
We ended up swizzling the way we access the 3D textures.
Textures have restrictions on their width and height, notably they must be a multiple of 4.
However 3D textures are able to have finer control on the amount of slices (aka depth) they have.
Originally we took the naiive approach and had world X and Y slices along the Z axis.
But we preferred having our world X and Z axis be treated uniformly, and we wanted finer control over the Y axis.
So this meant the texture restrictions lead our 3D volume to actually be world X and Z slices, along the world Y axis.
You can see this in the diagram on the right, the texture is 2D slices along world Y, of world X and Z dimensions.
The shader code needed to be adjusted in order to sample and write to the textures.
I've provided some demonstration code for this.

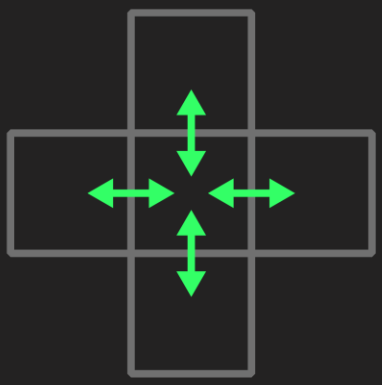
Diffusion

"Blurs" the attributes

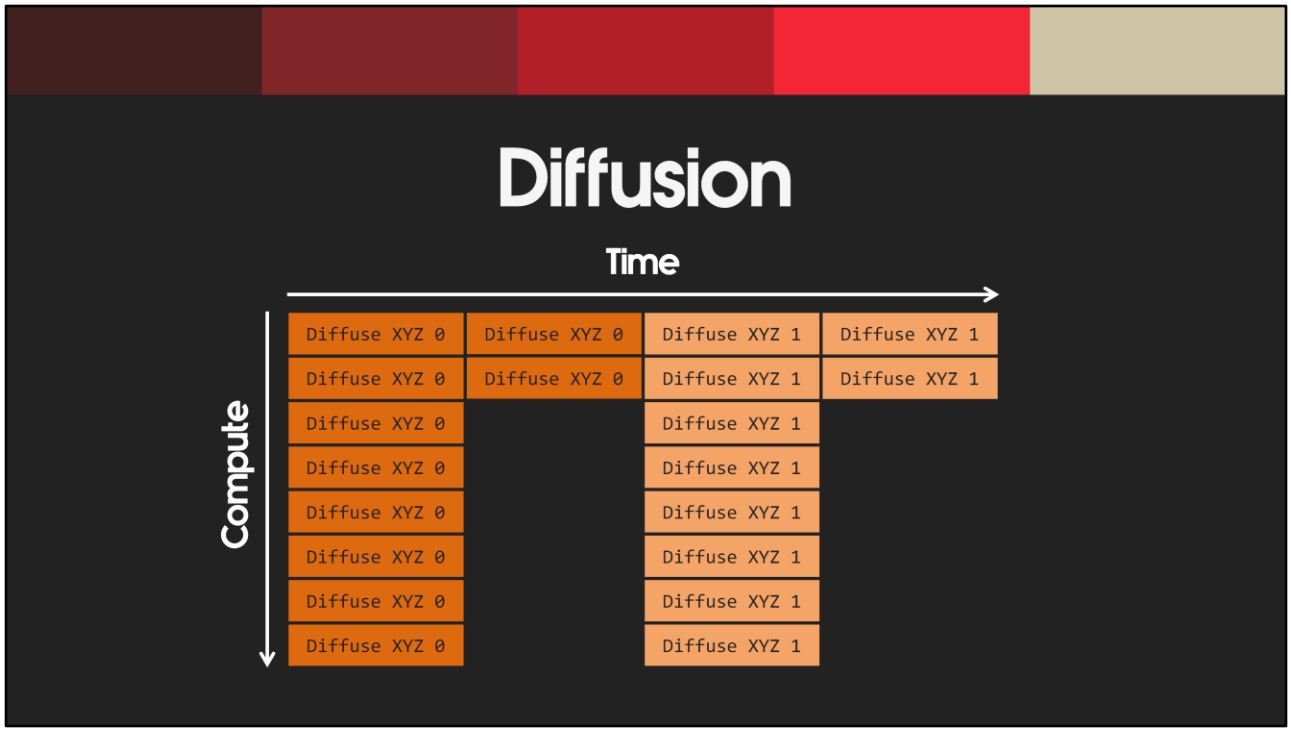
Ping pong between buffers for each step

Diffuse each axis separately

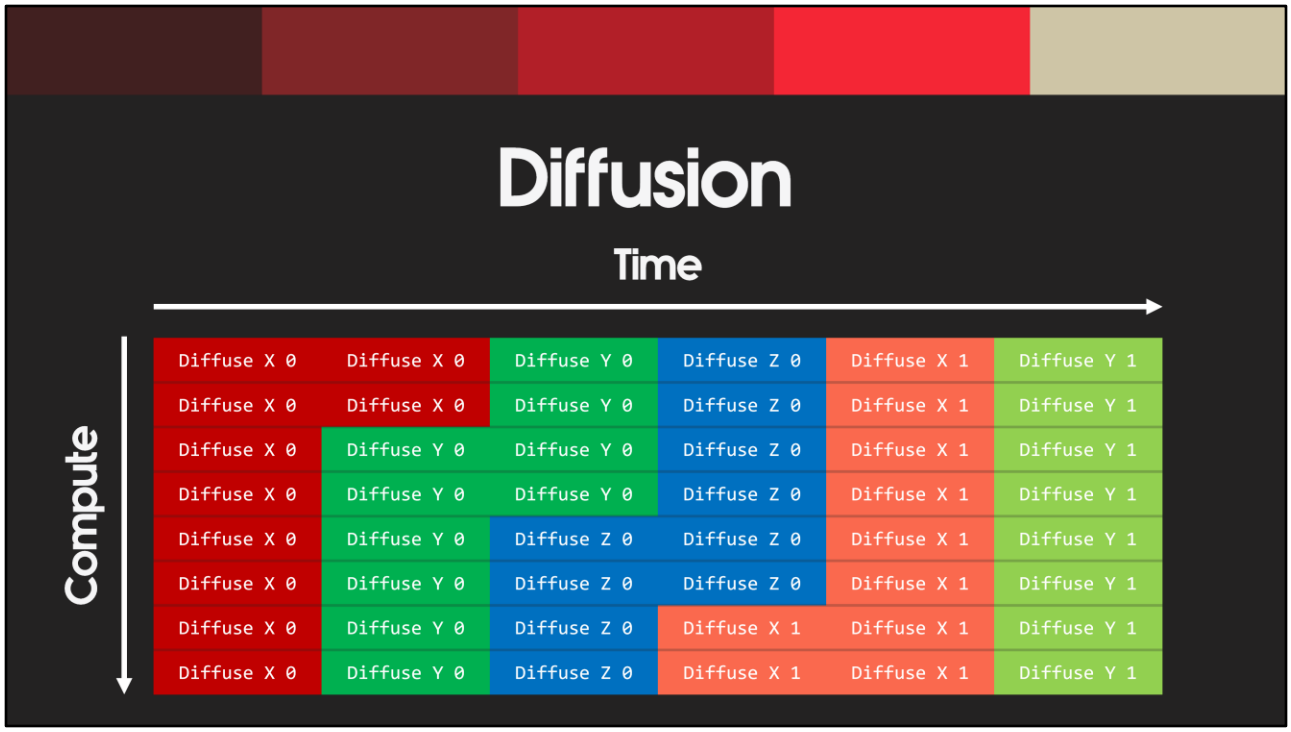
- Helps keep vector register pressure low
- Gets excellent GPU scheduling



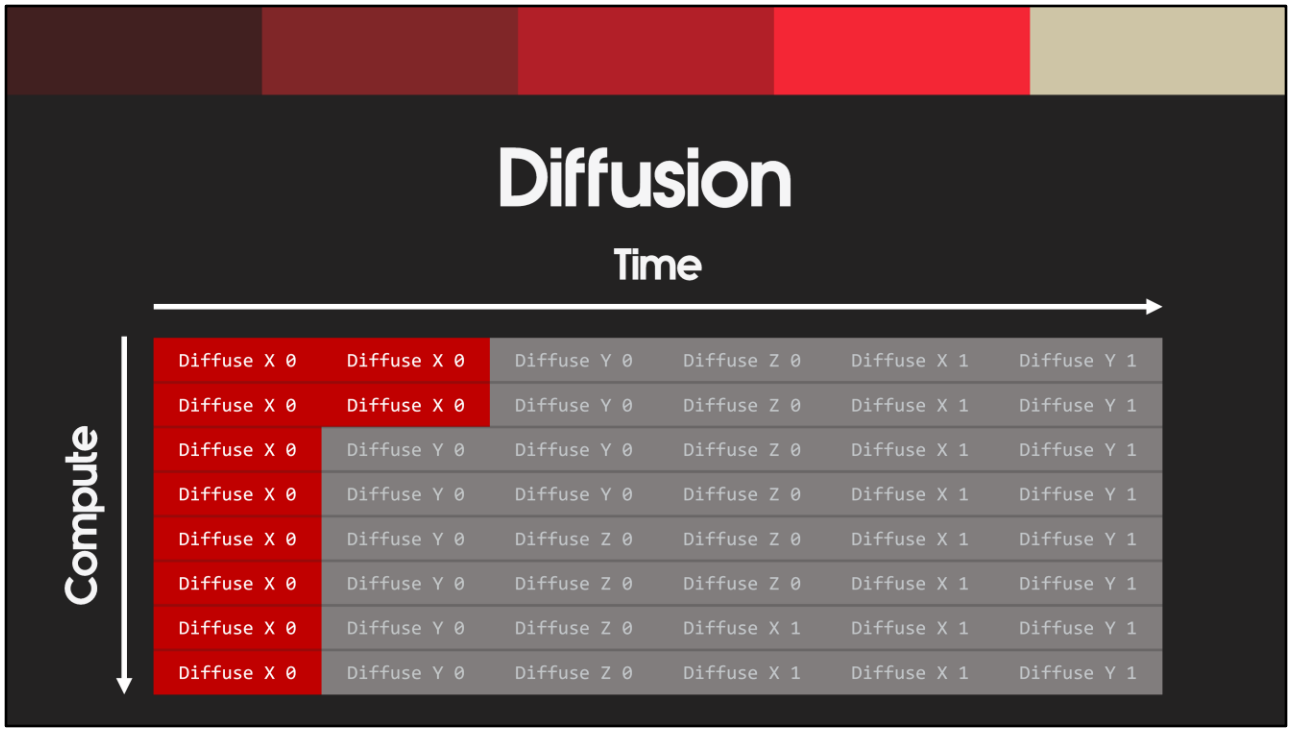
[09:06] [42 seconds]
click Diffusion is the step to spread the attributes with neighboring cells over time. Think of it as a blur.
You can also think of it as a mechanism to bring a fluid simulation to an equilibrium. It's used to transfer energy between neighboring cells, as they directly effect one another due to proximity.
click Diffusion requires double buffering, as you're diffusing one iteration at a time in to a separate buffer.
Multiple diffusion steps can simply ping pong between the double buffers.
click We found that by separating our velocity attribute in to 3 separate textures, we were able to achieve incredibly efficient performance with diffusion.



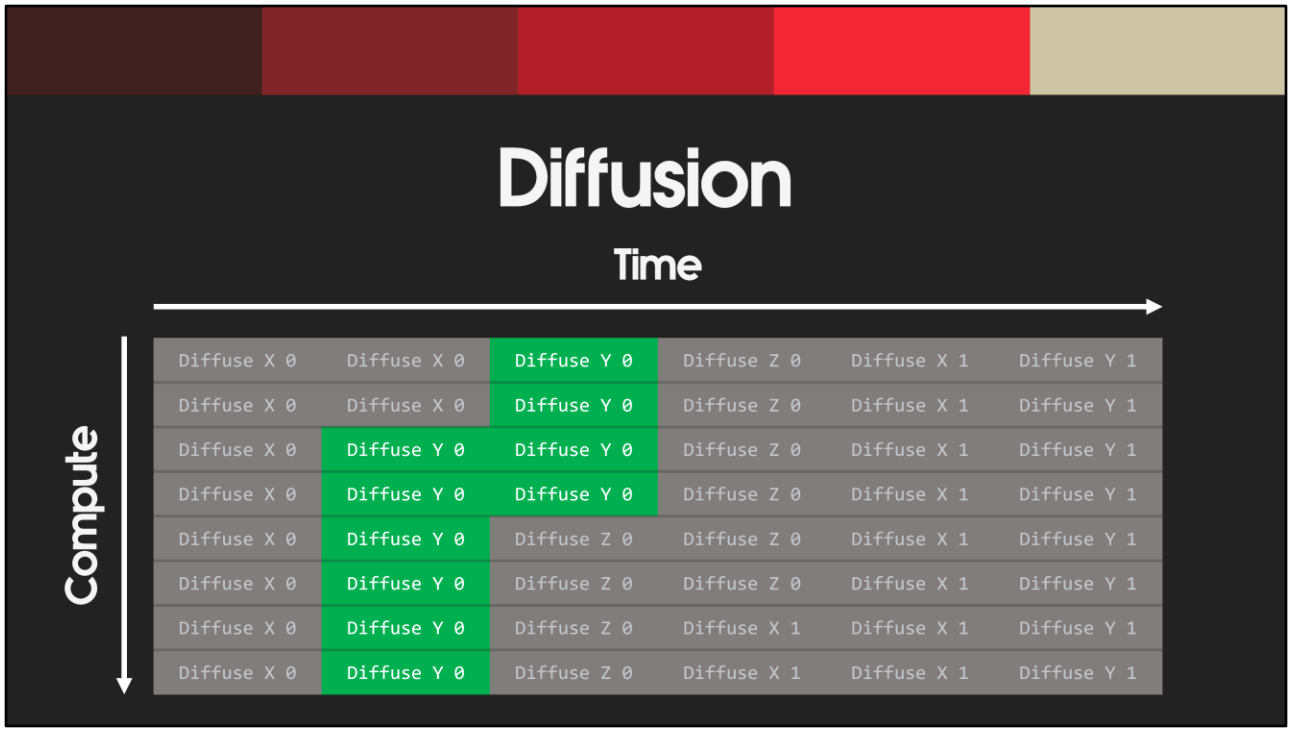
[09:48] [21 seconds]
While the overall bandwidth is the same regardless if you separate the axis or not, we can achieve faster iterations with the separation.
Without the separation, you would need to wait for the first diffusion iteration to completely finish before you can start the second.
Instead, this now only occurs per axis.



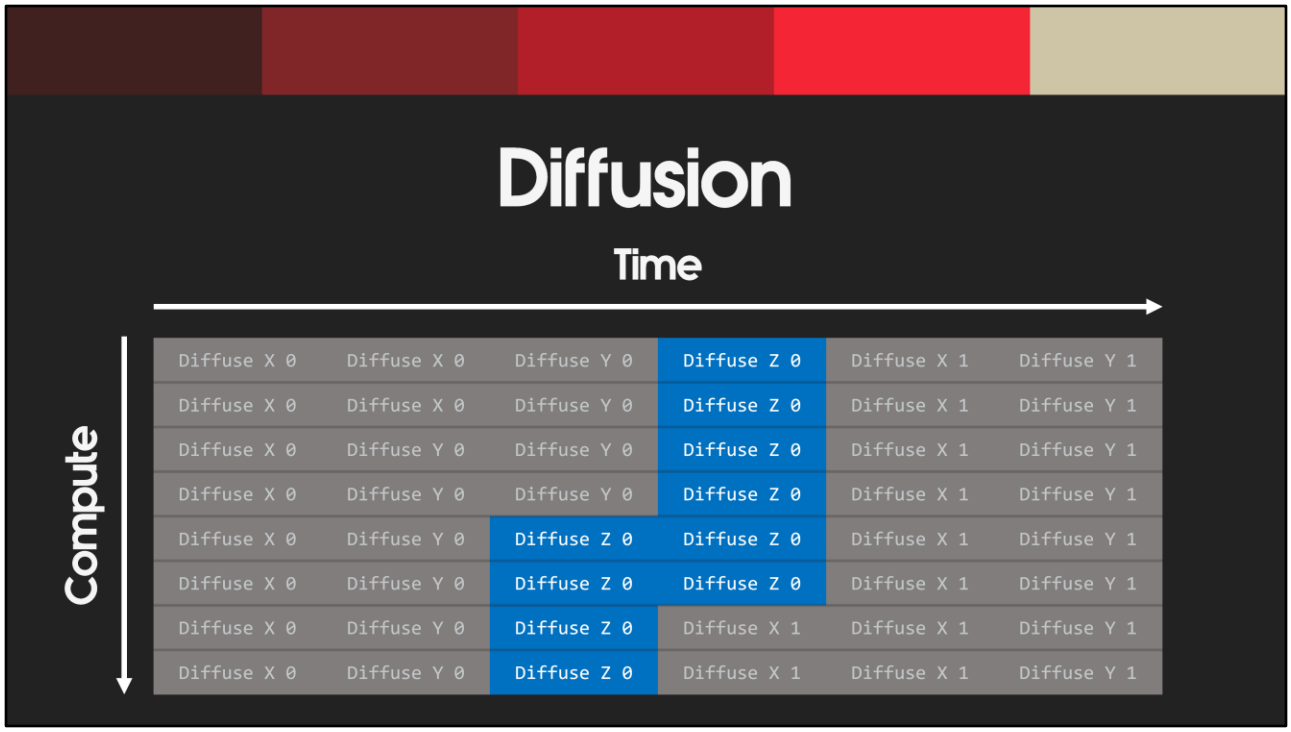
[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



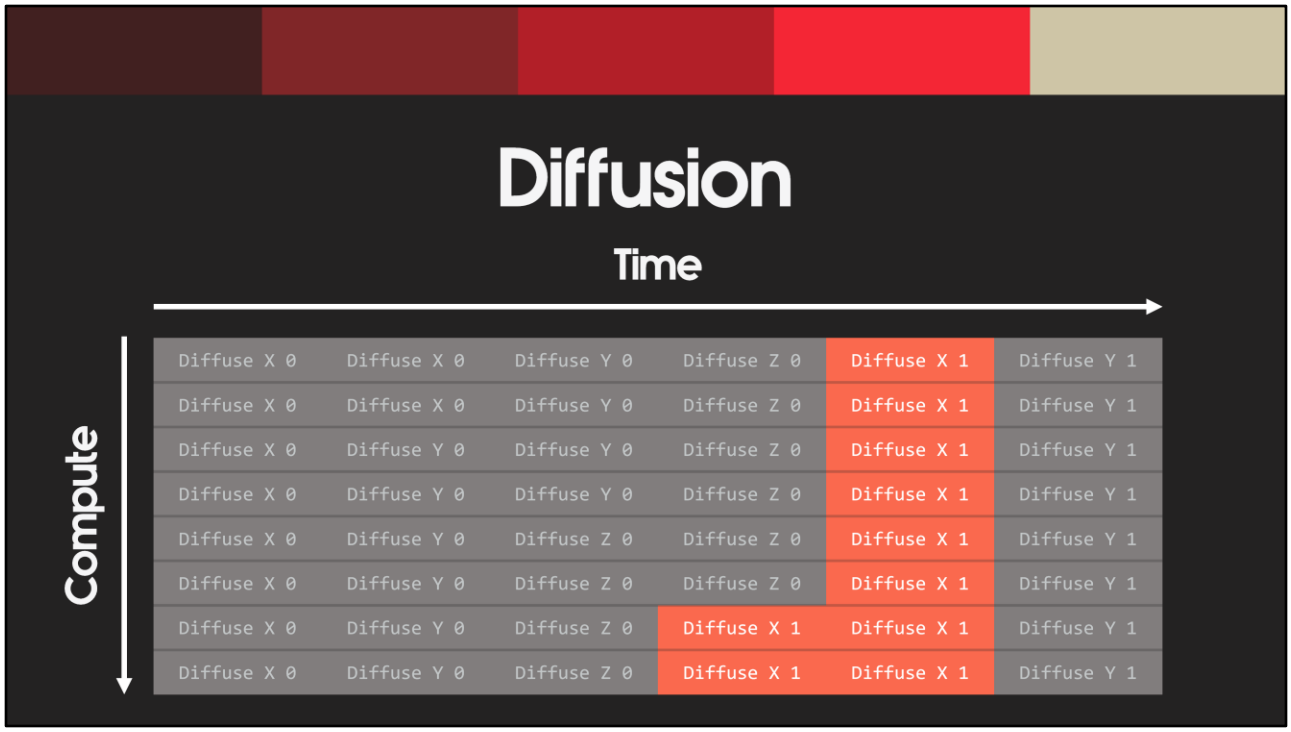
[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



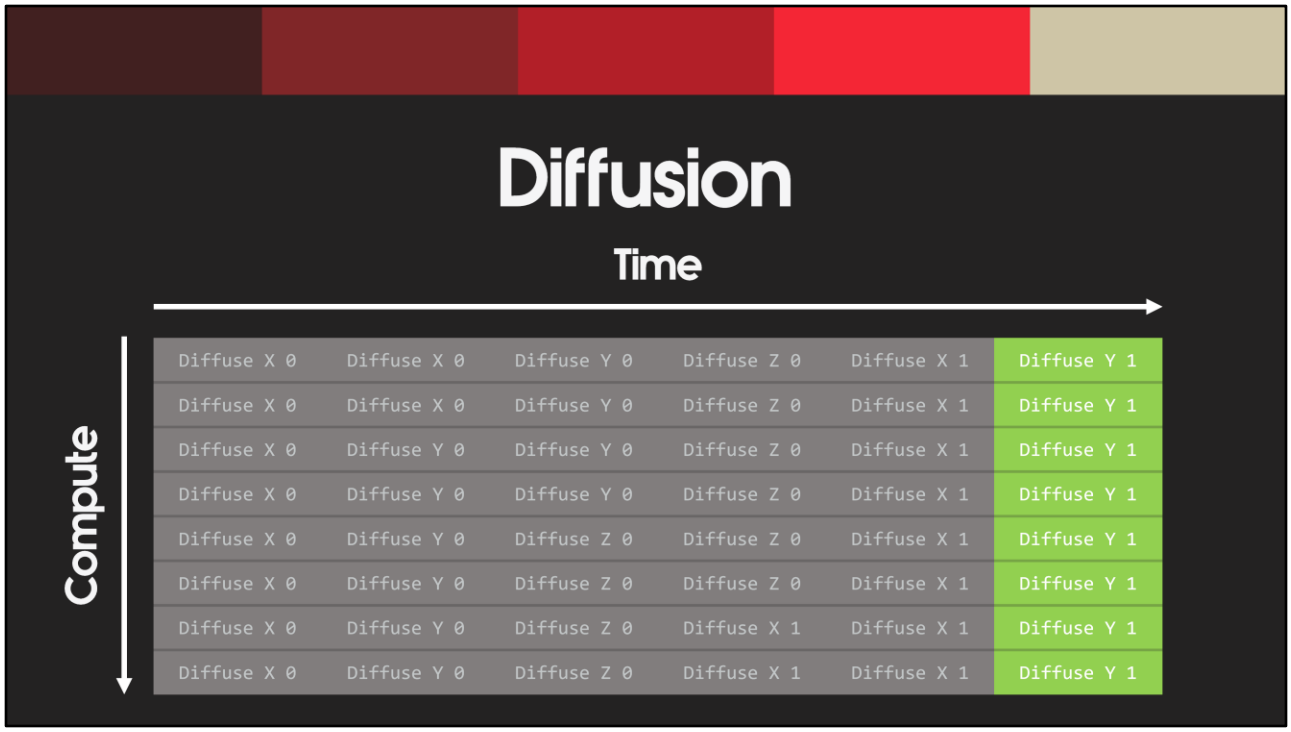
[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



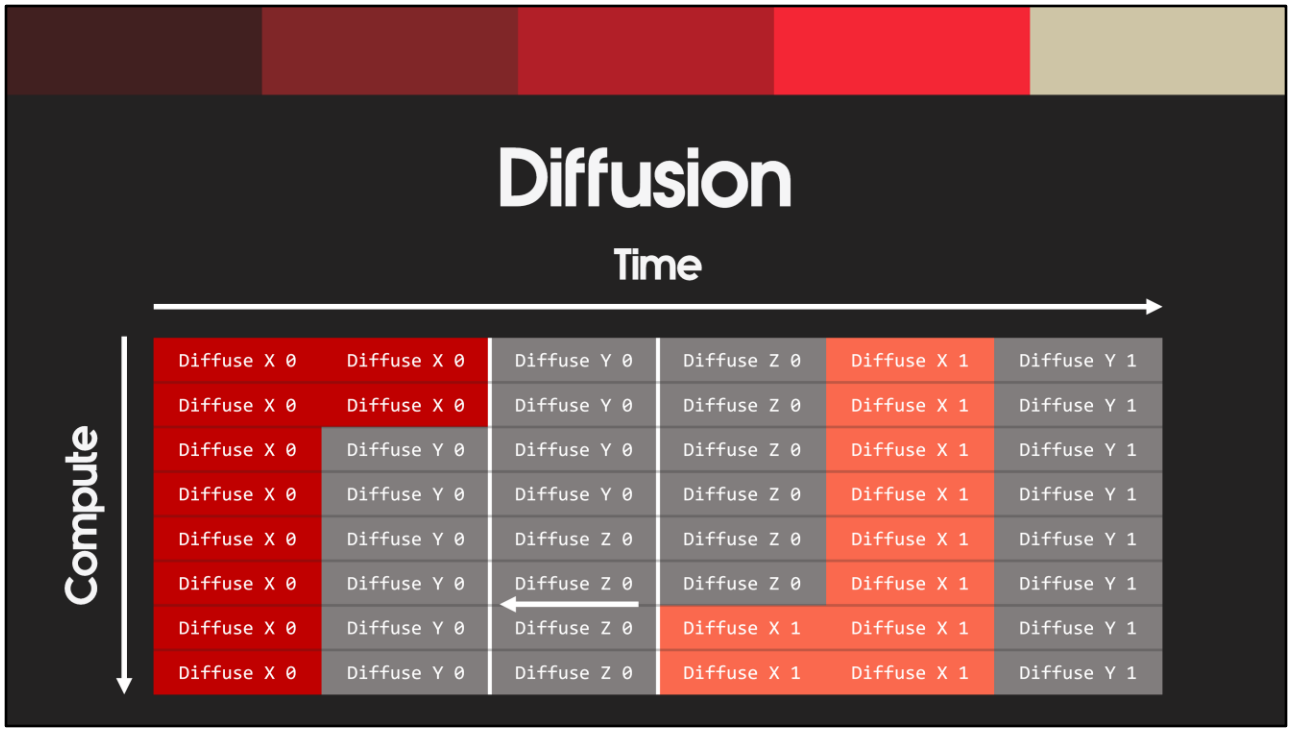
[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.



[10:09] [51 seconds]
Your shader ends up dealing with less bandwidth per thread, caused by less data per thread, which means fewer VGPR's per thread, which means better occupancy potential.
Having fewer VGPRs is also highly preferred for shaders that want to run asynchronously. *click*
You can schedule the first diffusion iteration of the x axis *click*, queued behind it is the first iteration of the y axis *click*, behind that the z axis.
Behind that *click*, the second iterations start. *click*.
The second iteration of the x axis *click* only has to wait for the first iteration of the x axis to complete.
We're able to fully utilize the GPU, and minimize any stalling between iterations.
You absorb the stalls between iterations, by doing work on the other axis.

Motors

Motors generate wind in the volume

Various motor types:

- **Directional:** Pushes wind in specified direction
- **Omni:** Inward/outward of central point
- **Vortex:** Around a specified axis
- **Moving:** Strength and direction calculated from movement of motor
- **Cylinder:** Behaves more like a sliced cone
- **Pressure:** Inject/withdraw pressure from a cell. Scrapped.

Aliasing problems, similar to rasterization

[11:00] [71 seconds]
click Motors are the main source of generating wind in our volume.
click Various motor types were supplied to the designers, some were tailor made for certain scenarios.
Most of the motors were applied as analytical shapes.
click The directional motor simply generates wind in a certain direction at a specified strength.
click Omni behaves radially from a central point, and can be outwards or inwards.
click Vortex behaves sort of like a tornado, wind is generated around an axis.
click A moving motor generates wind purely based on its movement.
The direction of wind generated is a cone shape in the direction of movement, and the strength is scaled from the movement speed.
click The cylinder was a very custom shape with multiple functions.
The radius at both ends of the cylinder were customizable.
The direction could be either unprojected, that is it's parallel to the cylinder axis.
Or the direction was projected, and would deviate from the cylinder axis based on the change of radius, sort of like a perspective projection matrix.
click A pressure motor simply injected or withdrew directly from the pressure attribute, it could create nice explosion effects, but was unfortunately dropped with the whole pressure simulation.
click We had some aliasing problems along the way, very similar to aliasing problems commonly found in rasterization.

Motors

```
void ApplyMotorDirectional(in float3 cellPosWS, uniform MotorDirectional motorDirectional, in out
float3 velocityWS)
{ // force = direction * strength * deltaTime
  float distanceSq = lengthSq(cellPosWS - motorDirectional.posWS);
  if (distanceSq < motorDirectional.radiusSq)
    velocityWS += motorDirectional.force;
}

void ApplyMotorOmni(in float3 cellPosWS, uniform MotorOmni motorOmni, in out float3 velocityWS)
{ // force = strength * deltaTime
  float3 differenceWS = cellPosWS - motorOmni.posWS;
  float distanceSq = lengthSq(differenceWS);
  if (distanceSq < motorOmni.radiusSq)
    velocityWS += motorOmni.force * rsqrt(distanceSq) * differenceWS;
}

void ApplyMotorVortex(in float3 cellPosWS, uniform MotorVortex motorVortex, in out float3 velocityWS)
{ // force = strength * deltaTime
  float3 differenceWS = cellPosWS - motorVortex.posWS;
  float distanceSq = lengthSq(differenceWS);
  if (distanceSq < motorVortex.radiusSq)
    velocityWS += motorVortex.force * cross(motorVortex.axis, rsqrt(distanceSq) * differenceWS);
}
```

[12:11] [40 seconds]
Designers were creating very small motors which the large 1 meter^3 resolution of the wind volume couldn't properly handle.
We had a few options to help this scenario, but we ended up keeping the simple aliased version for performance and continuity reasons.
One solution we proposed was to simply scale the intensity of the motor the further away it was from the texel center.
But the preferred solution, which we will likely adopt in the future, is to estimate how much of each texel is contained within the motor and scale the intensity based on that.
Here I've supplied some sample code of our wind motors.

Advection

Also separated per axis

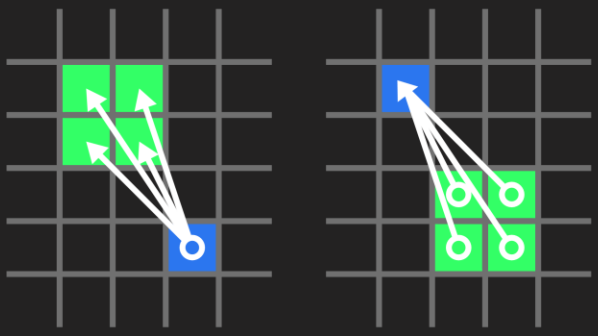
Similar performance benefits to diffusion

Forward Advection

Scatter-Write

Reverse Advection

Scatter-Read



[12:51] [70 seconds]
Advection is the process of transferring energy based on velocity, in this case between texels.
You “advect” all the attributes by the velocity attribute, including advecting velocity by itself.
Using advection, velocity pushes the energy of the attributes around your simulation, and behaves like momentum.
We can do similar performance tricks on advection like we did on diffusion,
We’re able to run the advection through separation of axis, and keep register pressure low.
Both forward and reverse are important to our simulation quality.
But both forward and reverse advection require being able to read-write to multiple texels in one iteration, and this causes contention.
Contention means multiple threads are trying to access the same resource at the same time.
Solving this on a single threaded CPU is fairly trivial, solving this on a multithreaded GPU is difficult.
In our case, we have potentially multiple threads trying to write to the same texel at the same time.
How to resolve these writes is case specific, but in our case we simply need to sum the results.

Spin Compare & Exchange

Initial solution for scattered writes

Works well

Contention wasn't too bad

Performance was better than expected

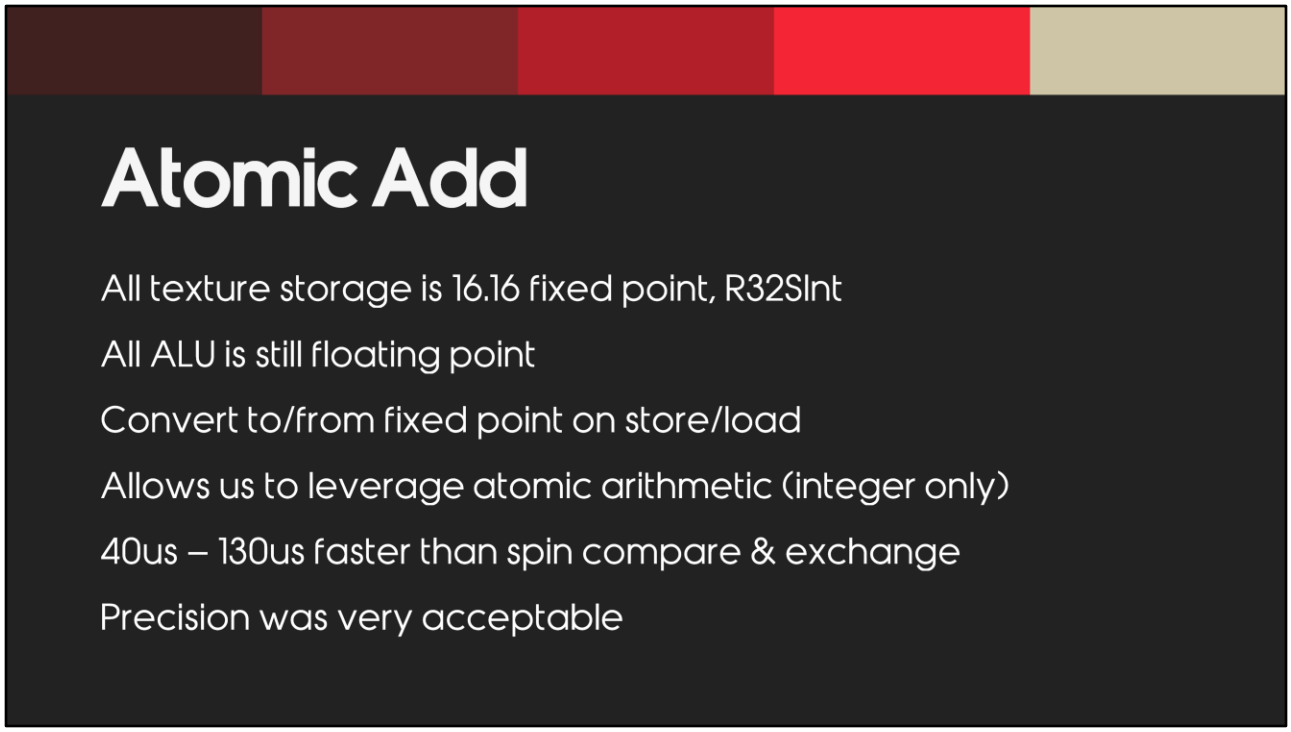
But we can do better…

[14:01] [21 seconds]
Unfortunately the hardware doesn't let us do atomic arithmetic on floating point values.
So the first solution I came up with was to spin on a compare and exchange instruction.
While not ideal, this did work, and the performance actually surprised me a bit, it wasn't too bad.
I would consider the performance to be shippable.

Spin Compare & Exchange

```
void SpinCompareExchange(uniform RWTexture3D<float> rwTex,
  uniform uint3 rwTexSize, in uint3 coord, in float value)
{
  if (all(coord < rwTexSize)) {
    float curVal = 0;
    for (;;) {
      float oldVal, newVal = curVal + value;
      InterlockedCompareExchange(rwTex[coord], curVal, newVal, oldVal);
      if (curVal == oldVal)
        break;
      curVal = oldVal;
    }
  }
}
```

[14:22] [25 seconds]
You can see in the example code, *click* the first attempt is guessing the current value of 0, *click* adding our value, and *click* attempting a compare and exchange to update the value in memory.
If the compare and exchange fails, *click* we can use the old value it gave us as our new current value, *click* add again, *click* and compare and exchange again.
But by borrowing some old-school technique, we can do better.



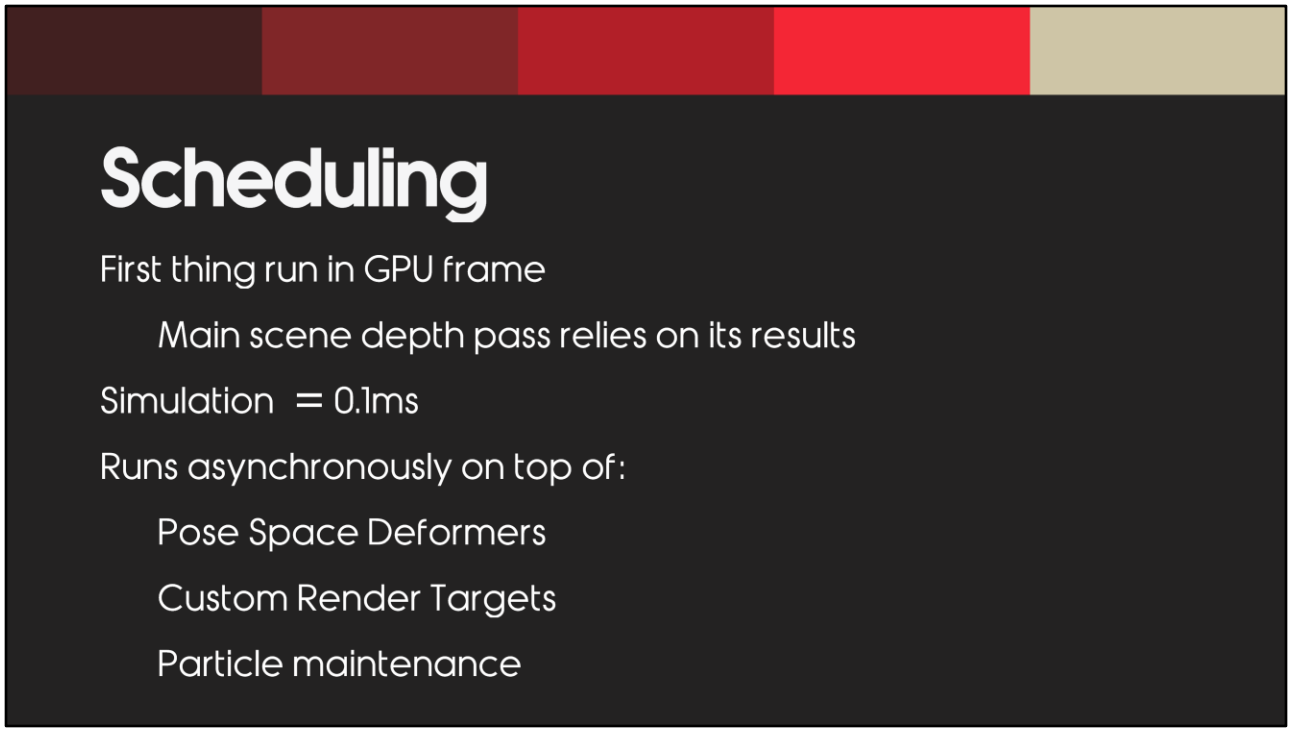
[14:47] [54 seconds]
As I said in the previous slides, the hardware doesn't allow atomic arithmetic on floating point values.
Floating point.
So why not try fixed-point? As suggested by our tech director Florian Strauss.
click We used 16.16 fixed point format, which gave us plenty of precision in both the upper and lower end.
click All of the shader ALU ops would still behave in floating point, *click* but when it came to storing and loading the values to memory we would do the conversion to and from fixed point.
click While this wouldn't decrease the contention itself, that's out of our hands here, but we were able to reduce the overhead of trying to store these scatter-writes.
click We saved approximately 40 microseconds right off the top, and noticed we got significant improvements in cases where we expect lots of contention, approximately 130 microseconds worth.
click These time savings proved well worth the precision loss, which wasn't even noticeable.

Atomic Add

```
#define FXDPT_SIZE (1 << 16)

void AtomicAdd(uniform RWTexture3D<int> rwTex,
  uniform uint3 rwTexSize, in uint3 coord, in float value)
{
  if (all(coord < rwTexSize)) {
    InterlockedAdd(rwTex[coord], (int)(value * FXDPT_SIZE));
  }
}
```

[15:41] [12 seconds]
On store, *click* we would simply scale our floating point values by the precision we needed, *click* convert to integer, *click* then use the atomic integer add available to us.
It worked great!



[15:53] [29 seconds]
click The wind simulation is quite literally the first thing that is run on our GPU frame.
click It takes about one tenth of a millisecond if it's run on the main graphics pipe.
click But it's actually run asynchronously, at the same time as Pose Space Deformers, Custom Render Targets, and some Particle maintenance shaders.
It's run first because some of the results are required for our depth pass, which is the first main pass used to draw the scene.

Step	# VGPRs	# SGPRs	Time
Diffusion	20	22	43.2us
Apply Motors	20	46	4.5us
Forward Advection Setup	8	30	2.2us
Forward Advection	20	38	16.1us
Reverse Advection Setup	8	46	3.5us
Reverse Advection	28	46	26.5us
Export	8	46	4.0us
Total			**107.0us**

[16:22] [107 seconds]
Here you can see the timings of our wind simulation.
Please note, all times were taken as if they were run on the main graphics pipe.
That is to say, all compute work that normally runs on the Async pipes are being run on the Graphics pipe.
In blue we have our diffusion steps, in red our motor application.
In orange we setup the forward advection pass, which is just clearing the memory which will accumulate the forward advection results, in yellow we do the forward advection itself.
In green we setup the reverse advection pass, but instead of clearing we duplicate the current advection results, so in cyan we perform the reverse advection.
Finally in purple, we export the results for easy access on the GPU and CPU.
You can see each pass has a very low amount of VGPR and SGPR pressure, ideal for running asynchronously.
A reminder that the diffusion pass is 5 iterations, so the total is 43.2 microseconds, averages out to around 8.64 microseconds per iteration.
You can also see the diffusion has very good scheduling, this is the separation of axis working for you.
I think if we had a larger volume than our 32 x 16 x 32 size, we could better demonstrate some of the high performance this simulation can achieve.
We simply don't have enough data to process in order to properly max out the GPU.
Hopefully the visual results of the game have proven we should consider a larger

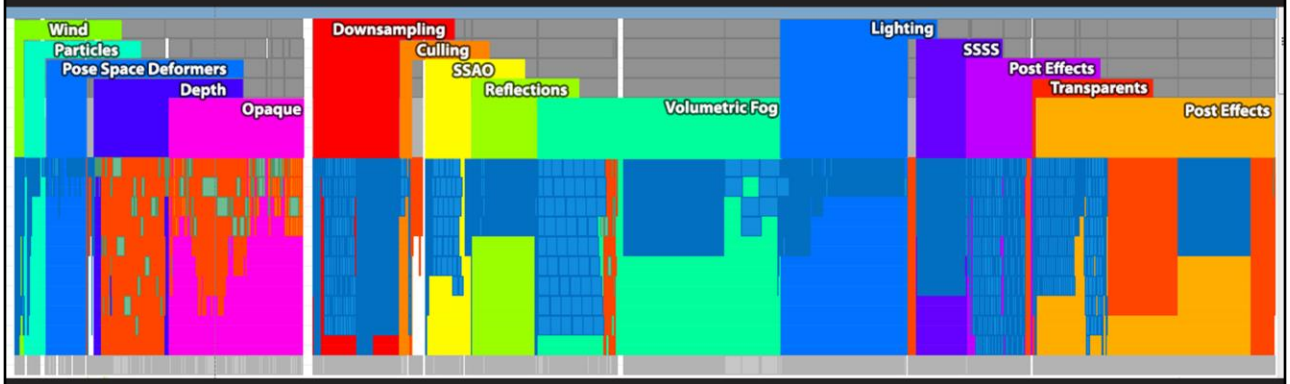
timing budget next time around.
In hindsight, I have realized I could've very easily absorbed both of the setup steps for advection by using a little bit more memory for an additional buffer.
This could've also reduced some of the synchronization between the steps, perhaps saving about 10 microseconds.

Step	Time (1x)	Time (8x)	Cost
Diffusion	43.2us	328.6us	7.6x
Apply Motors	4.5us	35.5us	7.8x
Forward Advection Setup	2.2us	14.9us	6.7x
Forward Advection	16.1us	119.5us	7.4x
Reverse Advection Setup	3.5us	31.7us	9.0x
Reverse Advection	26.5us	215.6us	8.1x
Export	4.0us	210.9us	52.7x
Total	**107.0us**	**975.5us**	**9.1x**

[18:09] [34 seconds]

Just for fun, I decided to double the resolution on each axis, giving us 8 times more resolution overall, and to time it to see how it pans out.
As you can see from the results, it scales quite nicely, except for the export stage.
The export stage is heavily bound by the Texture Addressor unit, I didn't have much time to investigate this any further.
But I suspect it's because it's effectively reading 3 lots of 32-bit textures, and writing 2 lots of 64-bit textures.
Bandwidth heavy, with very little ALU.
Async that with some ALU heavy shader work!

Timing – Full Frame



[18:43] [30 seconds]

Here I've shown the timing of our full frame.
You can see the Wind section is tiny in comparison with the rest of the frame.
Please note this is also with all Async compute turned off!
I've done this so you can see how it all fits together.
This is also why Post Effects shows up twice, we kick one lot of Post Effects to run on top of our Transparents pass.
Speaking of Transparents, this scene had very little, which is why it's a very small sliver on the timeline.

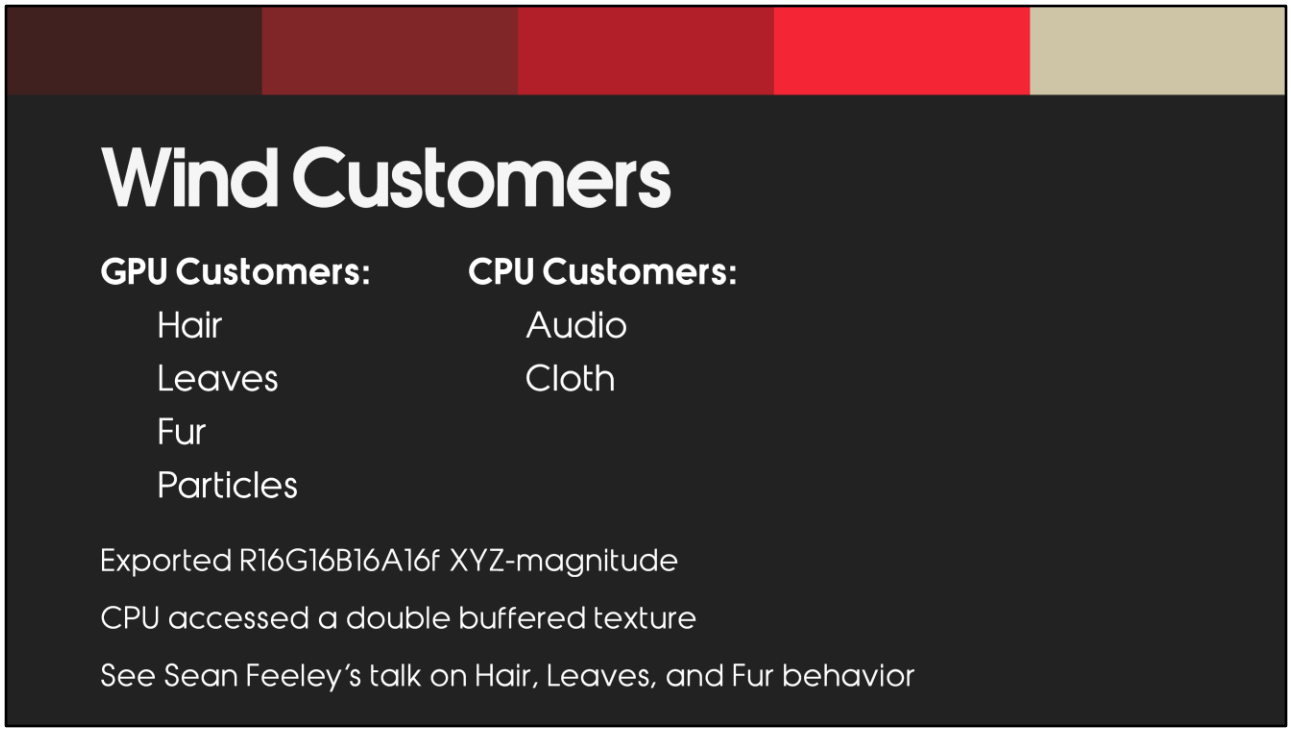
[19:13] [62 seconds]
Being able to temporarily cut out the simulation, and override it with a global uniform vector was very useful.
It lets us make sure that different systems responding to the wind were all visually relative to each other as much as possible.
Authoring assets to portray movement at X meters per second isn't easy.
This helped us find assets that weren't correctly setup, they may behave too strongly or weakly to the wind compared to the assets around them.
We could also crank up the override speed and easily find assets that hadn't been setup to be affected by wind at all yet.
I wrote a small particle emission system, which would cause particles to emit directly in front of the camera and follow the wind.
The goal was to mimic plucking some grass from the ground, and dropping it, watching how the wind affects it, like some sportsman do.
However I don't think anyone, ever, used it.
Being able to lock the volume in its current position proved useful on several occasions.

[20:15] [70 seconds]

DO NOT PLAY
Being able to visualize the 3d volume as 2d slices was by far the most important debugging feature we had available.
It lets me sanity check the simulation, as well as gives peace of mind for when artists or designers say they think something may be wrong with it.
Designers also found the visualization useful too.
play
Here you can see the green fan in the scene, that's a directional motor, it's blowing wind in the direction it's facing.
We also here have a Jotunn, whose attacks are able to generate wind.
Kratos' axe swings also generate wind, as do his axe throws.
As for the colours of the slices, mid-grey means zero wind, strong red means strong positive-x, and no red at all means strong negative-x. So red/green/blue indicate the x/y/z axis.
wait
We also have a second visualization mode, where we show the magnitude of the wind as red, the more red the larger the magnitude, and black means no wind.

[21:25] [100 seconds]

We also supplied two methods for sampling the volume around the camera and display the results on the screen.
This was mostly used by myself, and Sean Feeley from our Tech Art department, we used it to validate sampling and other obscure situations.
One method would sample the wind volume around Kratos at a specified interval, and draw the vectors in world space.
wait
We can also draw the magnitude of the wind as text instead of a vector, although that text may be a bit hard to read in this presentation.
wait
We can also change how many samples we take, as well as the spacing between them.
wait
The second method draws vectors in the exact locations the volumes' cells are in.
This can get a little overwhelming visually, you do get used to it, but it's not really meant to be used by anyone other than myself.

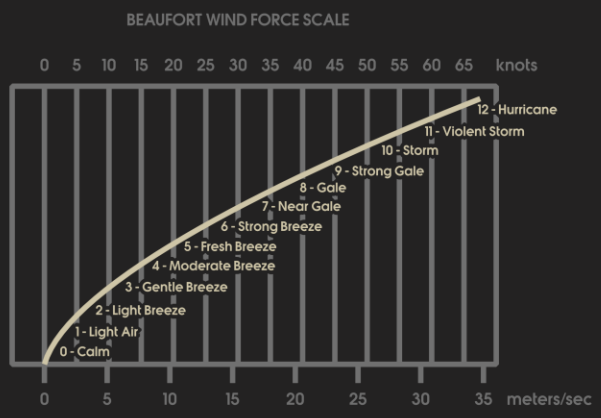


[23:05] [67 seconds]
There were various customers to the wind simulation results.
click Most read the simulation results on the GPU in the same frame.
Things such as Hair, Leaves, Fur, and Particles were all able to update their simulation immediately after the wind.
click However there were a few systems on the CPU which needed the wind simulation results too.
We don't want the wind customers to have to sample the attribute textures individually, that's 3 separate samples.
click So we ran a shader to combine the velocity attributes in to a unified 16bit per channel texture, and exported it as XYZ magnitude.
But we really had two exported textures, one for GPU in its GPU preferred tiled and swizzled form, and one as a double buffered linear texture for CPU accesses.
The Audio and Cloth simulations which ran on the CPU had to read from the previous frames wind results, but this was acceptable to us.
There is another GDC talk which goes in to more detail on how the hair, leaves, and fur behaves, I recommend you check it out.

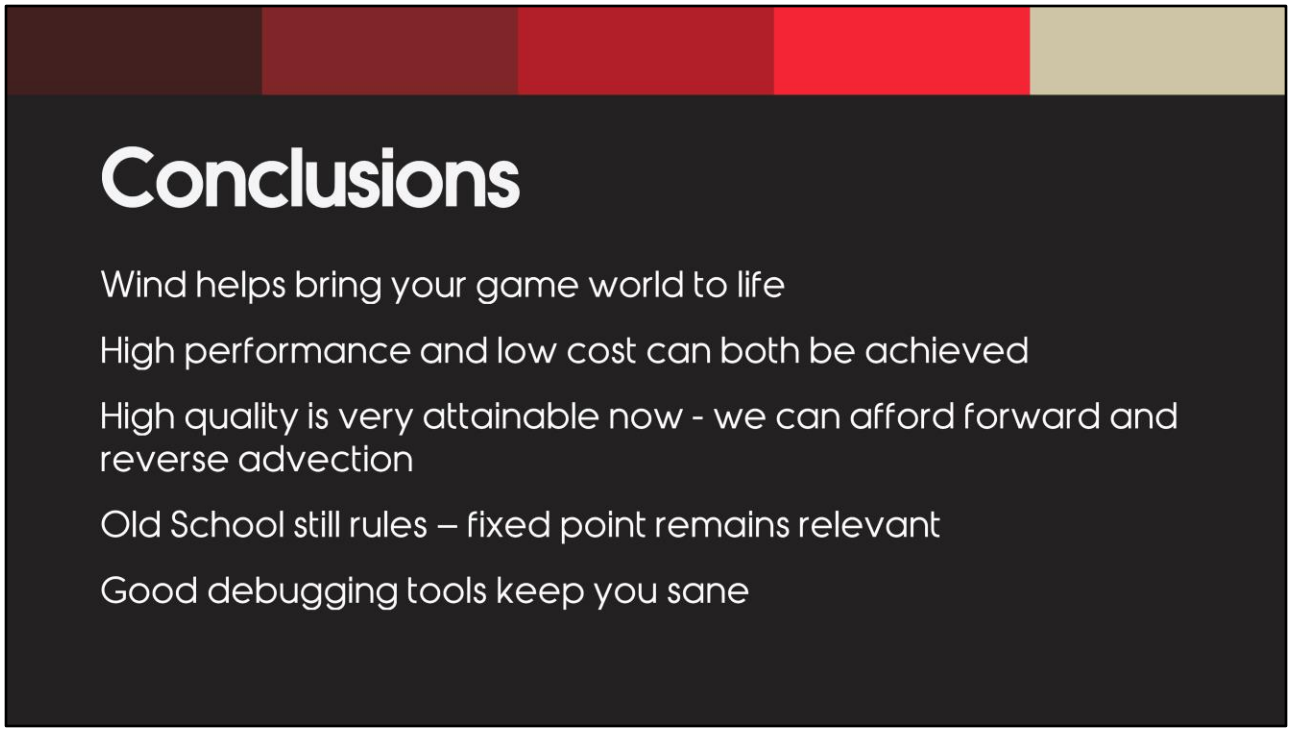
External Resources

Beaufort scale – A scale for measuring wind

https://simple.wikipedia.org/wiki/Beaufort_scale

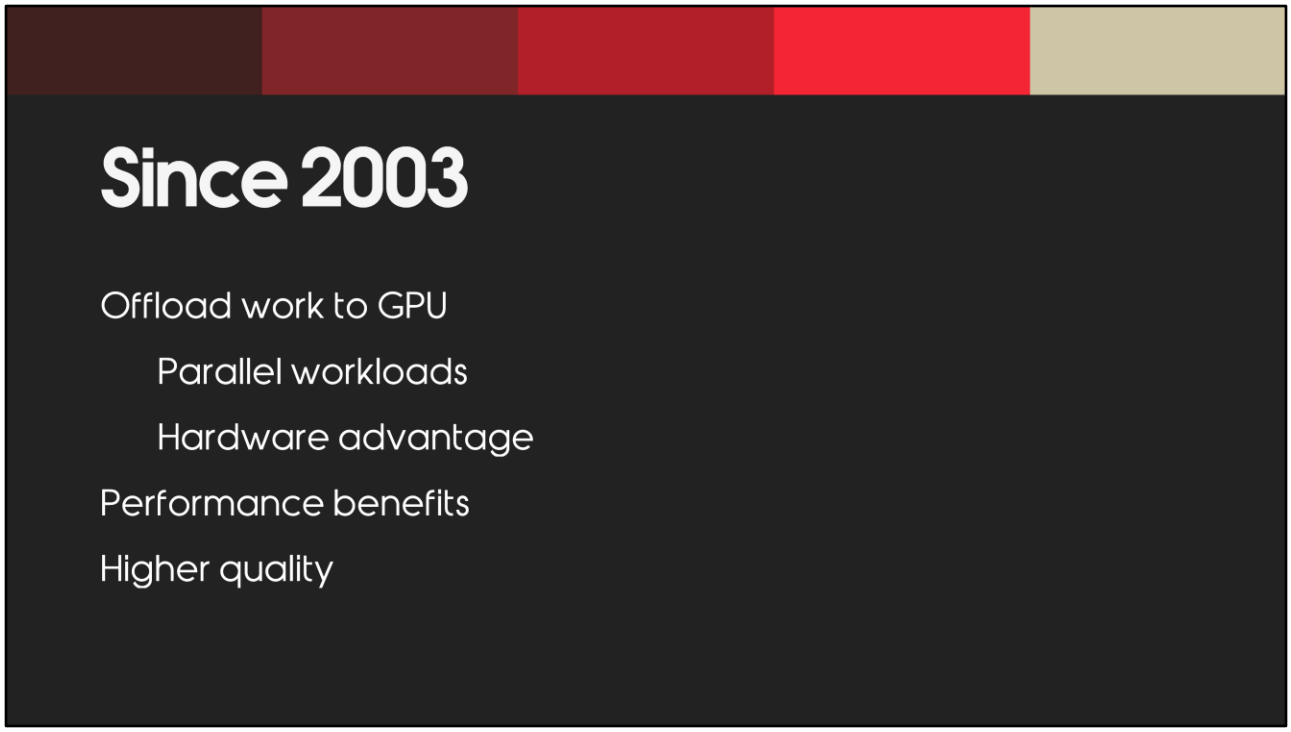


[24:12] [37 seconds]
The Beaufort scale was of great help to us.
It's generally considered a scale of zero to twelve, where zero is no wind, and twelve is hurricane force.
It gives very nice descriptions of what you would expect to see visually on land and sea at certain speeds of wind.
It would range from descriptions like "leaves rustle" for the Beaufort scale of 2, which is about 2 meters per second.
Or "small trees begin to sway" for Beaufort 5, which is around 9 meters per second.
If you're going to author assets for wind, I highly recommend you check out the Beaufort scale.



[24:49] [48 seconds]

So in conclusion, I hope I've convinced you of several things.
click Firstly, wind is a tool you can use to help bring your world alive, and add a little bit more interaction between the world and the player.
click It doesn't have to cost you much either, we've achieved a lot in a fraction of a millisecond.
click You can get pretty decent quality results, and also keep the simulation very stable.
Forward AND reverse advection compliment each other well, please use both!
click Old school techniques like fixed point will never truly ever go away, they always have a little niche to fit nicely in to.
click Finally, having good debugging tools will make the whole thing go a lot smoother, especially being able to visualize the slices.



[25:37] [15 seconds]

We've come a long way in 15 years.
What was once done on the CPU, can now be easily done on the GPU.
We have distinct advantages available to us on the GPU, it's another tool to have on your belt, so use it!

[25:52] [58 seconds]

Please go see Sean Feeley's talk on Interactive Wind and Vegetation.
His talk starts where mine leaves off.
If you liked my talk and I've managed to convince you to have wind in your own game, or maybe you're still on the fence and still need convincing, you MUST attend this talk too.
Sean's talk will focus a lot on the customers of wind which I've only very lightly covered.
Things such as foliage, hair, fur, trees, leaves, and how they interact with the wind.
He'll cover things we worked on such as leveraging compute workloads to update the state of dynamically animated objects, and logarithmically binned flow maps.
And go into detail on what goes on in the vertex shader for wind objects, covering topics such dual height field grass interaction.
He will also speak on some miscellaneous topics such as card clustering, and big o log n flood fill, and a bunch of other topics.

[26:50] [24 seconds]

Thanks everyone for attending, I'd just like to take a few moments to thank others who helped out in various ways.
Big thank you to Sean Feeley, he did more work than you can imagine in making wind in God of War a success.
Florian Strauss, Dale Son, and the rest of the Rendering Team.
Jack Mulholland, and Andreas Fredriksson.


Santa Monica Studio
Our journey
Your story
We're hiring for what's next!
10|10
DUALSHOCKERS
10|10
DARKSTATION
10|10
GAMEBLOG
"A TRIUMPHANT REINVENTION"
WAYPOINT
"THE BEST GOD OF WAR GAME"
VG247
"A MIGHTY SUCCESS"
WASHINGTON POST
95
We're expanding our family across disciplines and would love to meet you. Please visit sms.playstation.com/careers for all openings or drop us a line at sms@sony.com
@ santamonicastudio
@ SonySantaMonica
@ santamonicastudio

JOIN US AT GDC 2019

GOD OF WAR

BUILD YOUR GOD OF WAR GDC AT:
SCHEDULE.GDCONF.COM

BRUNO VELAZQUEZ • Animation Director
God of War: Breathing New Life into a Hardened Spartan • ANIMATION BOOTCAMP
MONDAY, MARCH 18 • 10:00AM - 11:00AM • ROOM 2010, WEST HALL

MELISSA SHIM • Senior Animator
Animation Bootcamp: Animation Tricks of The Trade • ANIMATION BOOTCAMP
MONDAY, MARCH 18 • 4:40PM - 5:10PM • ROOM 2010, WEST HALL

ROB DAVIS • Lead Level Designer
Level Design Workshop: The Level Design of God of War • LD SUMMIT
TUESDAY, MARCH 19 • 11:20AM - 12:20PM • ROOM 301, SOUTH HALL

ERICA PINTO • Lead Narrative Animator
What They Don't Teach You in Art School: Lessons for First Time Leads • ART DIRECTION BOOTCAMP
TUESDAY, MARCH 19 • 1:20PM - 2:20PM • ROOM 2001, WEST HALL

AXEL STANLEY-GROSSMAN • Lead Technical Character Artist
Santa Monica Studio Presents: God of War (Presented by Autodesk) • VISUAL ARTS
TUESDAY, MARCH 19 • 2:40PM - 3:40PM • ROOM 3020, WEST HALL

RUPERT RENARD • Senior Programmer
Wind Simulation in God of War • PROGRAMMING
WEDNESDAY, MARCH 20 • 10:30AM - 11:00AM • ROOM 303, SOUTH HALL

RUPERT RENARD • Senior Programmer
Disintegrating Meshes with particles in God of War • PROGRAMMING
WEDNESDAY, MARCH 20 • 11:30AM - 12:00PM • ROOM 303, SOUTH HALL

ED BEARIEN & JEET SHROFF • Assistant Producer & Gameplay Director
Playtesting God of War • PRODUCTION
WEDNESDAY, MARCH 20 • 2:00PM - 3:00PM • ROOM 2001, WEST HALL

KORAY HAGEN • Senior Programmer
The Future of Scene Description on God of War • PROGRAMMING
WEDNESDAY, MARCH 20 • 2:00PM - 3:00PM • ROOM 302, SOUTH HALL

DORI ARAZI • Director of Photography
Creating a Deeper Emotional Connection: The cinematography of God of War • VISUAL ARTS
WEDNESDAY, MARCH 20 • 3:30PM - 4:30PM • ROOM 2005, WEST HALL

JASON MCDONALD • Design Director
Taking an Axe to God of War Gameplay • DESIGN
THURSDAY, MARCH 21 • 10:00AM - 11:00AM • ROOM 303, SOUTH HALL

ERICA PINTO • Lead Narrative Animator
Keyframes and Cardboard Props: The Cinematic Process Behind God of War • VISUAL ARTS
THURSDAY, MARCH 21 • 11:30AM - 12:30PM • ROOM 2001, WEST HALL

MIKE NIEDERQUELL • Lead Sound Designer
The Sound Design of God of War • AUDIO
THURSDAY, MARCH 21 • 2:00PM - 2:30PM • ROOM 3002, WEST HALL

SHAYNA MOON • Associate Producer
Shipping Greatness: Practical Lessons from Audio Production on God of War • PRODUCTION
THURSDAY, MARCH 21 • 3:00PM - 3:30PM • ROOM 3002, WEST HALL

HAYATO YOSHIBOME • Sr. Staff Technical Combat Designer
Raising Atreus for Battle in God of War • DESIGN
THURSDAY, MARCH 21 • 4:00PM - 5:00PM • ROOM 2005, WEST HALL

JOSH HOBSON • Lead Rendering Programmer
The Indirect Lighting Pipeline of God of War • PROGRAMMING
THURSDAY, MARCH 21 • 4:00PM - 5:00PM • ROOM 2010, WEST HALL

SEAN FEELEY • Sr Staff Technical Artist
Interactive Wind and Vegetation in God of War • DESIGN
THURSDAY, MARCH 21 • 5:30PM - 6:30PM • ROOM 2005, WEST HALL

CORY BARLOG • Creative Director
Reinventing God of War
FRIDAY, MARCH 22 • 10:00AM - 11:00AM • ROOM 303, SOUTH HALL

MIHIR SHETH & JEET SHROFF • Lead Combat Designer & Gameplay Director
Evolving Combat in God of War for a New Perspective • DESIGN
FRIDAY, MARCH 22 • 1:30PM - 2:30PM • ROOM 2005, WEST HALL

Santa Monica Studio GDC