
inFAMOUS
SECOND SON
ENGINE POSTMORTEM
By Adrian Bentley
Lead Engine Programmer
SUCKER PUNCH

What is inFAMOUS Second Son? One major pillar is an open world action game where you can climb all over everything. This means making dense, streaming environments.

In moving to the PS4, we wanted to push our visuals. To accomplish this we switched to a physically based HDR shading model.

Another major pillar of the game is super powers. And with super powers you have to have cool effects. We invested heavily in our particle systems, including running them on the GPU.

Another pillar is strong narrative. To convey the emotion and subtlety we wanted without destroying our budget we built a new performance capture pipeline.

OVERVIEW

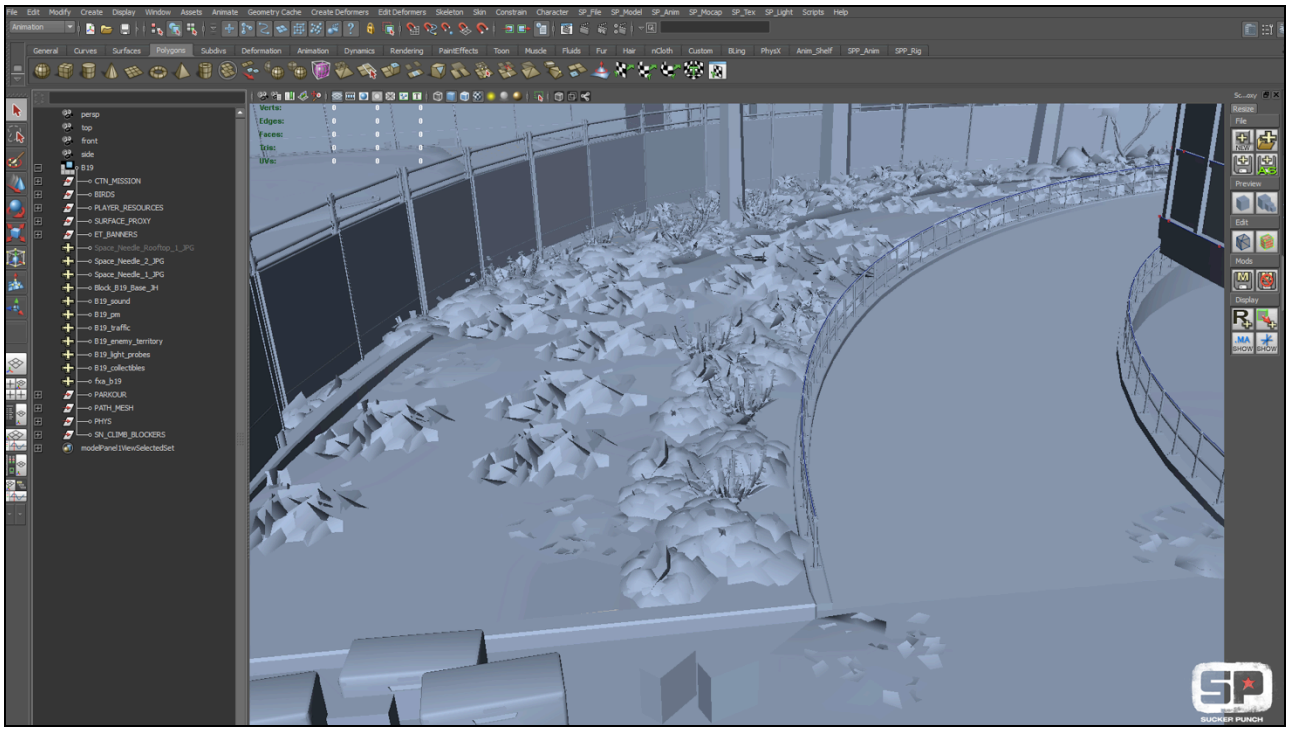
World Building
In-Game Editing
What to do with 8 GB?
CPU Threading
Graphics Overview
Compute Usage

WORLD BUILDING IN MAYA

"Proxy" Maya plugin
- Instances a model and all its configuration
  - Like Maya References but way faster to draw
- Proxies can contain proxies (build recursively)
- Intuitive and powerful, shipped 4 titles

World split into square blocks (streaming unit)
- Block proxies buildings, building proxy windows, etc.

We come from a small team mentality. For level building, we just use maya. We need it anyway for mesh and animation. We implemented a plugin to instance models and render larger scenes faster. This produces large draw call and object counts (not a problem).



Shot of maya with lots of proxy instances.

PROXY ISSUES

Simple, but can be hard to work with

- Single OpenGL calls (speed), simplistic shading if any
- Can't snap verts, uvs, etc.
- Selection is buggy (thanks Maya)
- Getting to proxy 5 layers down a pain
  - Open in new maya, repeat 4x (or open by name)

A number of authoring issues.

IMPROVED PROXY

Import/Export/Save to multiple files
- Vertex snapping, full Maya shading

More tweakability
- Transform/remove nodes a few layers down

Some Maya gotchas, but worth it
- Cleaning up non-dag nodes
- Avoid or break cross file connections

More complex code than drawing only plugin. Still it's been worth it.

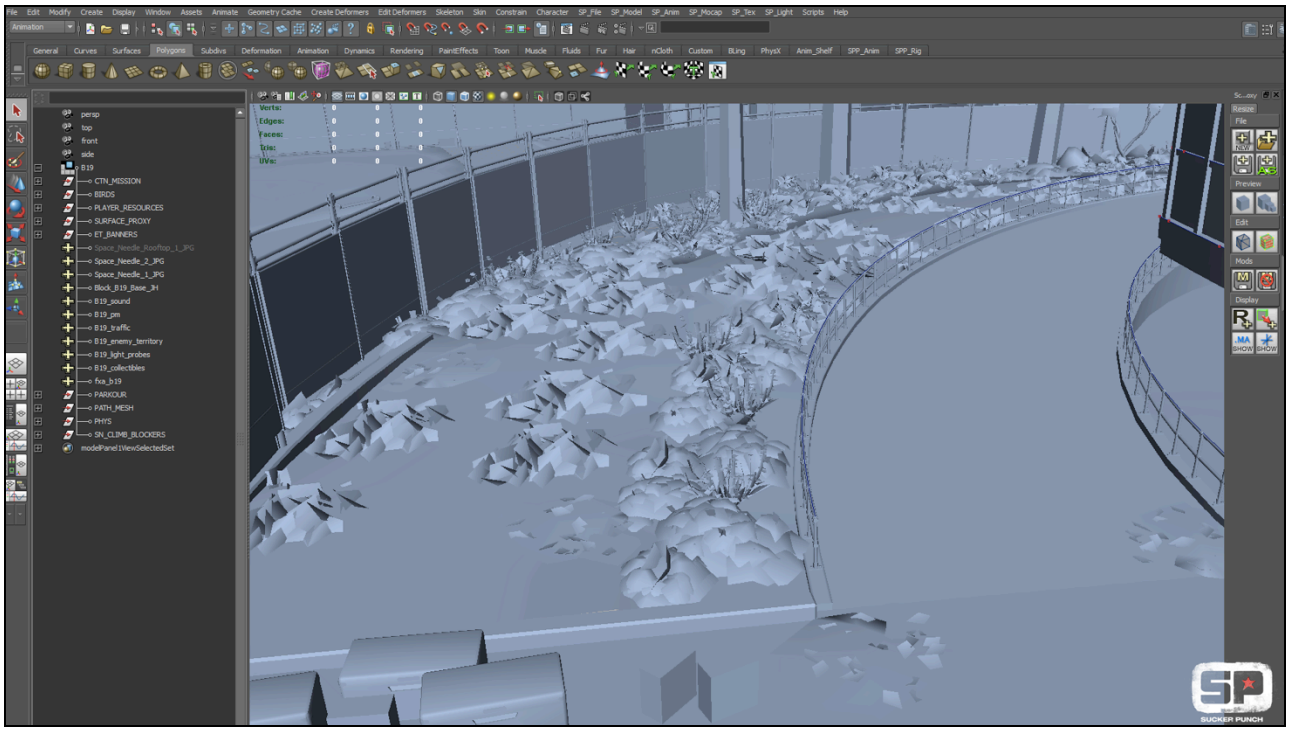
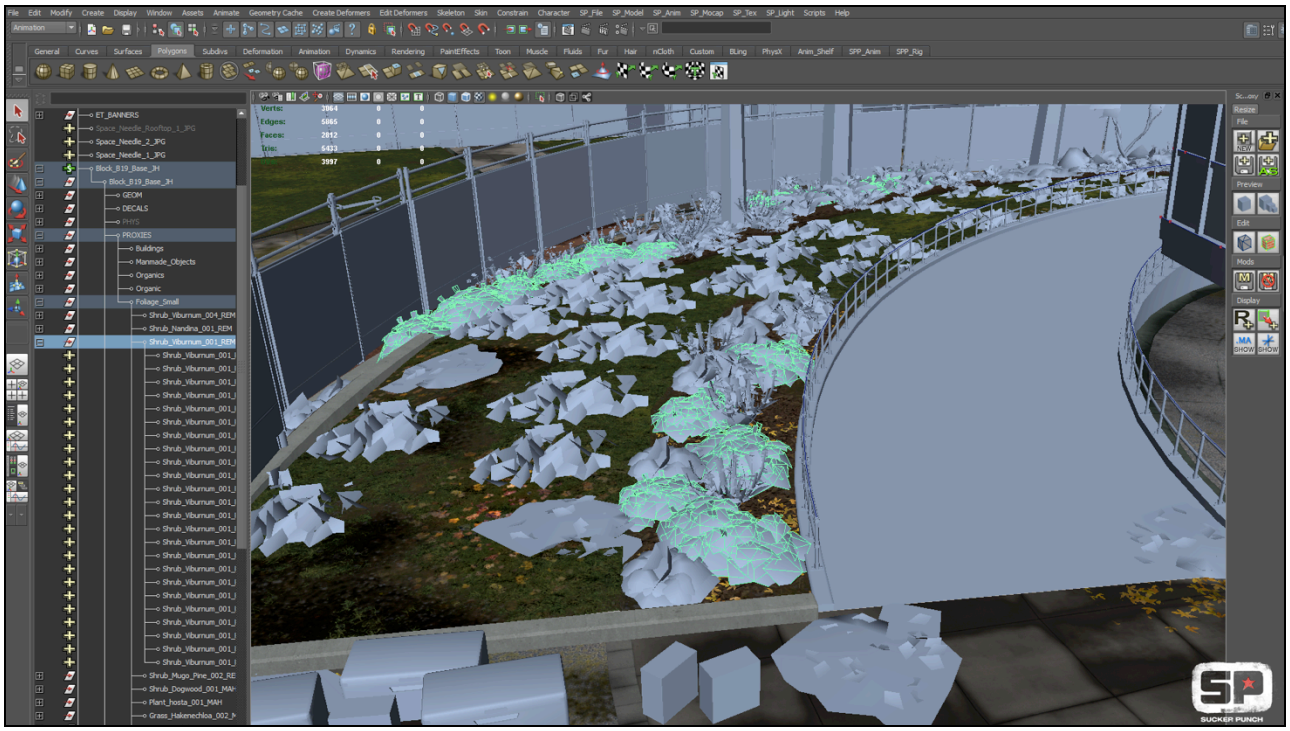
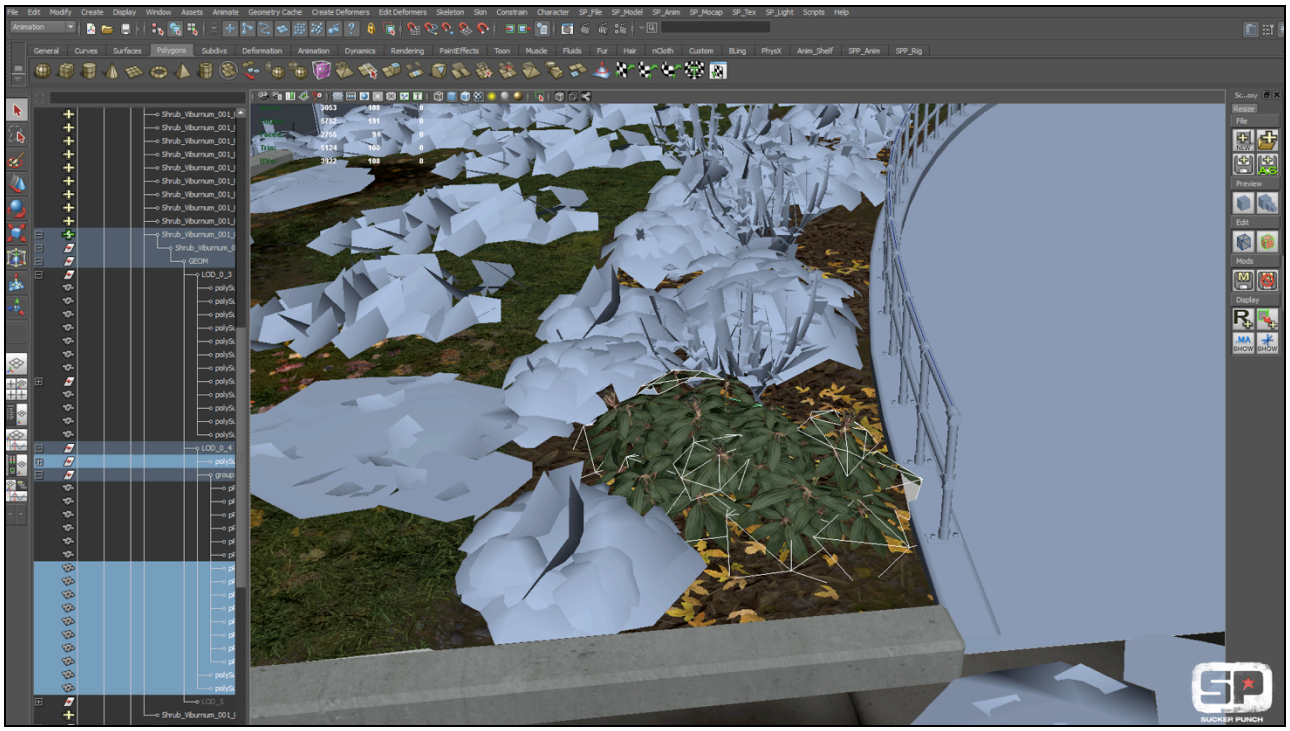


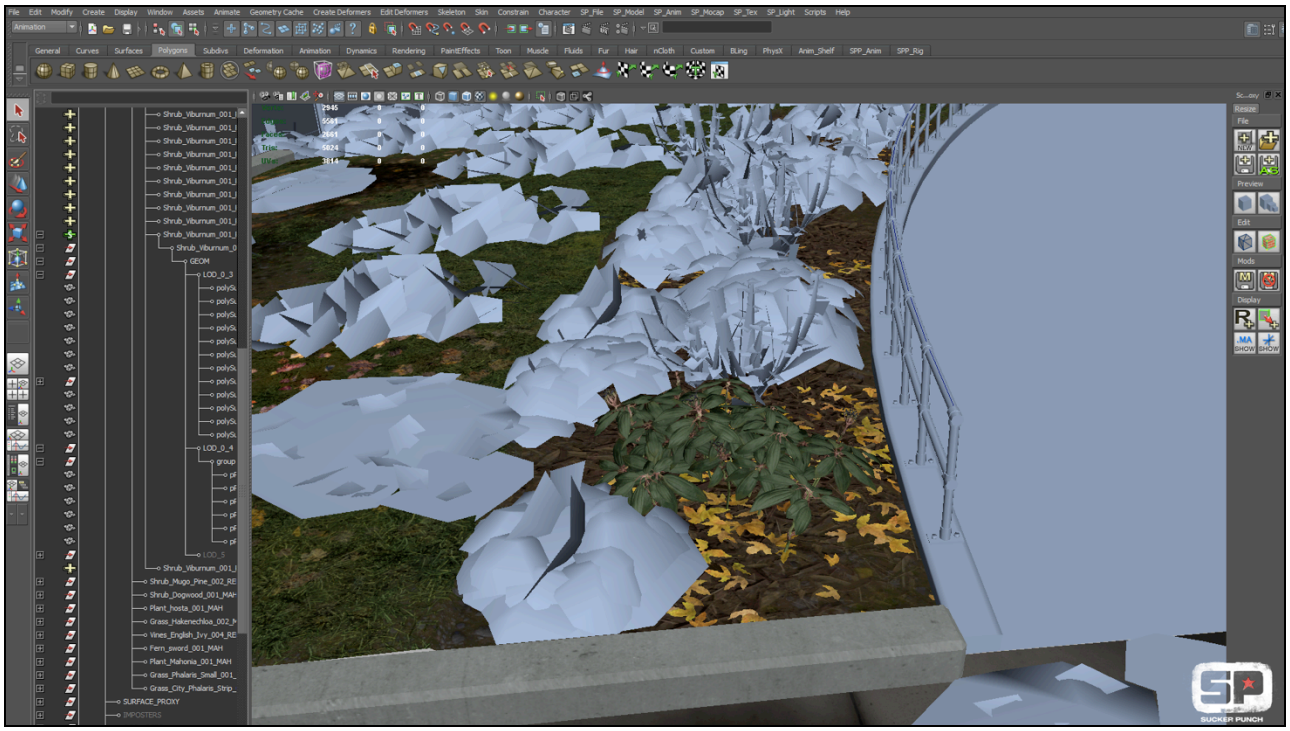
Image showing fast drawing proxy mode with a block open.



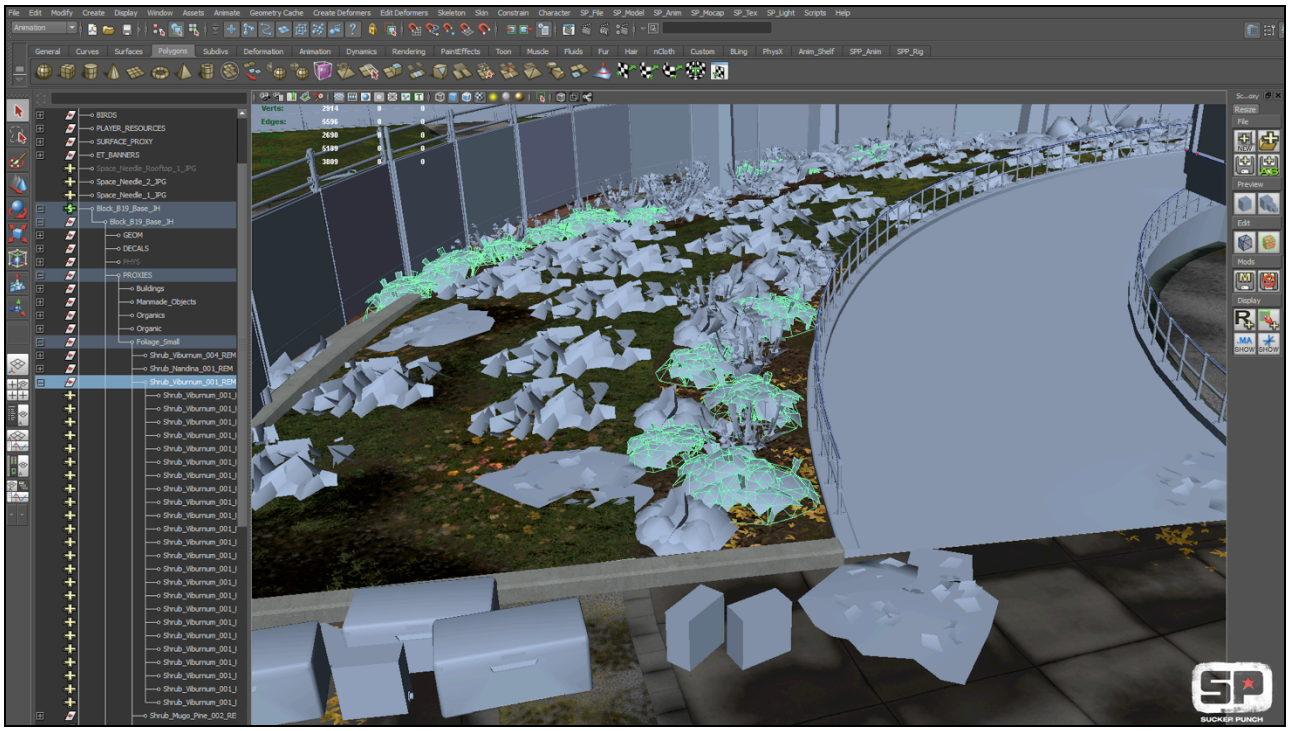
I edit the base file.



Edit the bush and select the leaves I want to delete.



Delete them.



Pop it back into preview mode. Notice the change propagates to all instances.

So proxy makes maya usable for most editing. But looking at the entire world, it's still slow. For cases where you need world scale context or accurate lighting we have an in game editor.

IN-GAME EDITING

Mostly object layout with properties
- Movement, parenting similar to Maya

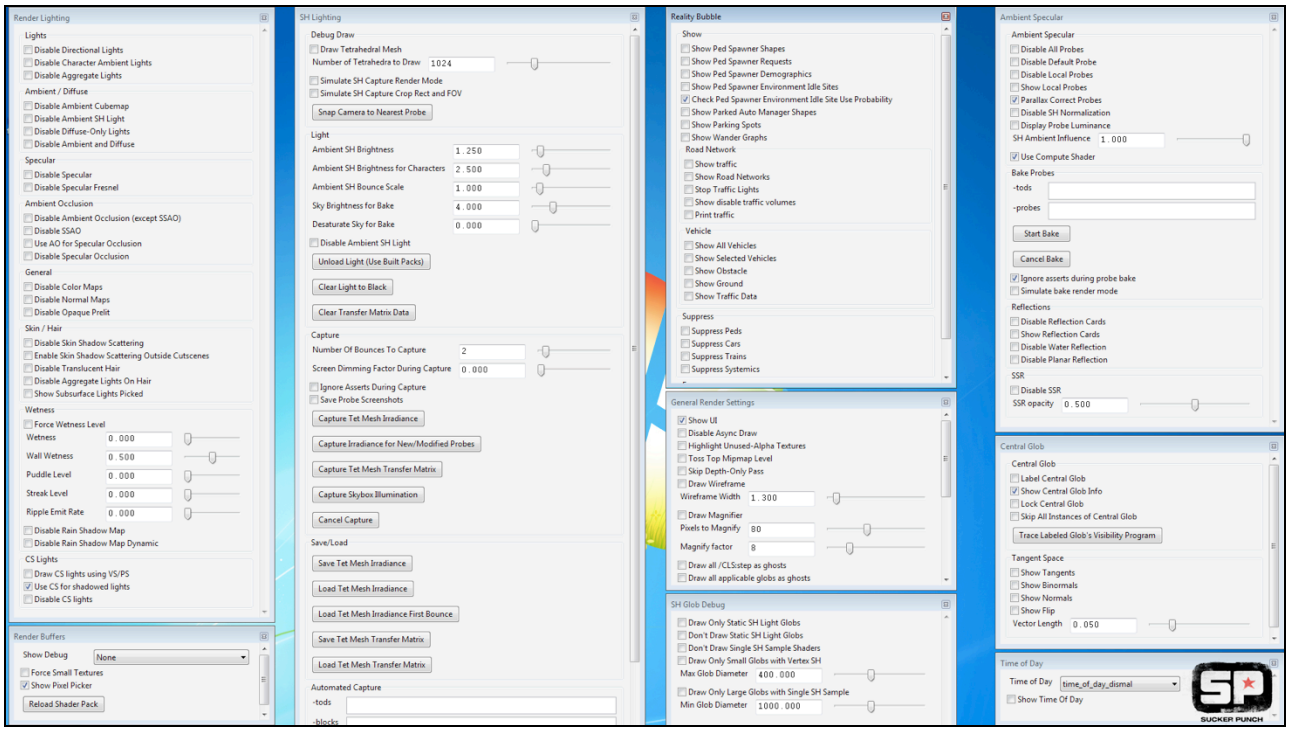
Instant feedback for some things
- Visualizing specific character flavors
- Cutscene editing with final lighting and post effects
- Full particle system authoring

UI is networked to avoid getting in the way
- Can toss to host and back

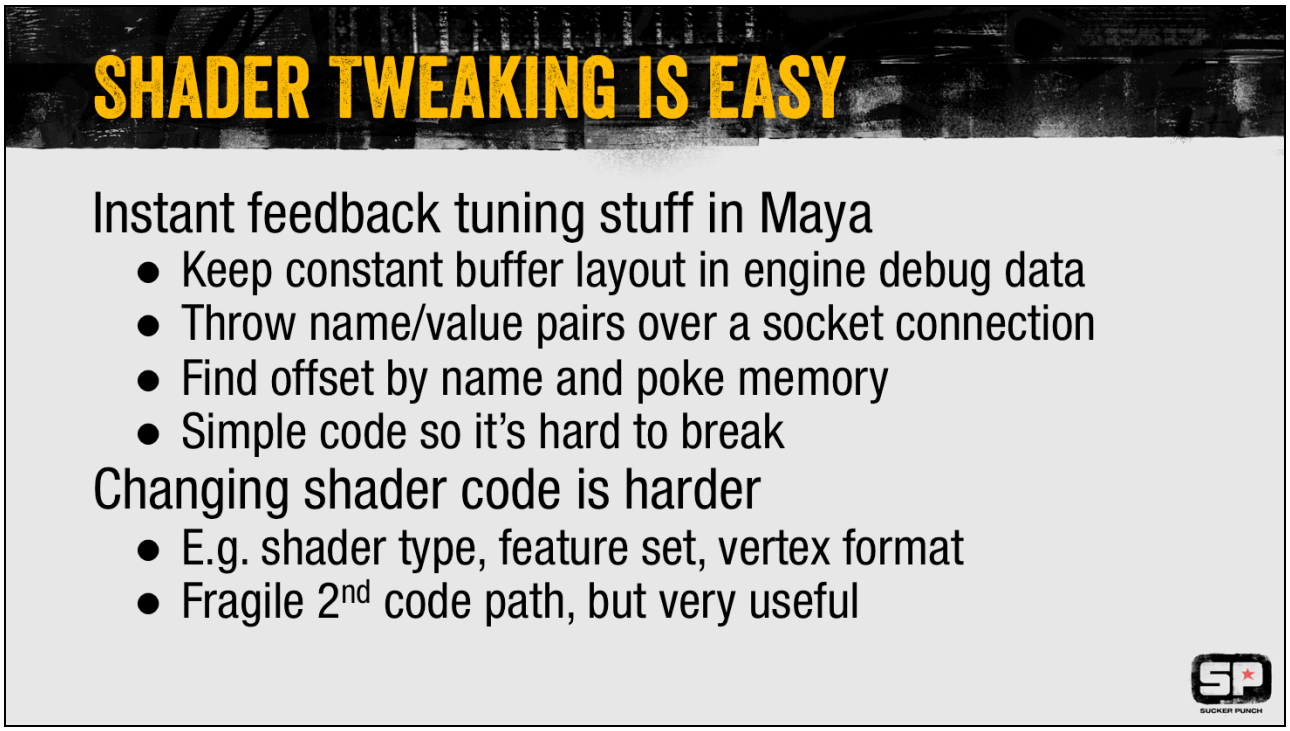
Getting instant feedback on some systems is especially powerful.

See bonus slides for more detail on networked UI.

Was a video of a particle system demo. Cut for space.



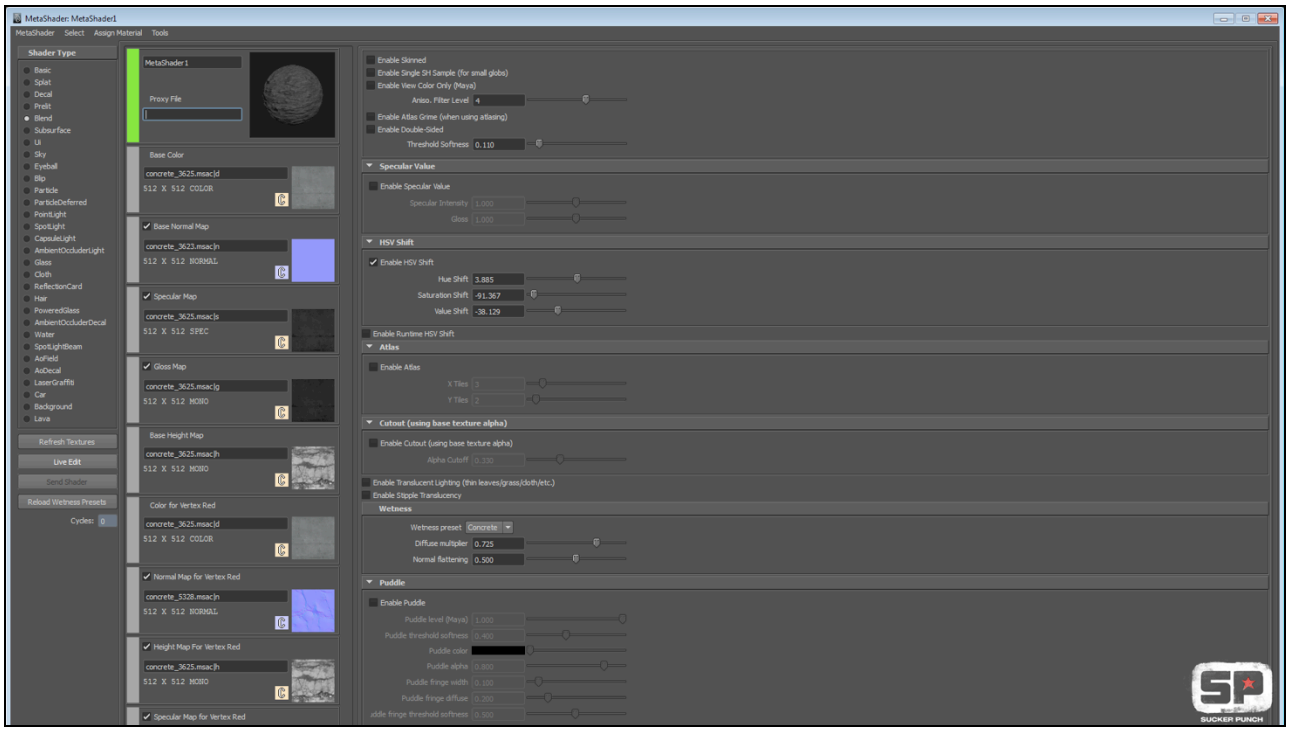
When you have networked UI, you can avoid opening and closing windows all the time to keep your screen real estate.



Sending newly compiled shader also sends associated textures. There are better ways to do fast texture iteration though.

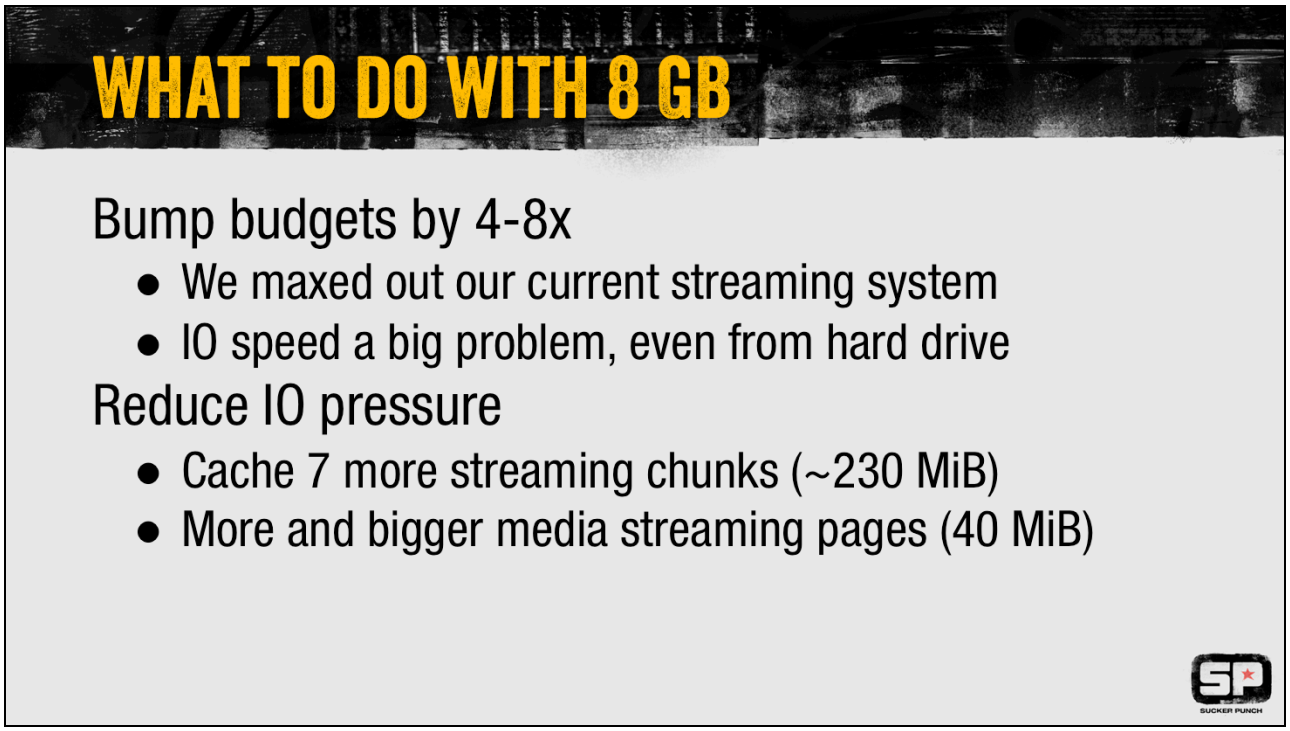
Since sending shader can change vertex format, send geometry too. It's a lot of data to get connected up right.

We did bust it for a few months at the end of the project when optimizing.

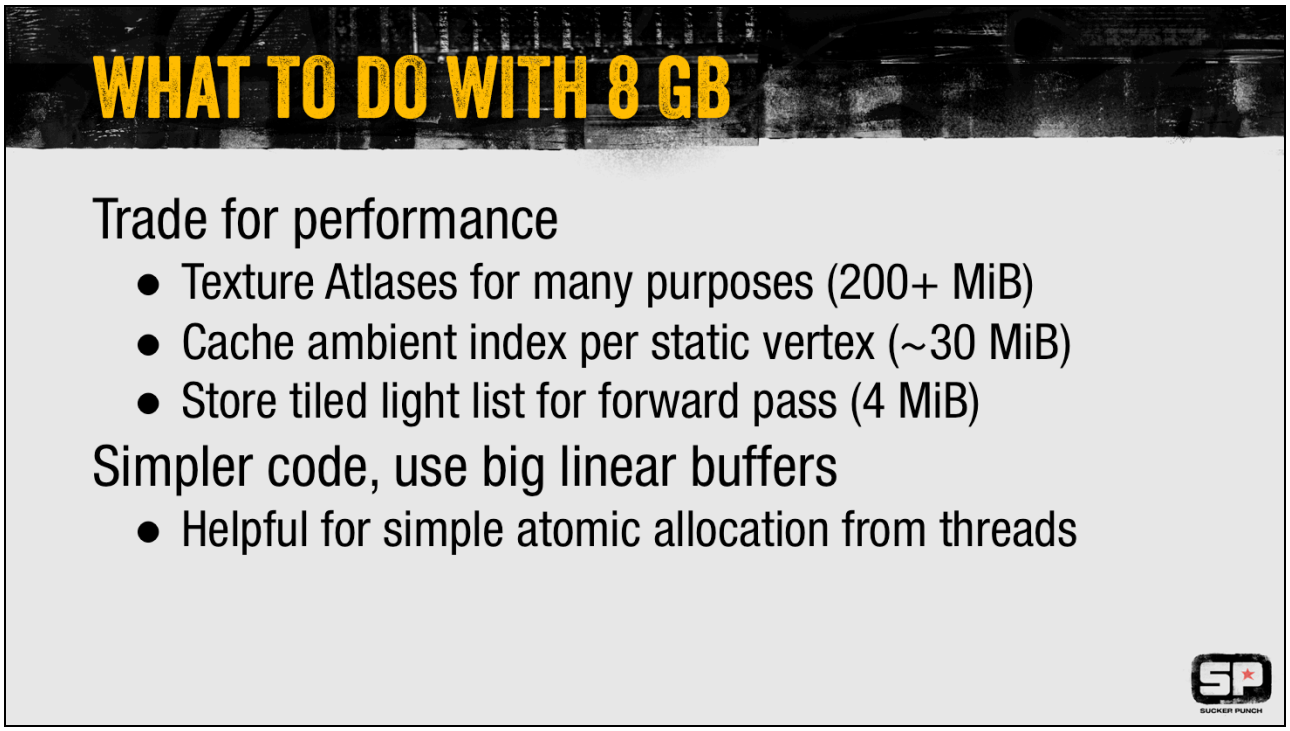


Checkboxes on the left change features and require shader recompile. We end up with about 3000 variations loaded at any one time (they stream in and out like everything else).

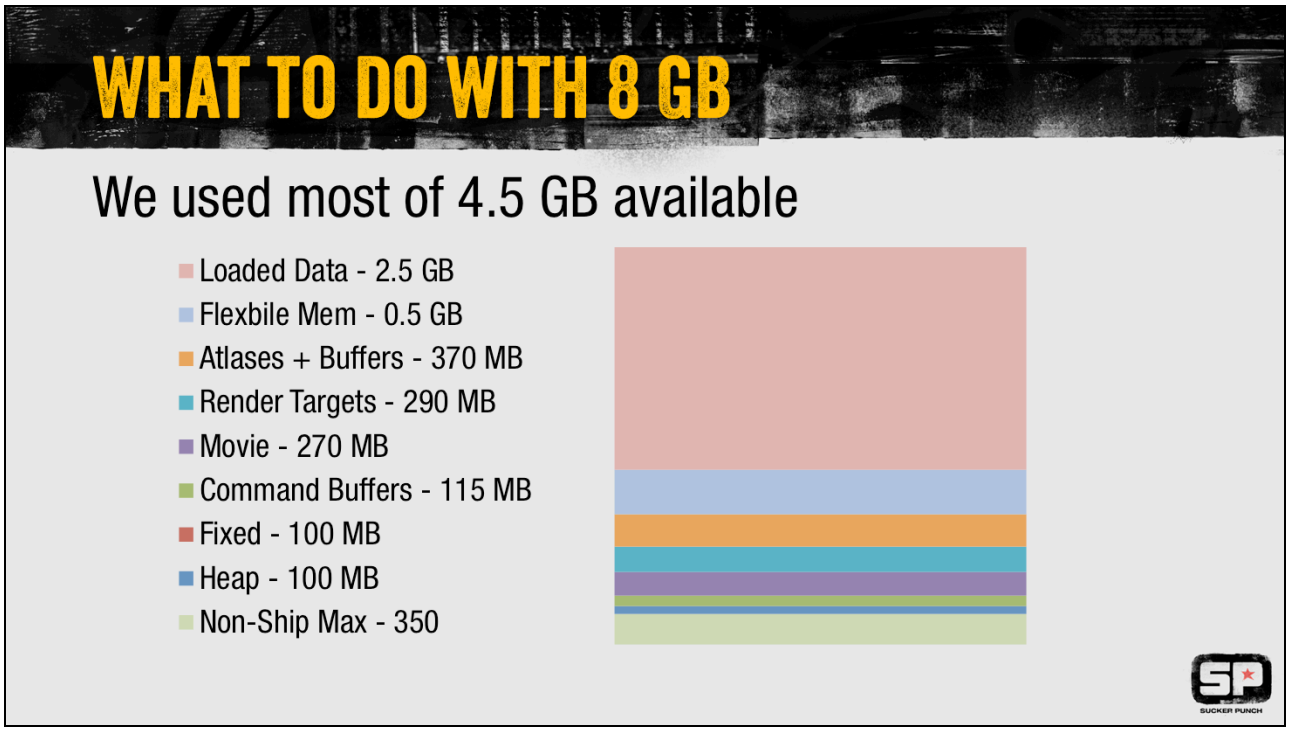

WHAT TO DO WITH 8 GB
SUCKER PUNCH



In the theoretical worst case, we barely survive 20 meters per second player movement with our world density.

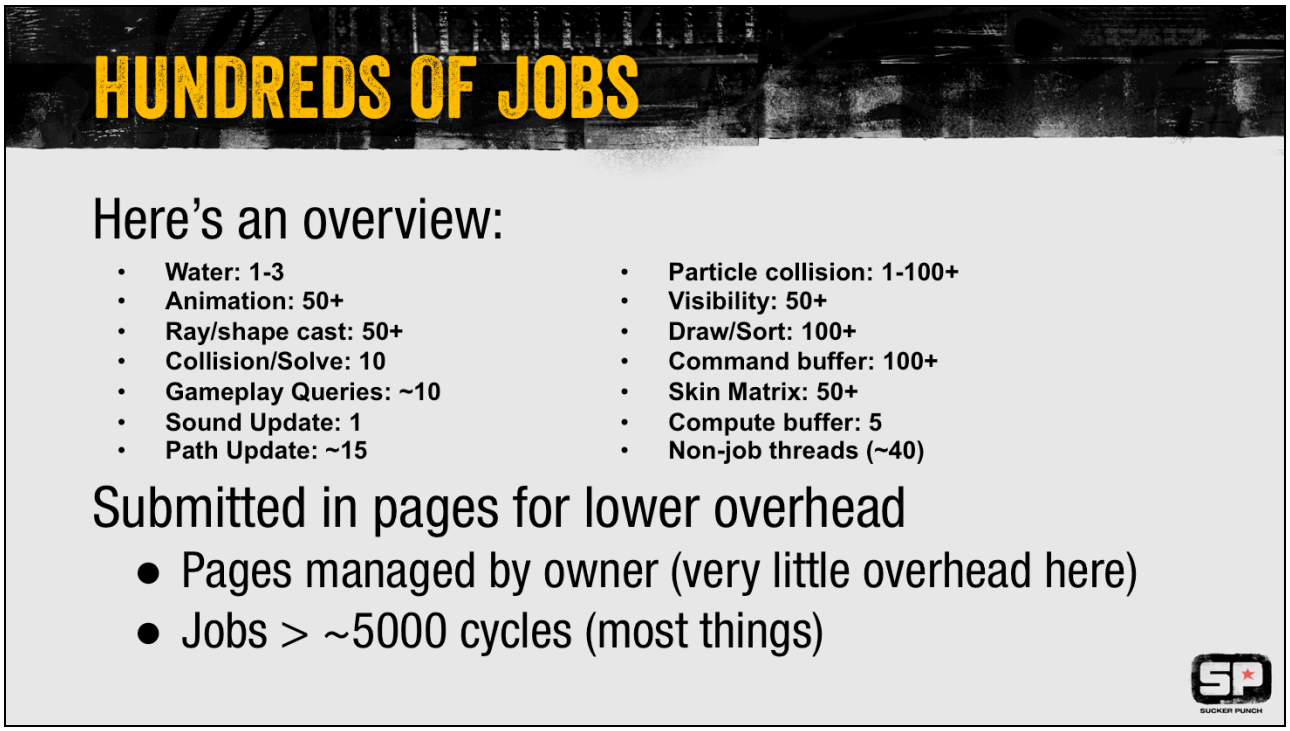


It's incredibly helpful to have enough memory to make things faster and simpler.

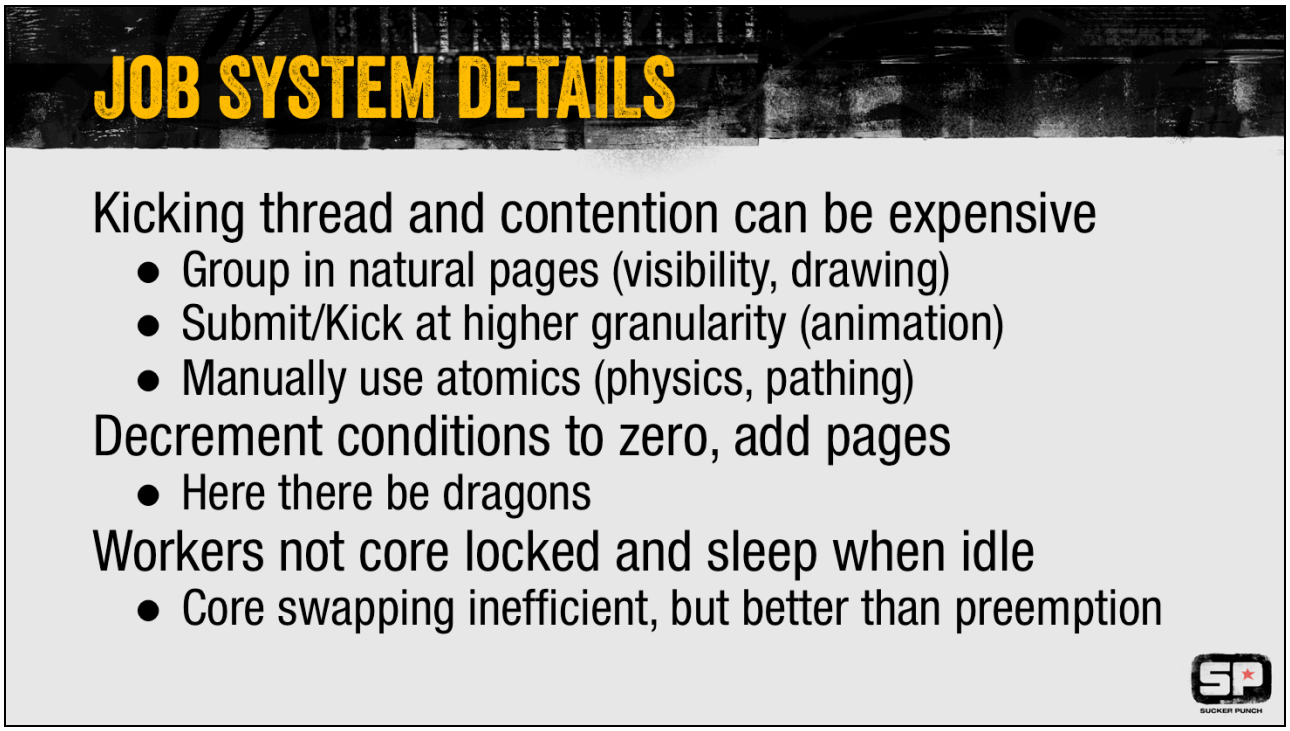


We're close to the limit, but there are hundreds of MB of slop to be had. We're still coming up with useful ways to use the memory.

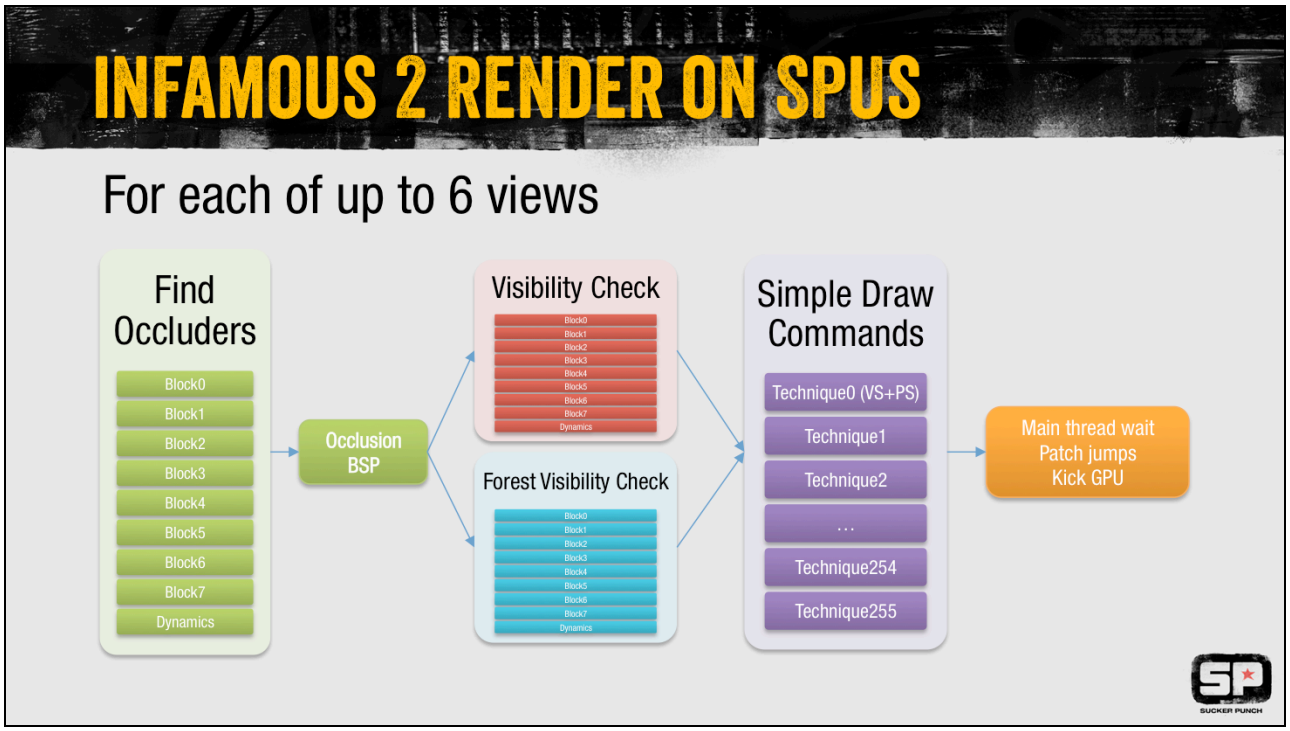
CPU THREADING



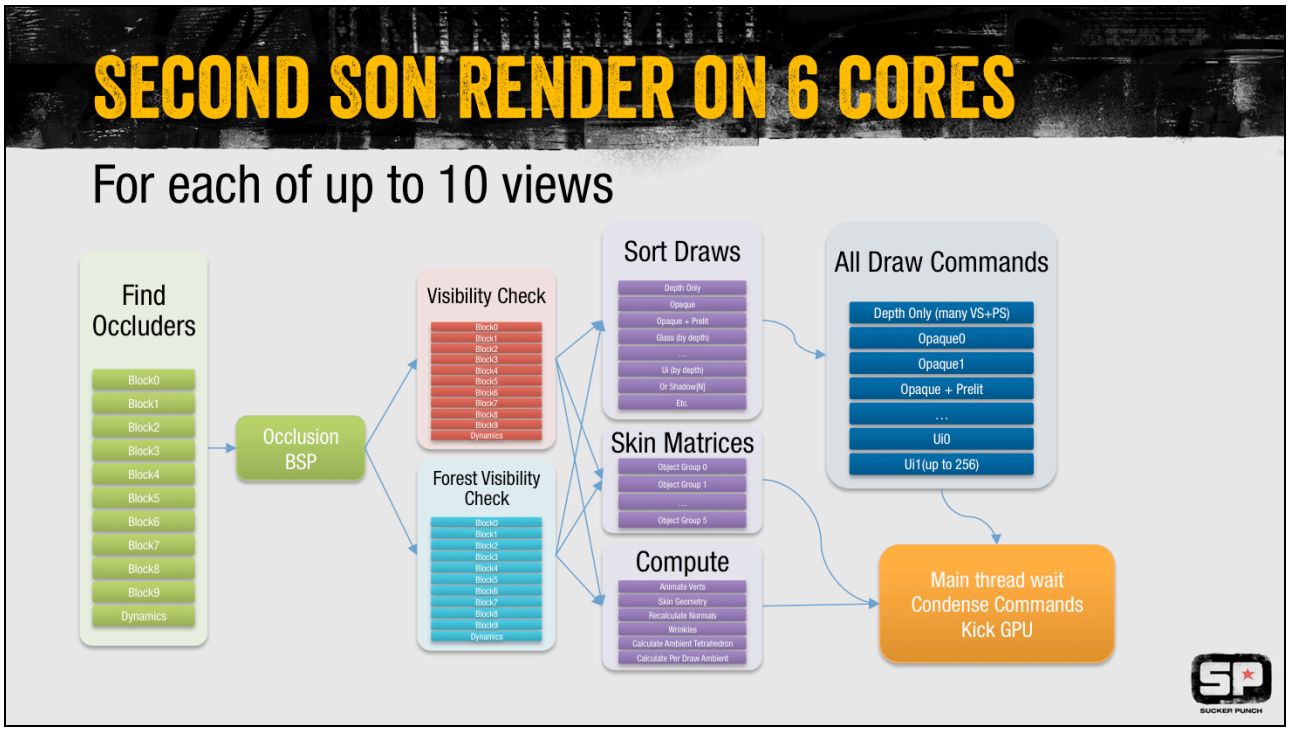
Worker thread per job pulling overhead likely higher than other systems. Still since main thread is optimization target, this wasn't a huge deal.



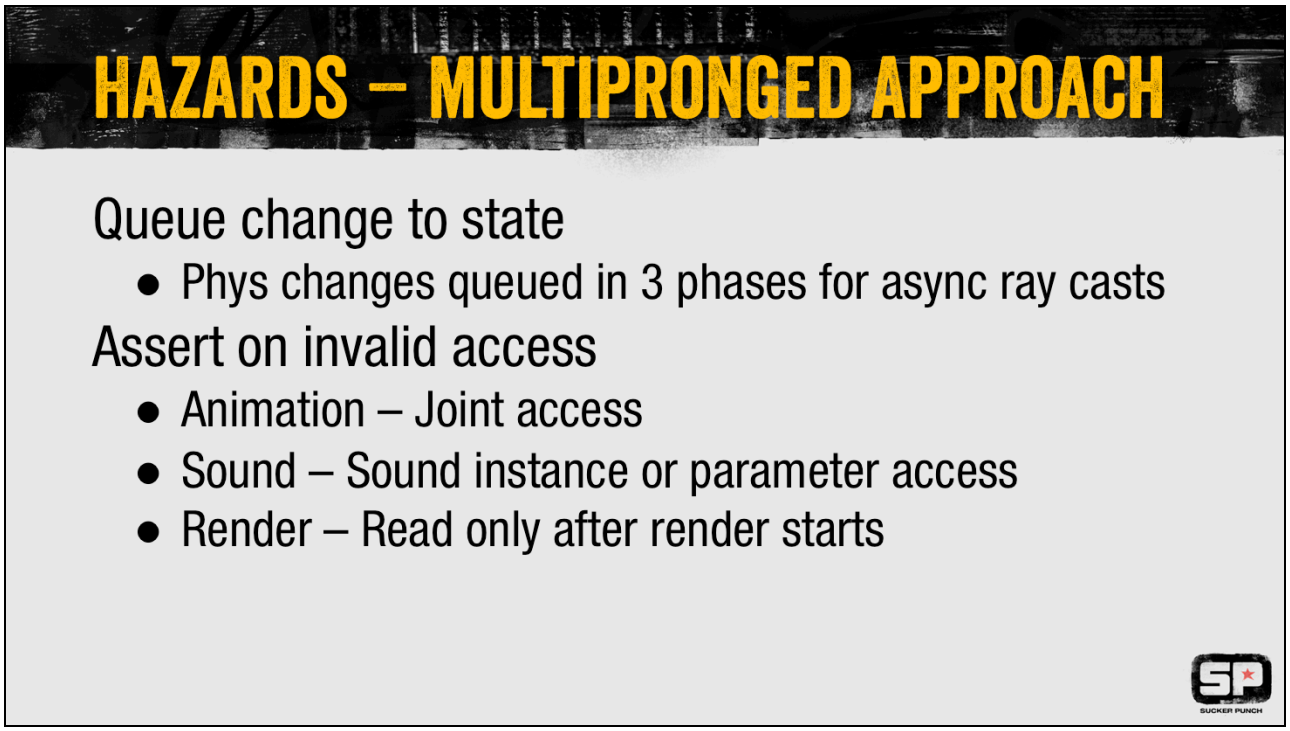
Room to improve here for sure, but simplicity has its bonuses.



Here's a layout of our PS3 rendering system. These steps are all simple pipelined DMA with execution code and writing out linear buffers of memory.



In Second Son we added more jobs. We front load a lot more work in visibility checking once we've decided to draw something. Overall the jobs are much more complex than they were before.



We use all the general hazard avoidance mechanisms that I know of.

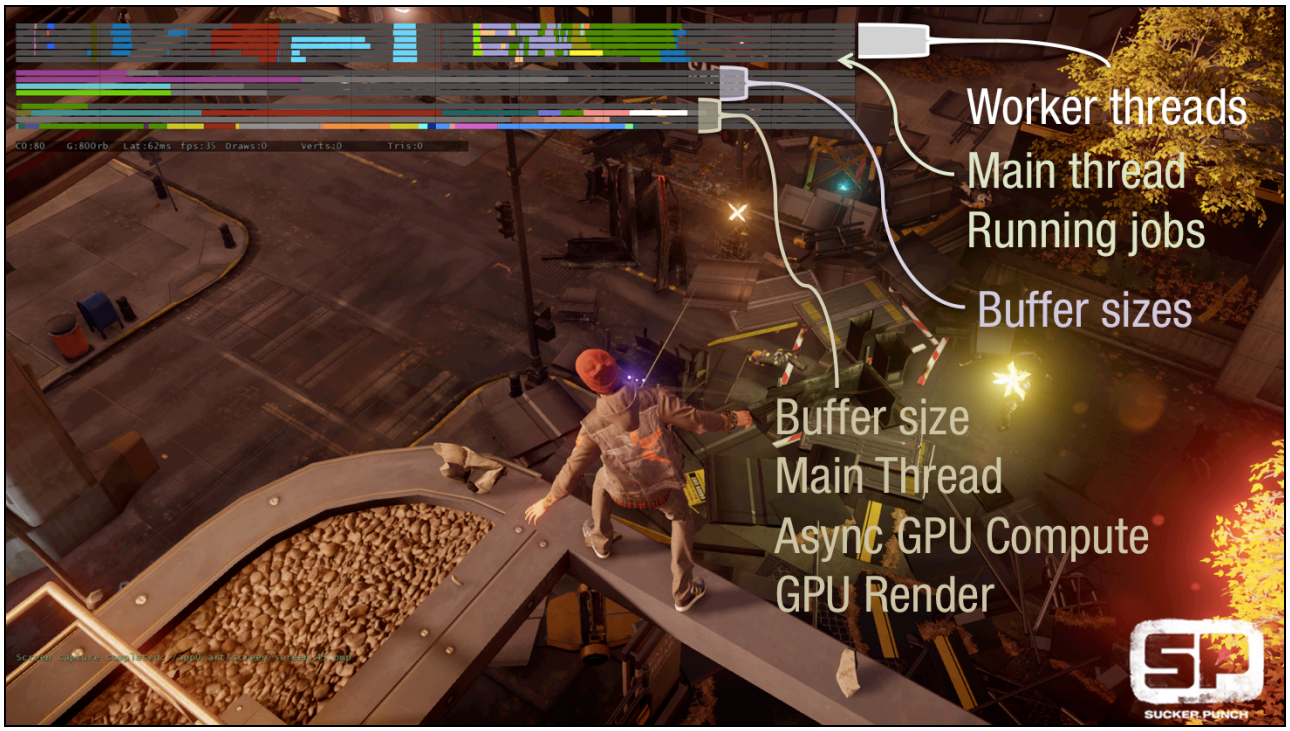
HAZARDS – MORE PRONGS

Build read only data structures
- For AI searches

Copy parameter data
- Water – current state
- Ray cast – Position, options, filter data
- AI queries – Position, options

Ad hoc
- Path mesh - assert where possible, but difficult

Still there is a lot of room for improvement in some areas here. We're likely going to double buffer more data going forward.



One 5 bars are worker threads. 6th bar is main thread running jobs (only when it can't do other work). Some of the empty space on worker threads may be due to sound thread kicking in.

Physics world changes are queued so we can run parallel jobs against a stable state.
See blue bars (ray casts) in the first window.

Objects may need to know about parent objects or objects they're standing on so there is explicit wait code to handle that.

We fork and do the render pipeline for as many views as we can. This allows us to get a lot of utilization out of the cores in this phase.

Havok also does fork join, though considerably less well balanced.

CPU THROUGHPUT

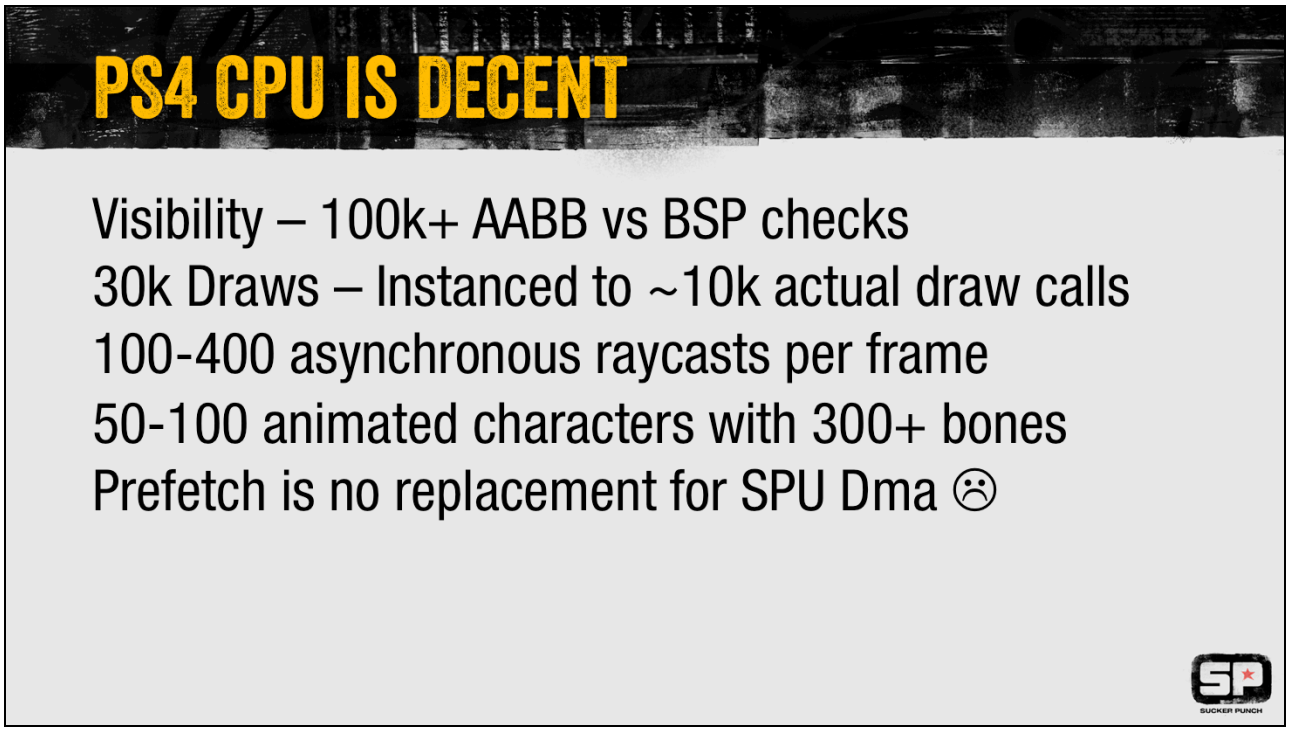
Fork/Join with Main thread overlap

- Simpler to code
- No extra input latency

Harder to max out CPU

- 50-70% main/jobs + 5-16% other threads

Draw code does have less branching per drawable thing. But there are drastically more shaders than in the previous game so that's probably a wash.



While the CPU has ended up working pretty well, it's still one of our main bottlenecks. It's also less easy to optimize after the fact because of the out of order nature.


McClain
Bank
AND NOW FOR SOME GRAPHICS
SP
SUCKER PUNCH

Here's how we store our material properties. Up to 8 gbuffers (5-6 + detph/stencil) is 41 bytes written per pixel. Yes that's 85 MB just for fullscreen buffers. Good the the PS4 had a huge amount of fast RAM. ☺

SHADING AND LIGHTING

Physically Based Materials and Lighting

- Less intuitive for artists, better for lighting changes
- GGX for specular, but started with Blinn Phong
  - Runtime conversion (see bonus slides for more context)

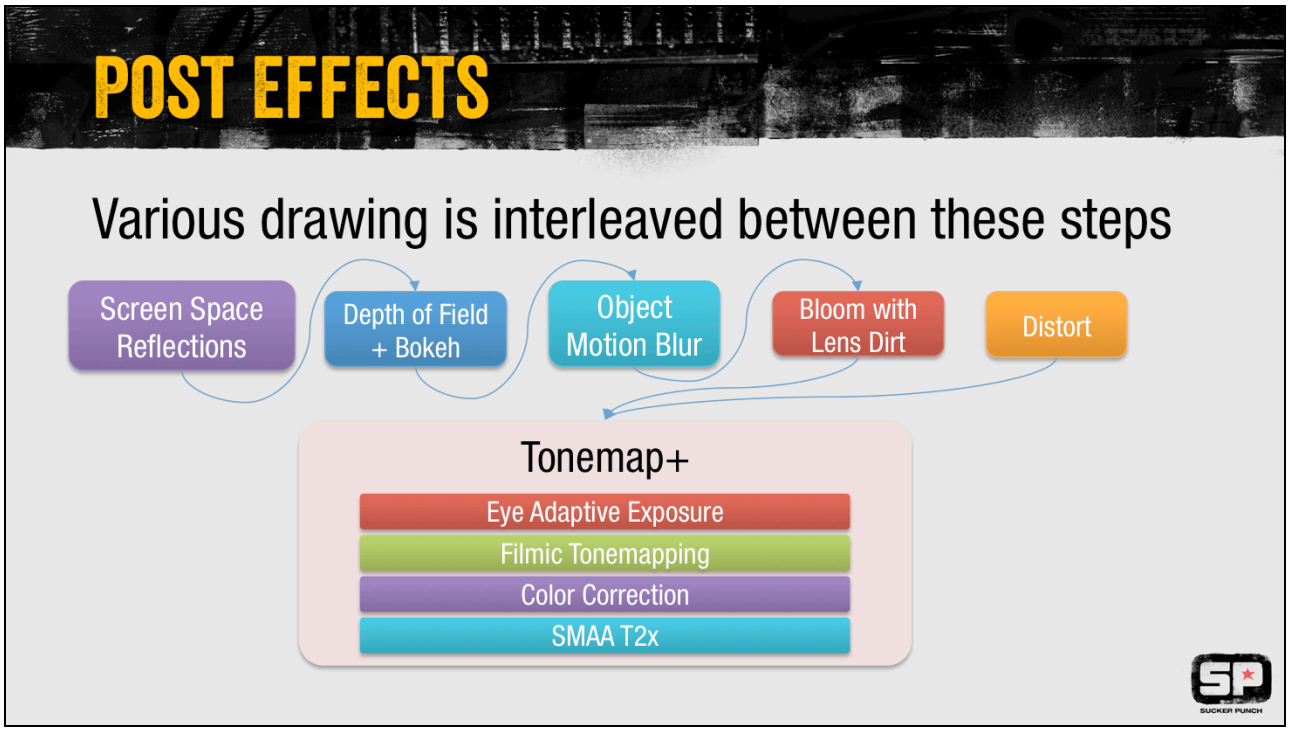
$$\frac{0.00209781 - 0.00294856g + 0.00111787g^2}{0.00257877 - 0.00024165g + 0.01471809g^2}$$

Deferred Shading, except when not

- Preintegrated skin, Anisotropic cloth/hair, glass
- Shadows – Normal Offset, 8x8 PCF, resolved to screen

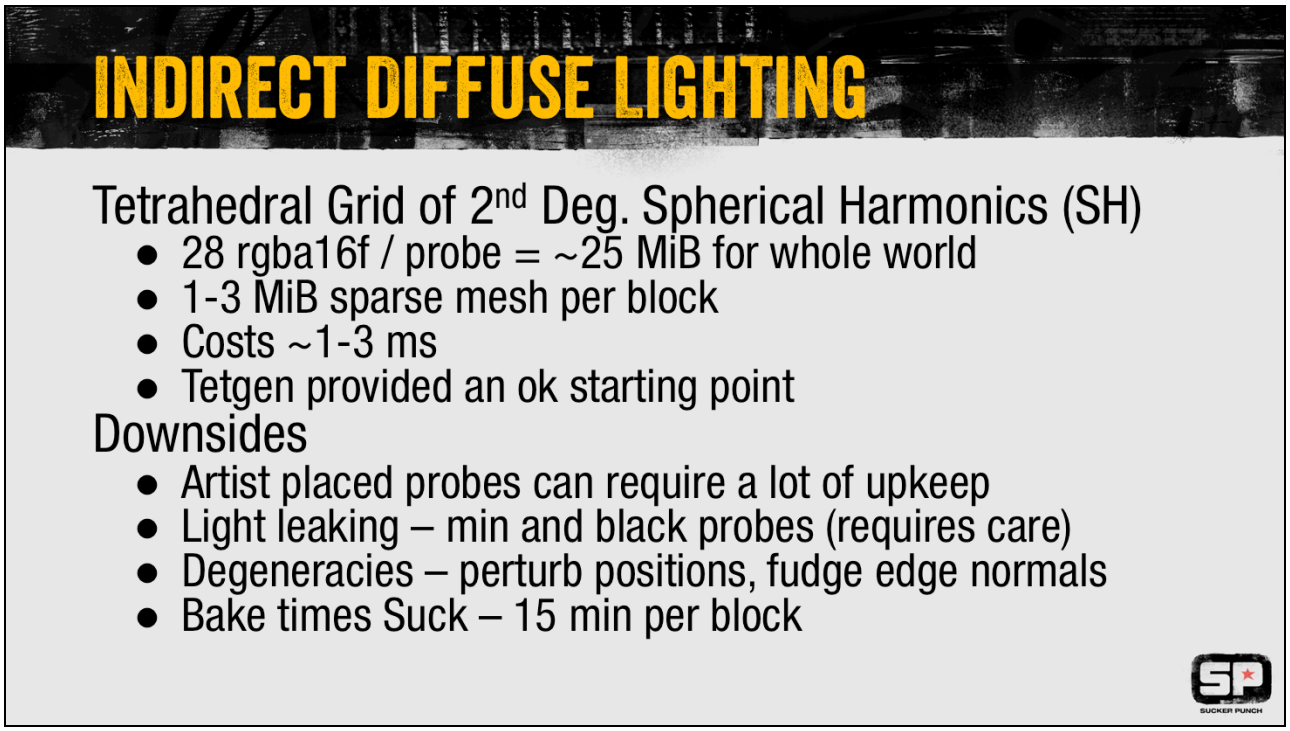
We spent a bunch of time training artists on physically based shading and added a photoshop constraint layers to keep things in plausible ranges. Artists paint full black and white range and named layer rescales that to physically plausible range. Not perfect, but worked ok.

We also switched to GGX late in our project. We ended up running a conversion at runtime.



Fairly standard state of the art stuff here.


MONORAIL



Seattle has overcast, so indirect is important. This was visually a bigger deal than we thought.

We thought about lightmaps, but they would have taken ~12 MiB per block just for UVs not to mention textures.

SAMPLING INDIRECT DIFFUSE

Accelerate initial lookups with 16x16x16 grid

Search/sample per draw and/or vertex

- Per pixel too expensive
- Artists can cut more verts if need be

Caching data to avoid redundant computation

- Cache last tet index per drawable
- Cache static per vertex tet indices

Without SH, single very low res diffuse cubemap.

With SH. Notice the higher contrast in occluded areas and how everything fits together better.

INDIRECT SPECULAR LIGHTING

Local Specular Cubemaps

- Prefiltered, select mip level using gloss
- Distort based on distance to edges
- Requires lots of space, barely a fit for open world

Screen Space Specular Reflection

- Mostly brute force at quarter resolution
- Lots of fiddling to get right balance of artifacts
- Avoid artifacts by limiting blockers and reflectors

Seattle is often wet so spec is important. Spec probes 256x256 and we have up to 256 of them. Practically about 175 MB of RAM, which takes a long time to load. Atlas takes more space, though we could optimize away 2nd copy.

With out local spec probes the sky shows up more than it should.

With local spec probes. Everything feels much more anchored.

Even local spec probes can’t capture everything though. Dynamic objects, destructible stuff and tall thin constructs are hard to deal with.

Screen space reflection gets us more grounding for all of that.

ISSUES WITH SMALL DRAWS

Merge geometry for fewer draws?

- Less culling, shader overhead, texture atlasing, meh.

Sort by shader – 3x faster observed

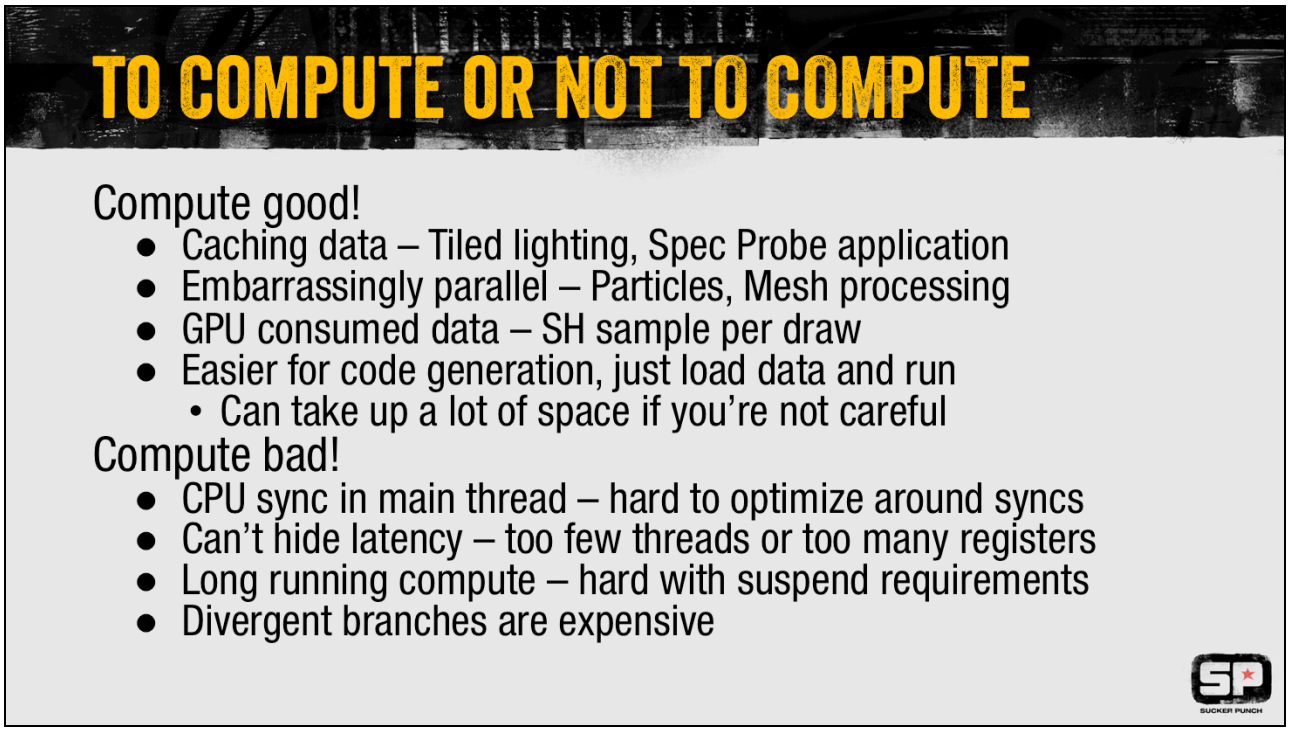
Instancing – 5-20% observed, 6x in ideal case

- Atlas all per draw data in buffers
- Can map all of memory as a single buffer

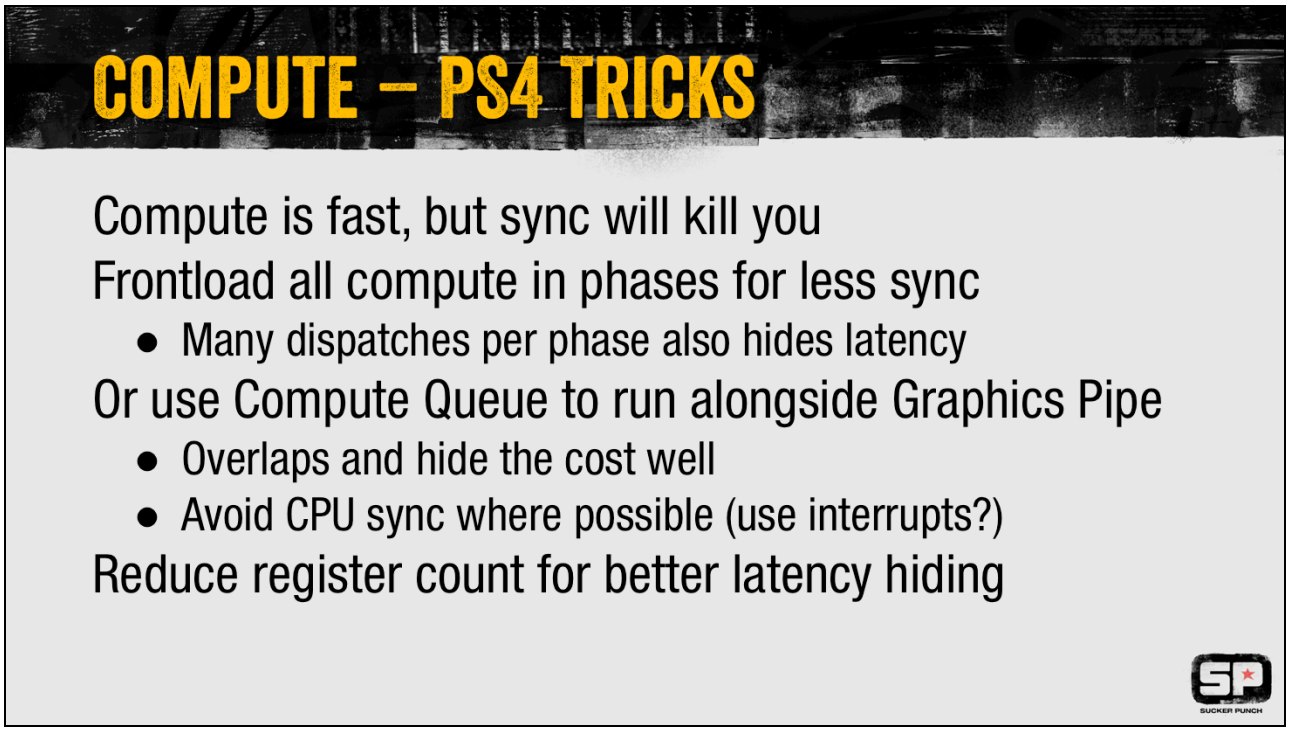
Doesn't work for depth sorted translucent draws

Capturing your frame for later inspection will be very difficult if you are mapping all of memory though.


COMPUTE
SUCKER PUNCH



Fairly standard advice for compute still applies. Especially if you're running it in the graphics pipe.



Through a combination of code changes and compiler improvements, we were able to speed up our tiled deferred pass by 75%. This is the power of registers. It will also matter a huge amount if you're running in a compute queue. Too many resources and won't be able to overlap with other execution.

Compute queues are definitely the way of the future though.

Synchronous compute put in phases at the beginning of the frame given to the graphics pipe. Particle compute is run overlapping with the graphics pipe (at arbitrary time).

COMPUTE FOR POST EFFECTS

PS often slightly faster than same compute code

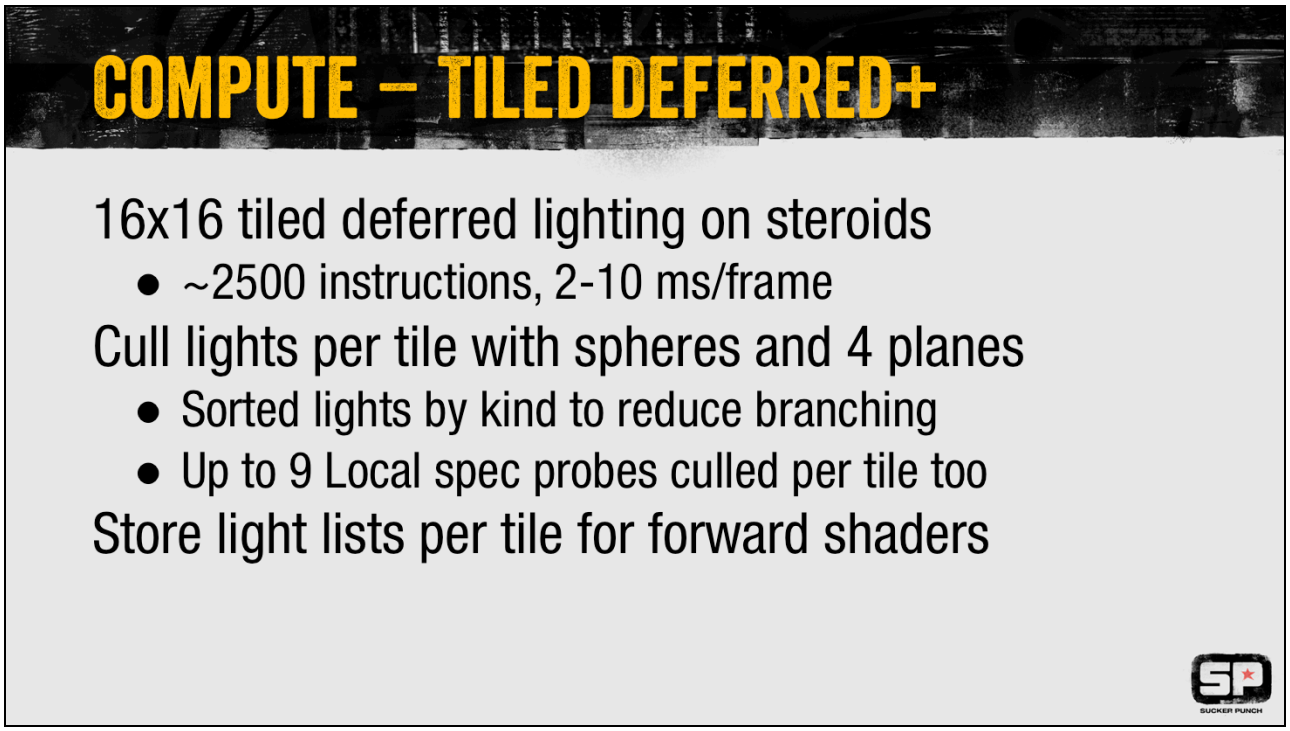
- Raster tiling matches complex image format?
- ROP caching and better pipelining?

Best if using compute specific feature (e.g. LDS)

- Care still required to beat PS (e.g. texture tap for blur)

Always use Tiling (8x8 or bigger)

- Try not cross 64b cache lines in fetches



Register count can be a killer in big shaders like this.

COMPUTE – FACIAL ANIMATION

Always 1 Dispatch per mesh
Streamed vertices – 3b/vert w/ scale/bias
- 215 instructions, 0.03 ms for 140k verts

Skinning – Required for wrinkles
- 155-271 instructions, 0.17 ms for 272k verts

Recalculate Normals/Tangents – avoids streaming them
- 137 instructions, 0.17 for 140k verts

Wrinkles – Sparse triangle area calculation
- 81 instructions, 0.003 ms for 4 faces (768 verts total)

Running compute up front requires lots of memory
- ~512k per character per view!

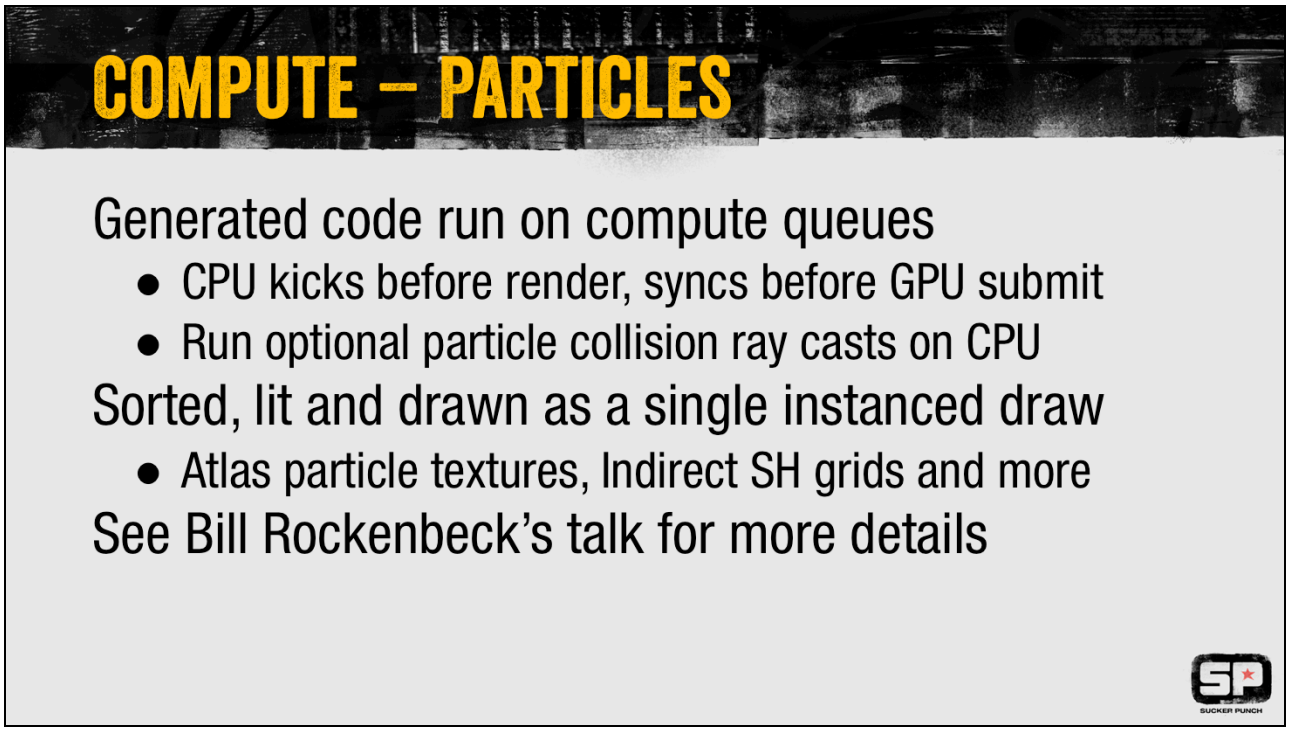
COMPUTE – AMBIENT SH

Calculate tetrahedron index per vert

- For loop to find tetrahedron (similar to VS)
- 1 dispatch/mesh, 233 instructions
- 0.2 ms for ~61k verts in 379 dispatches

Calculate per draw SH sample

- 1 dispatch for frame, interpolates SH per draw
- 315 instructions, 0.017 ms for 1334 SH samples
- Move to GPU from CPU (~40x faster)



Biggest downside is generated compute shader size (15+ MB of loaded data).

THINGS I WISH I KNEW

Retroactive GPU validation is gold
- Could have saved months of people's time

Importance of dealing with small draws
- Instancing everything and generic draw sort was big change

Compute is Super Awesome
- Give more systems GPU friendly data structures

Baking is Super Slow
- Need a solution that doesn't eat artists alive

Perforce sync time eats my soul
- Asset sizes demand more thought here

FUTURE

Much more threading, compute
Lighter weight instantiation
Perforce sync time improvement
Less manual ambient & speed up baking
Better distant environment LOD
Pathing system needs an overhaul
Would like easier scripting reference to parts of objects
Fewer heavy weight objects would be nice
I should really stop….

THAT'S A WRAP

Thanks to all SPers who worked on these systems!

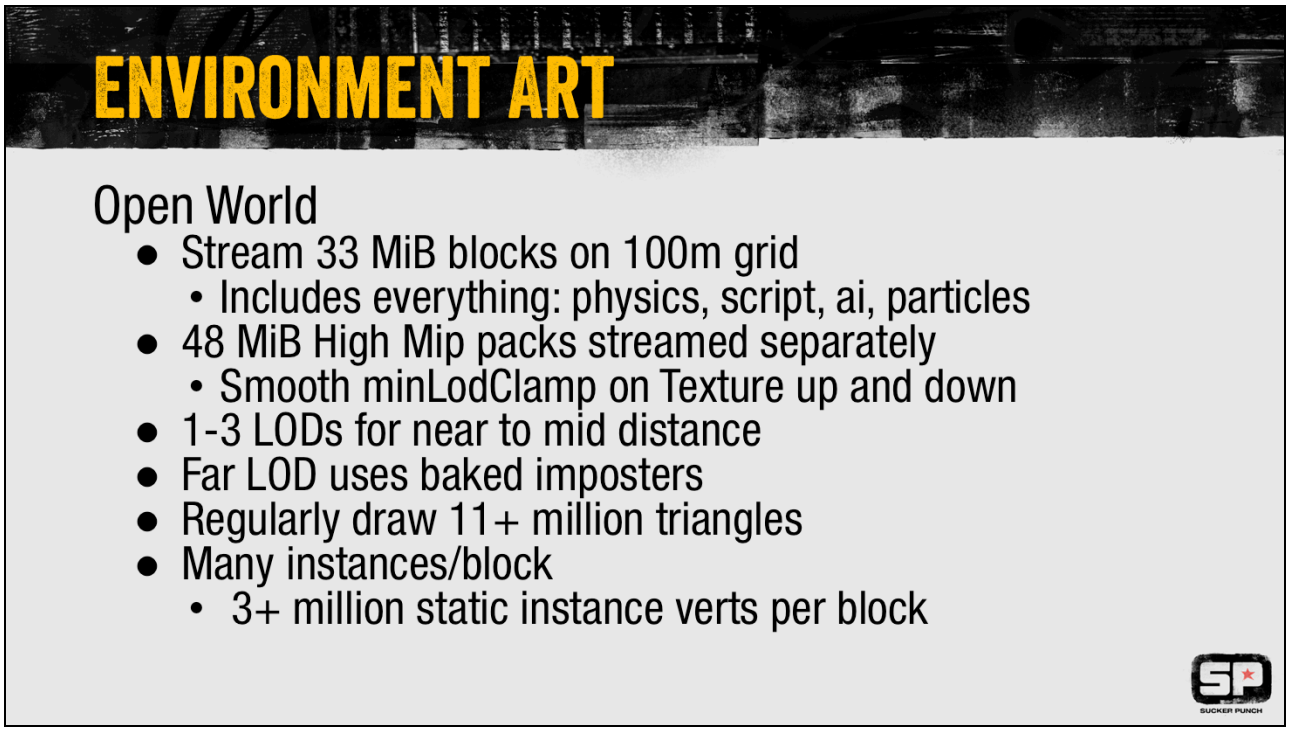
References

- Light probe interpolation using tetrahedral tessellations
  - http://robert.cupisz.eu/post/75393800428/light-probe-interpolation
  - TetGen – http://wias-berlin.de/software/tetgen/
- Siggraph13 Physically Based Shading course is a good intro
  - http://blog.selfshadow.com/publications/s2013-shading-course/
- Specular cubemap warping
  - http://seblagarde.wordpress.com/tag/specular-lighting/

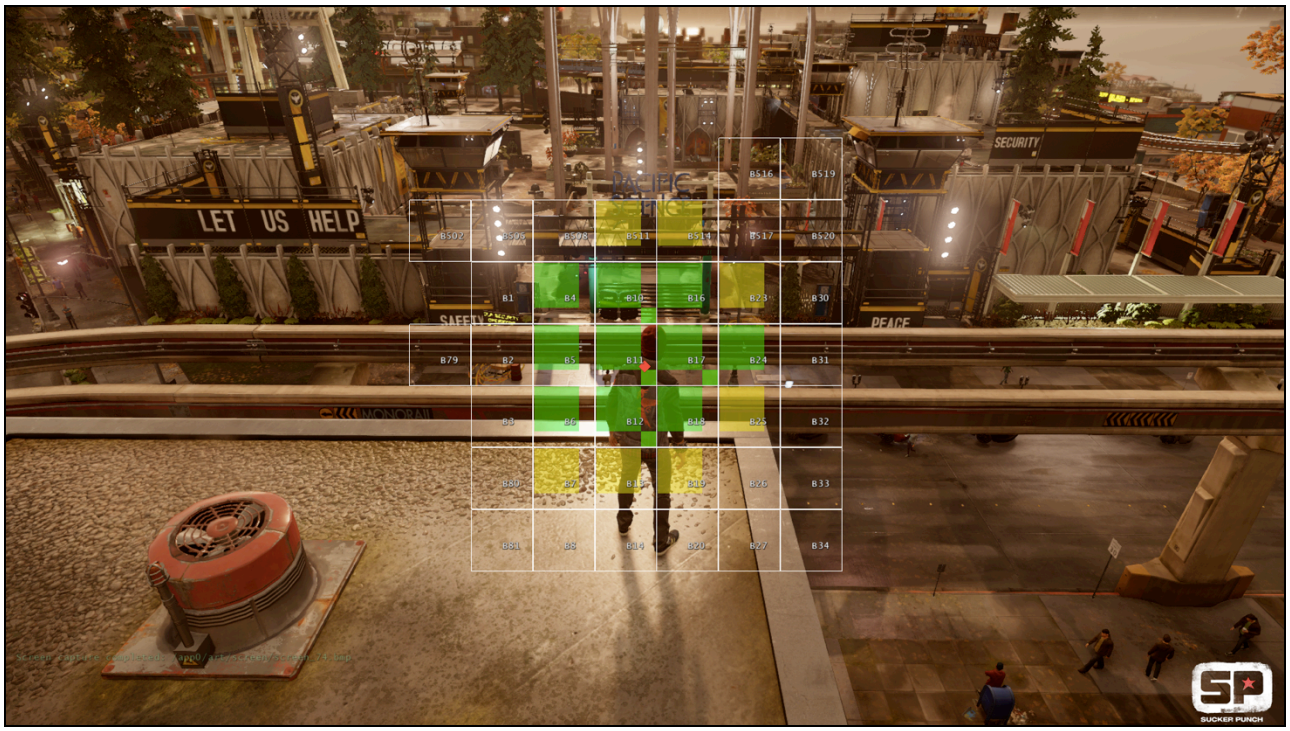
QUESTIONS?

Email: adrianb@suckerpunch.com
Twitter: adrianb3000
Slides: posted or linked on adruab.net

BONUS SLIDES



We only display 10 blocks for performance reasons, an extra 7 are kept around to reduce file IO in pedantic cases. Only 4 high resolution texture blocks are included (for legacy memory overhead reasons).



Big squares are the primary loaded data (geometry, physics, behavior, low res textures). Yellow means loaded but now spawned (doesn't cost us runtime performance). The little squares in the lower right are if the high resolution textures are loaded for said block.

CHARACTER ART

Characters

- 300-500 bones – lots for face, cloth, and hands
- 120k polys, more LODs for NPCs
- Up to 28 MiB, mostly textures
- Thousands of animation clips
- Per vertex facial animation for cutscenes

FACIAL ANIMATION – SEPARATE TALK

Lots of freedom in Maya

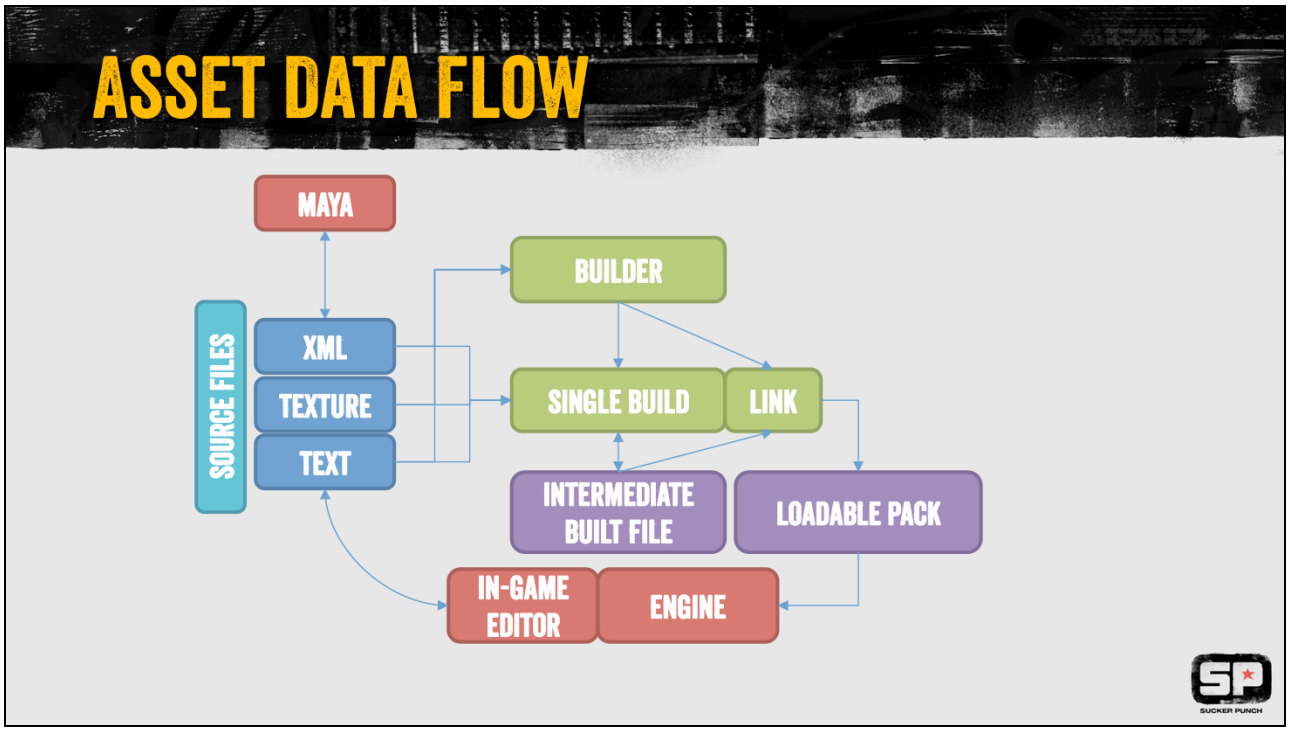
- High resolution and animated mesh retargetting
- Local pose space deformation, wire deformers, etc.

Build and Runtime

- Vertex animation – 3 bytes per vertex per frame
- Smooth normals at runtime, procedural wrinkles
- Care needed when unskinning a mesh

See Spencer's Performance Capture Talk

If you unskinning a mesh requires taking the interpolated skinning matrix per vertex and inverting it (cross your fingers its invertable). If you just skin using animated bones to bind space bones, cross terms will cause mesh distortion.



Maya reads in exported files for fast preview (called proxy).

Builder knows all, including complex recursive merging dependencies.

A build can read in child built files previously built by the builder.

In game editor can also trigger builds using command line options.

There are more tools than shown here.

If you have a PC build multi monitor is a very useful thing and satisfies many of our current desires.

NETWORKED UI

Throw UI to host PC

- Send commands to devkit and contents to host

Declared near related code

- Easy to setup something to tweak
- Menus gathered from globals (has pros and cons)

Index all menus/commands for searching

NETWORKED UI DOWNSIDES

Two code paths to maintain
- Quite a bit of code to write twice

Windowing system quirks

Can pound network connection
- Refresh big window every frame

Less compelling for multiplatform engine
- Better tuning on each platform though

Better than a throw away PC engine ☺

STREAMING

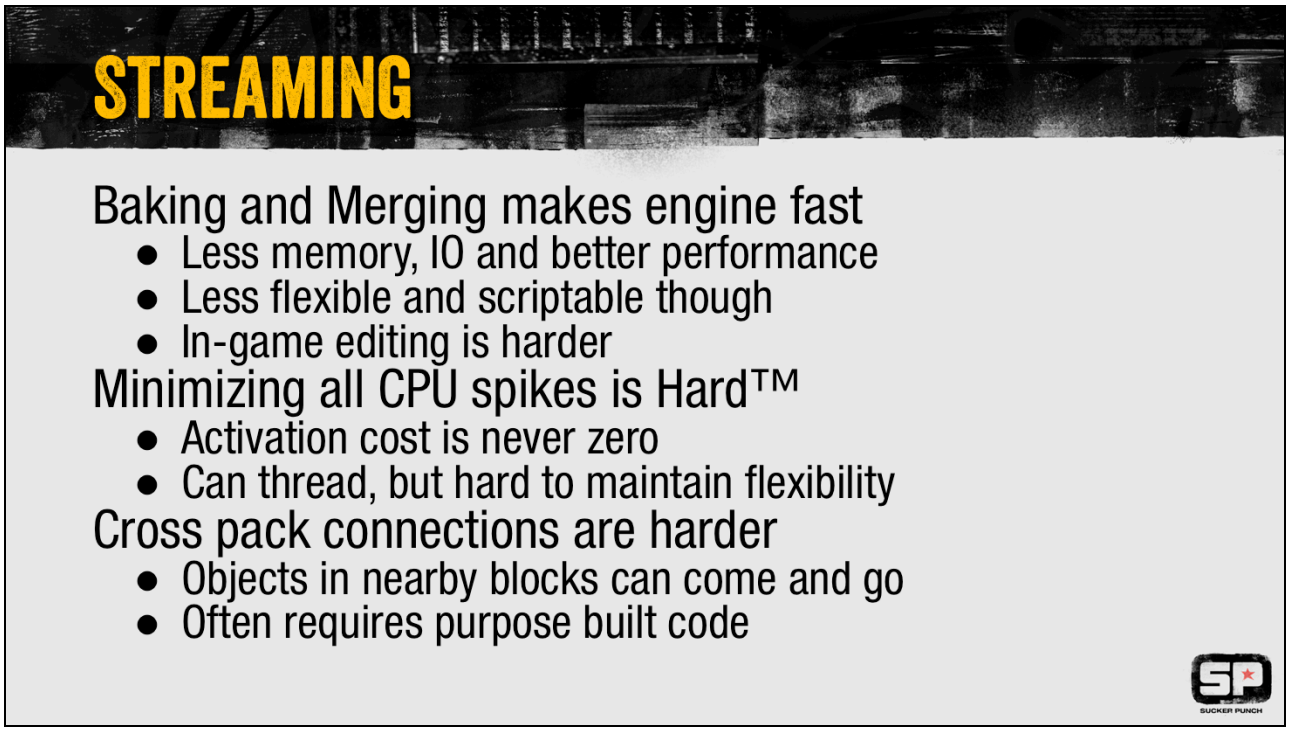
Per block/character page budgets

- Simple so artists can be responsible for memory
- Bake low level bits together and instance
  - drawing, physics, parkour, breakables, drainables, etc.

Load directly to memory plus pointer fix up

- Reduce pack activation cost as much as possible
- Development only data and patching
  - enums, script types, string table, maya paths

Can use uniform page flexibility to cache more copies of characters or blocks than you need to reduce file IO pressure. Does restrict you to maximum page size though. Moving toward using virtual memory.



Can't bake relationships across blocks without having much slower multi pass compilation. Entirely runtime relationships can work, but require more code.

GLOSS ENCODING

Old Blinn-Phong gloss encoding

- Power function = 8192^gloss

Convert at runtime to GGX roughness

(courtesy of JasminP)

$$\frac{0.00209781 - 0.00294856g + 0.00111787g^2}{0.00257877 - 0.00024165g + 0.01471809g^2}$$

SHADOWS

Quantized Cascade Shadow Maps
- 3-4 2k x 2k shadow maps + 4 for spot lights

Percentage Closer Filtering
- Mostly 8x8, 4x4 for farthest cascade
- Variable sharpening per cascade for look
- Skin 16x16 plus preintegrated shadow scattering
- Hair 4x4 plus screen space ray marching

Normal Offset Shadows

Resolved to screen space for performance