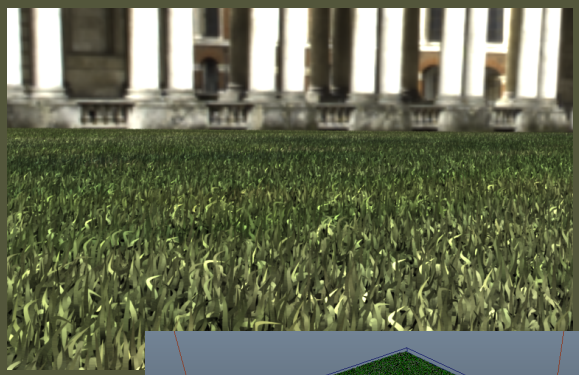
A GPU-Accelerated Node Based Framework for Hair Simulation and Rendering

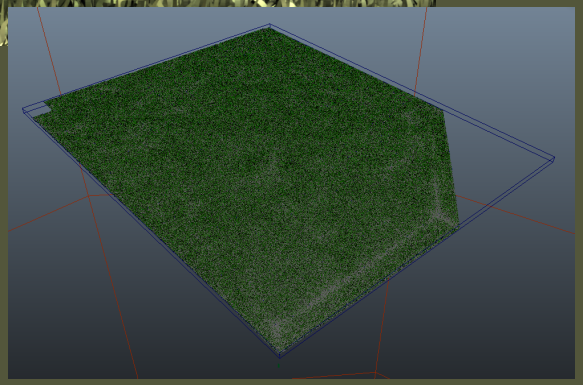
Francesco Giordana
Sarah Macdonald
Gianluca Vatinno

Double Negative VFX

Hair

- Creatures:
  - Digi-doubles hair / facial hair (100k - 150k)
  - Digital creatures fur and feathers (few Ms)
- Environments:
  - Grass, moss, seaweed, etc.. (many Ms)

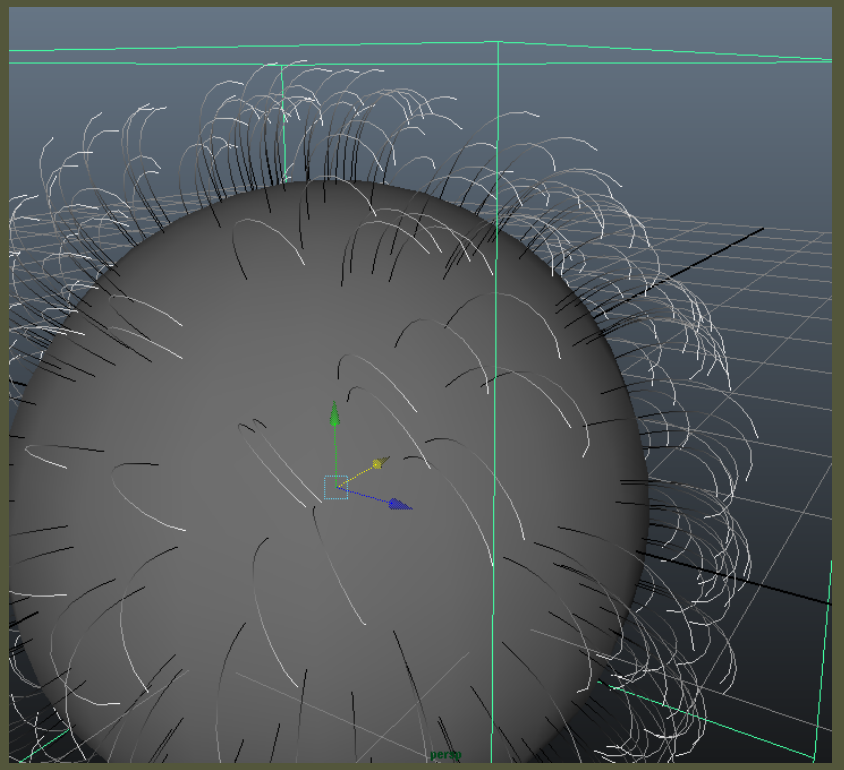




Hair Pros

Why hair on the GPU?

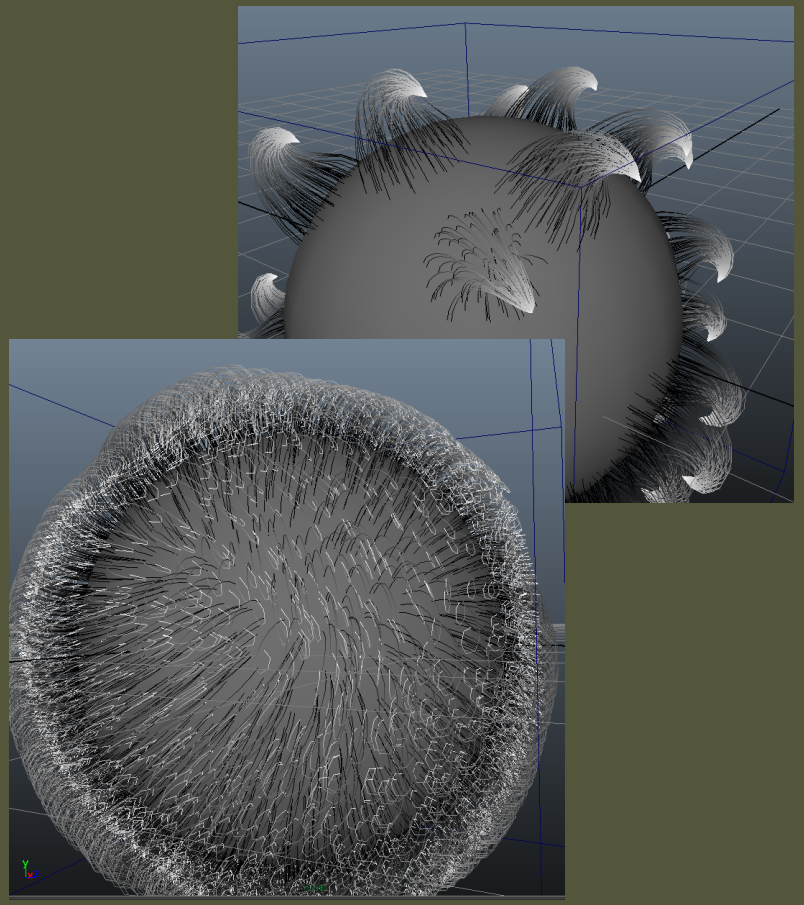
- A lot of repetitions of very similar data
- Each hair can be computed in parallel
- Partially uniform domain
- No need for high precision



Hair Cons

Why NOT hair on the GPU?

- Inter-dependency between hairs
- Walk down the hair to propagate constraints
- Number of curves can change
- Arbitrary spatial extension



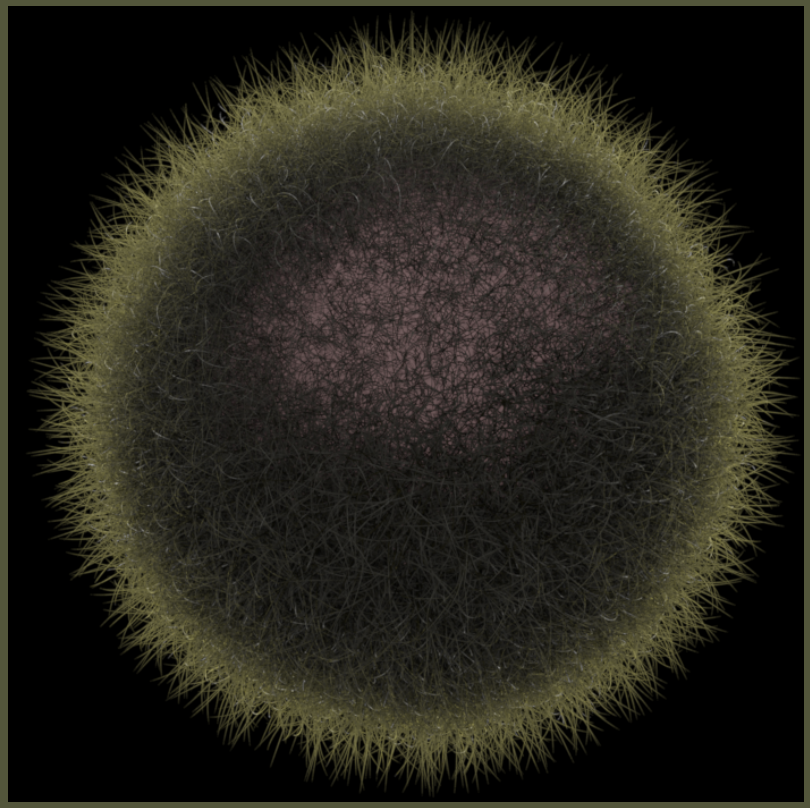
What language / library ?

Thrust:

- Fast and easy to use: **STL**-style containers and algorithms
- Has lots of fancy iterators to keep code cleaner
- Can handle host code: makes code reuse easier
- **CUDA** backend quite optimized (sorts out automatically grid size, block size, shared mem usage)
- Has CPU backends (**TBB** and **OpenMP**)
- Limitation: no streams, no manual control of shared mem
- We quickly prototype in **thrust**, then if needed we optimize writing specific **CUDA** kernels

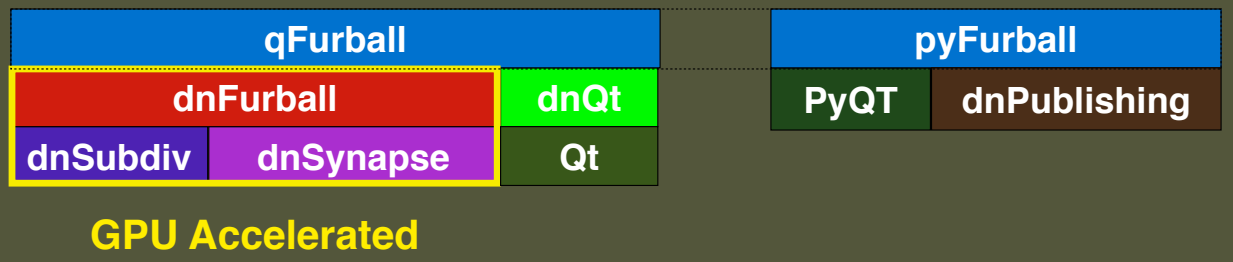
Furball

- Procedural node-graph
- Custom node graph editor
- Embedded in existing 3D sw packages (Maya, Houdini, ...)
- High-quality previews in viewport
- Modular
- C++ Core
- Qt / PyQT UI



Historical first render with Furball

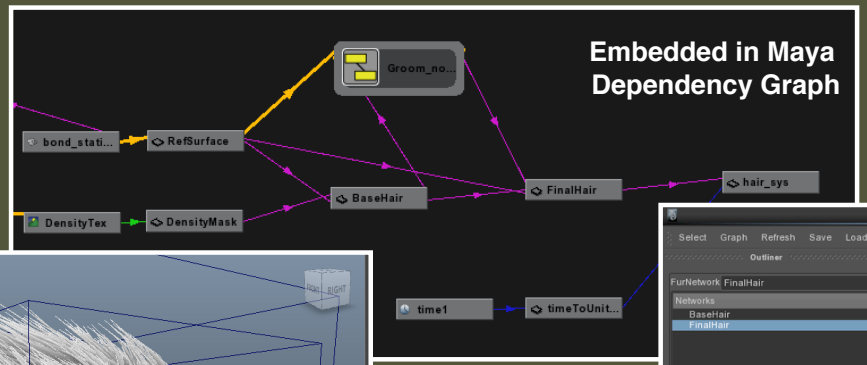
Furball Framework

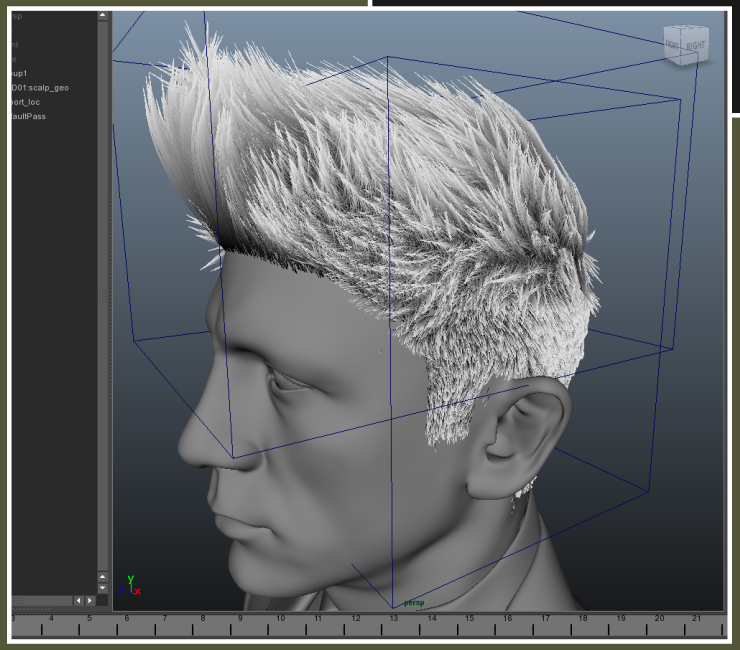


Tools integration

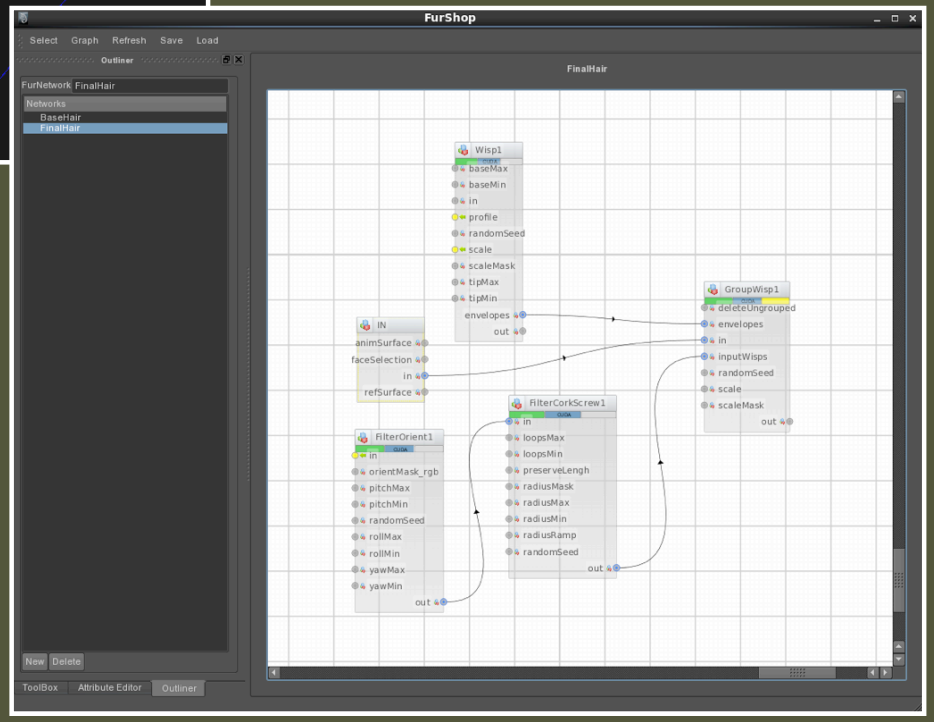
Maya
PRMan
Houdini

FurShop - Maya Integration



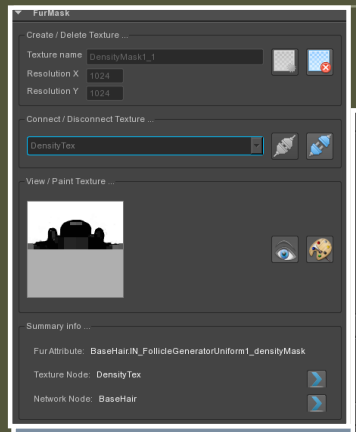


Real-time preview in Maya viewport

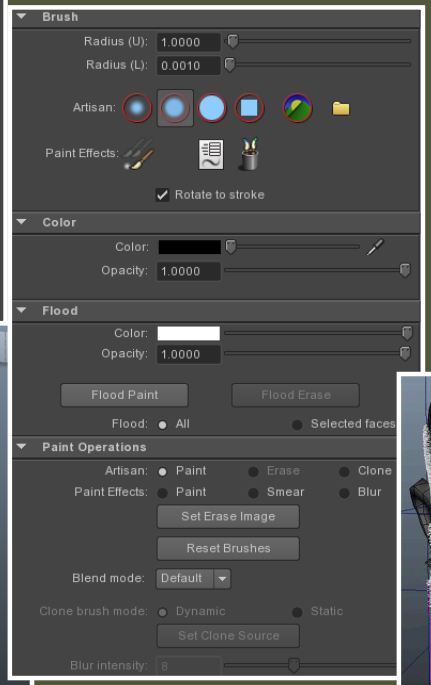


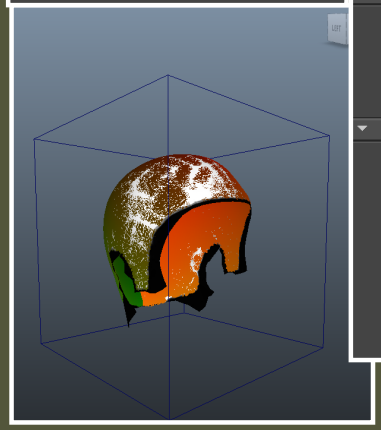
Custom Graph Editor

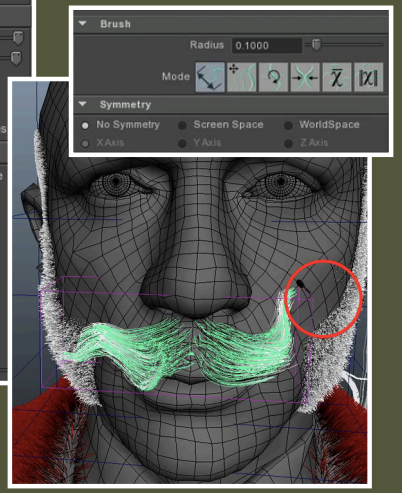
FurShop - Tools



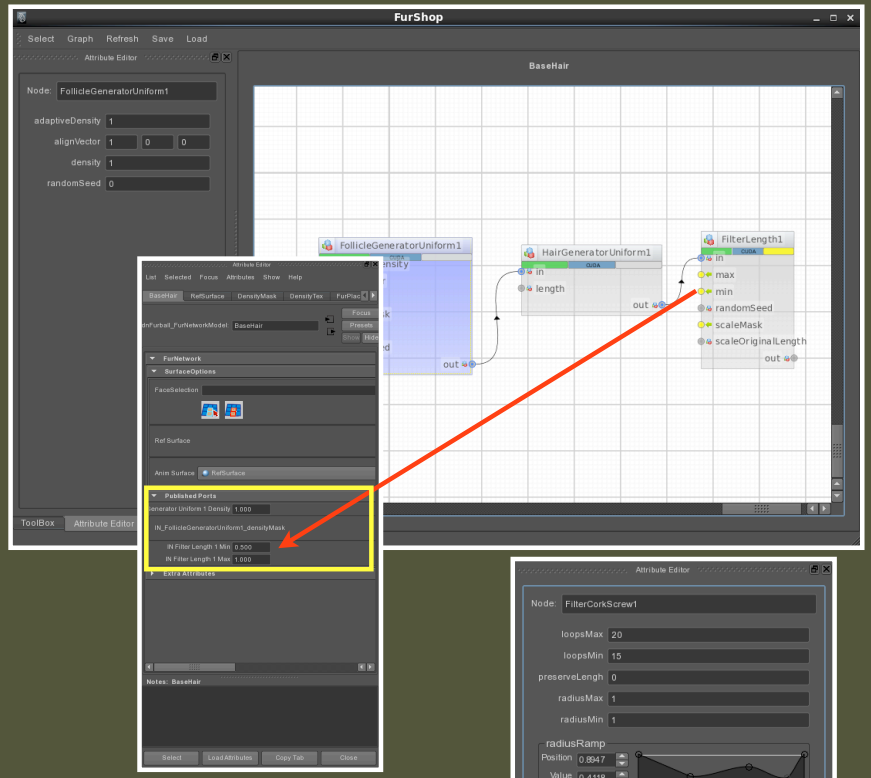
Mask painting tool



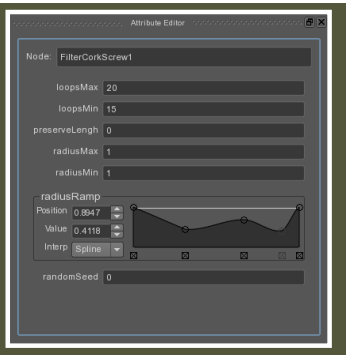




Interactive brush tool



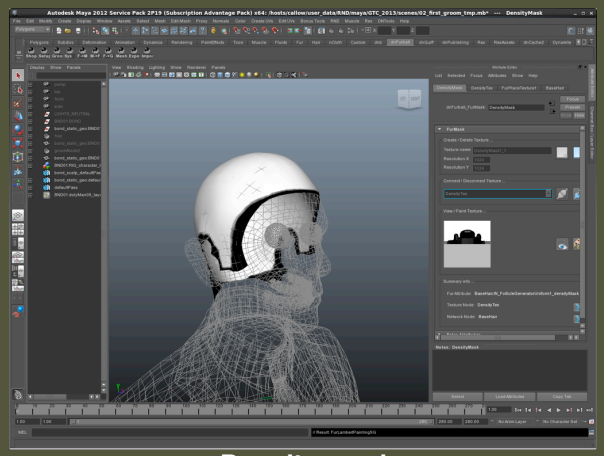
Attribute publishing



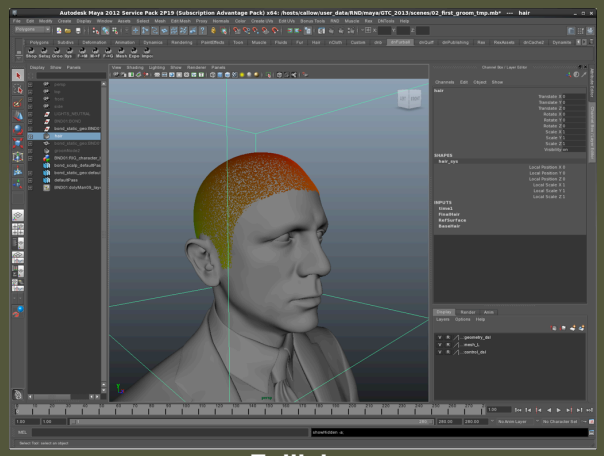
Custom UI elements

FurShop - Example Workflow

Static geometry



Density mask



Follicles

Hairs

Guides

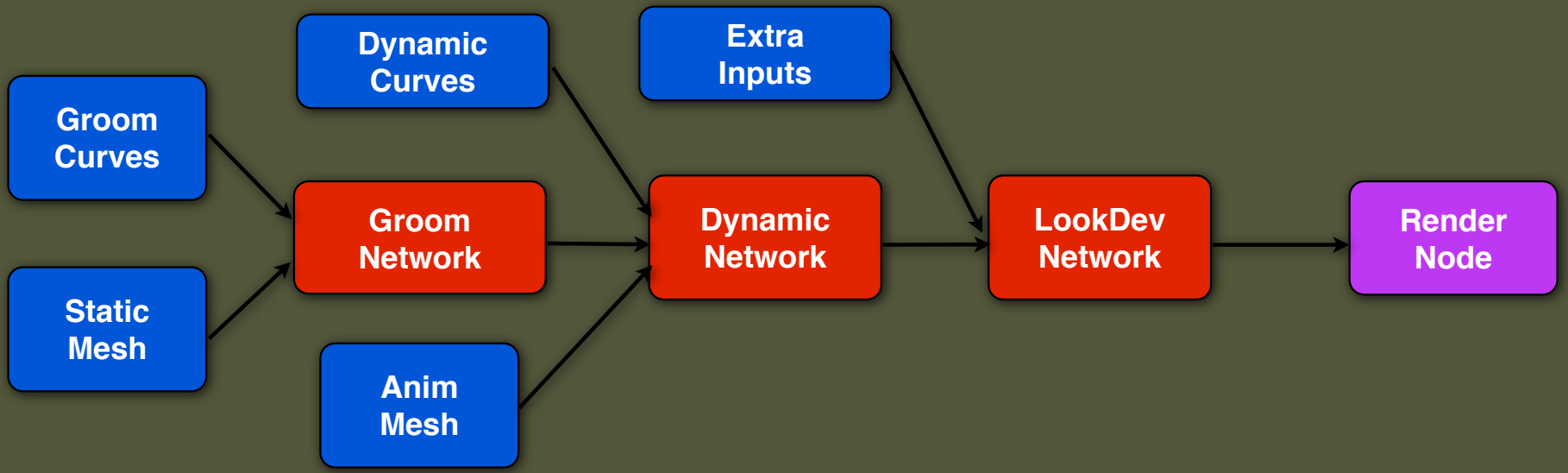
Final

FurShop - Example workflow

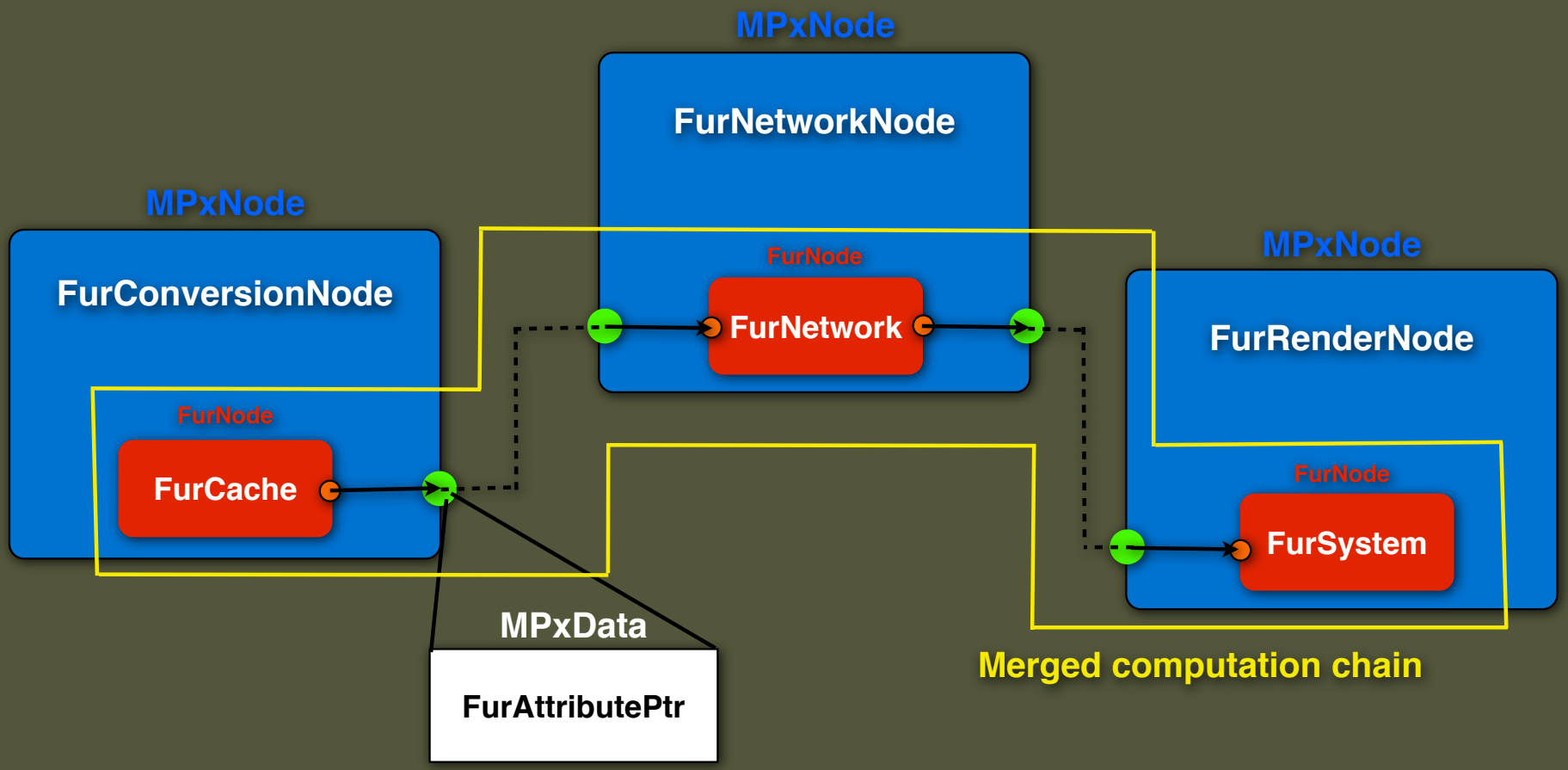
Blue:
- external inputs (maya curves, houdini simulation data, curve manipulation tools, etc)
- stored in data caches

Red:
- procedural networks
- GPU accelerated elements

Purple:
- rendering environment (PRMan DSO, OpenGL, etc.)

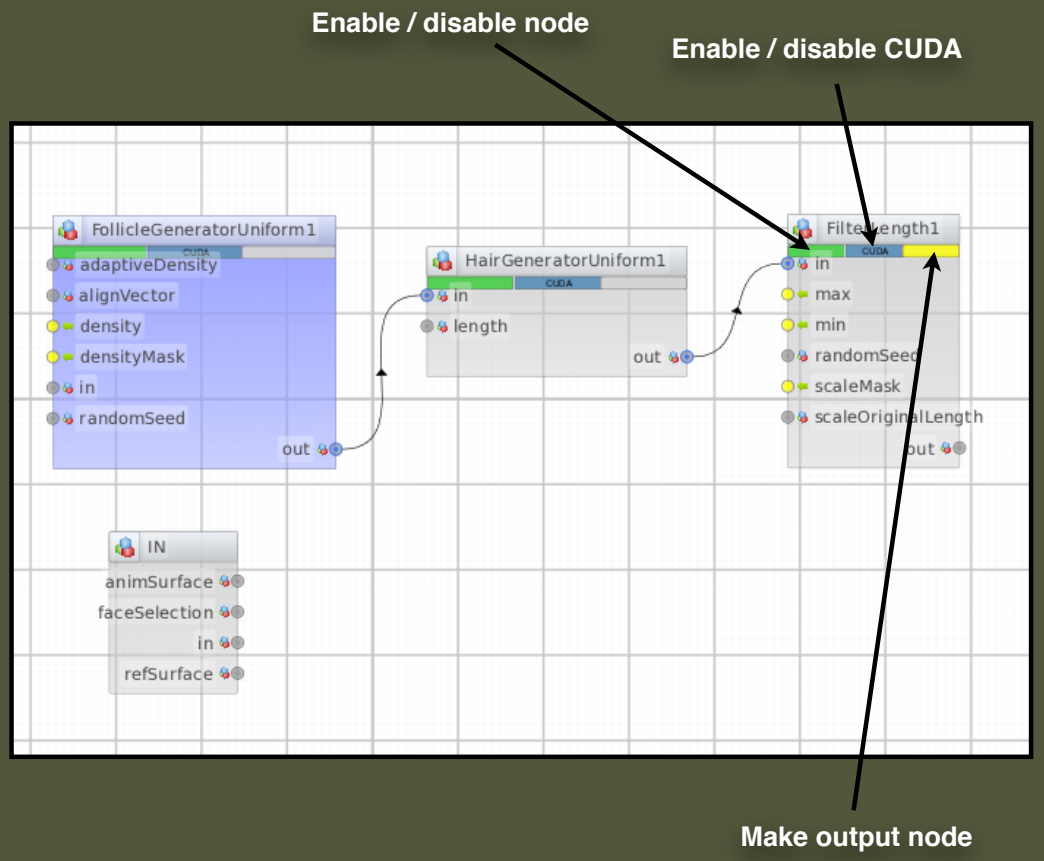


FurShop - Maya Nodes



dnSynapse

- DAG with lazy-pull computation model
- Two types of objects: Node, Attribute
- Data flow through Attributes
- Nodes for computation
- SubGraphs: nodes can contain an entire graph inside
- Proxy attributes: attributes from subgraph can be exposed to the upper layer



dnSynapse - Device Controller

- Initialize and select device
- Create CUDA Context
- Handle resources (e.g. available memory)
- Enable / disable GPU acceleration

```
struct DeviceController
{
   void enableGPU( bool enable );
   void isEnabledGPU();
   void selectBestDevice();
   bool canHandle( const DataGPU* data );
}
```

dnSynapse - Dual Data

- Abstract Data wrapper with interface exposed to user
- Two separate implementations for CPU and GPU
- Data conversion triggered with **getDataGPU()** or **getDataCPU()**

```
struct Data
{
  DataCPU* dataCPU;
  DataGPU* dataGPU;

  void clear();
  ...

  DataCPU* getDataCPU();
  DataGPU* getDataGPU();
};
```

```
struct DataCPU
{
    thrust::host_vector<...> ...;
    void clear();
    void save( char* filename );
    void load( char* filename );
};
```

```
struct DataGPU
{
  thrust::device_vector<...>  ...;
  void clear();
  void copyTo( DataCPU* dst );
  void copyFrom( const
  DataCPU* src );
};
```

dnSynapse - Dual Nodes

- Nodes have a **CPU compute** and a **GPU compute** (optional)
- Try **GPU compute** first, fallback to **CPU compute**
- At first **GPU compute** data is transferred to Device
- Data will stay on Device until the next **CPU compute**
- Can enable / disable GPU computation with flags (for debugging)

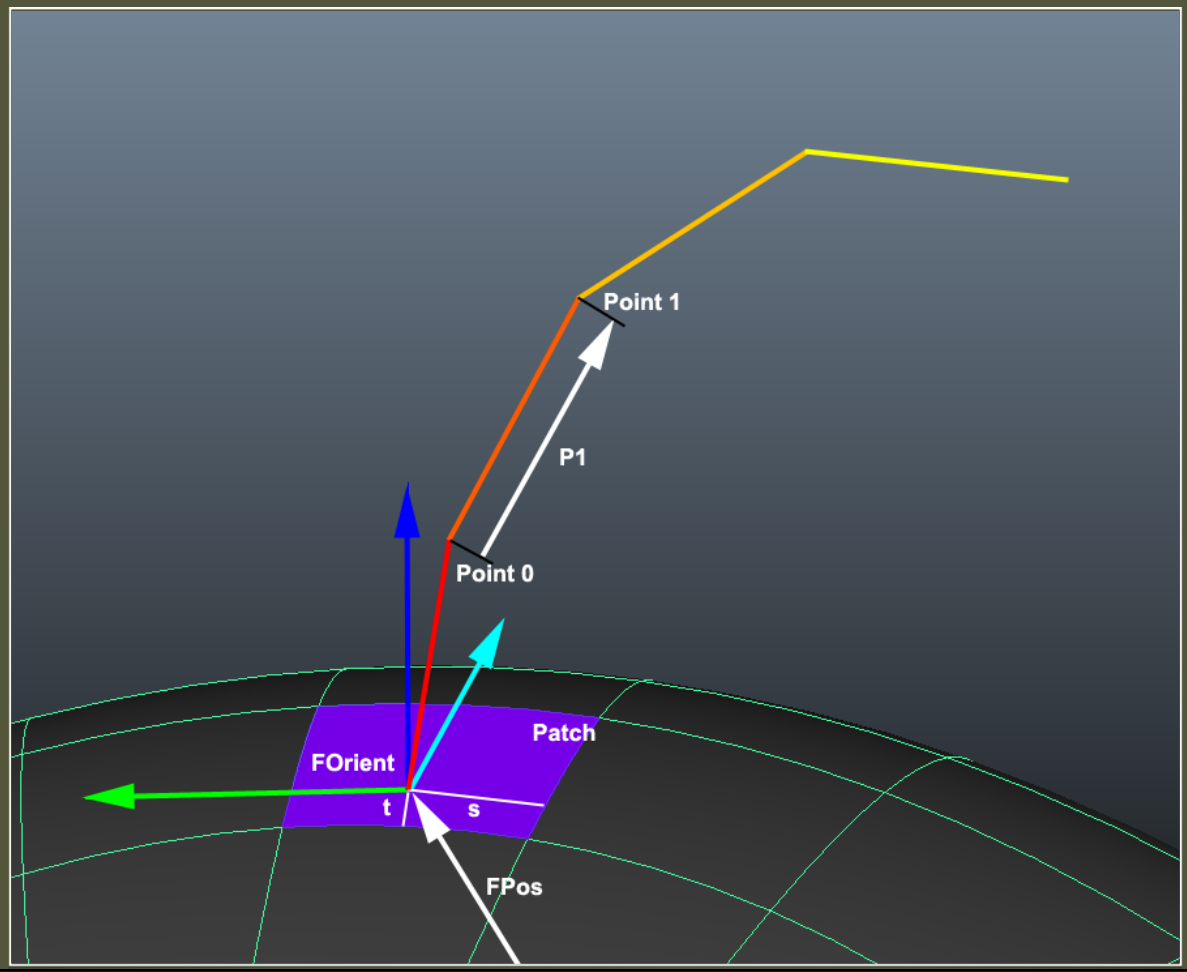
```
void compute( Data* outData, Context* inContext )
{
  bool result_CUDA = false;

  if ( cudaEnabled() && canUseCUDA() )
    result_CUDA  = computeCUDA( outData->getDataGPU(), context );

  if ( ! result_CUDA )
    computeCPU( outData->getDataCPU(), context );
}
```

Furball - Hair

- Follicles
  - Surface Patch ID
  - Surface Patch ST
  - Surface Reference Orient
  - Follicle Position
  - Follicle Orient
  - Follicle Reference Position
  - Follicle Reference Orient
  - Follicle UV
- Curves
  - n Curve Points

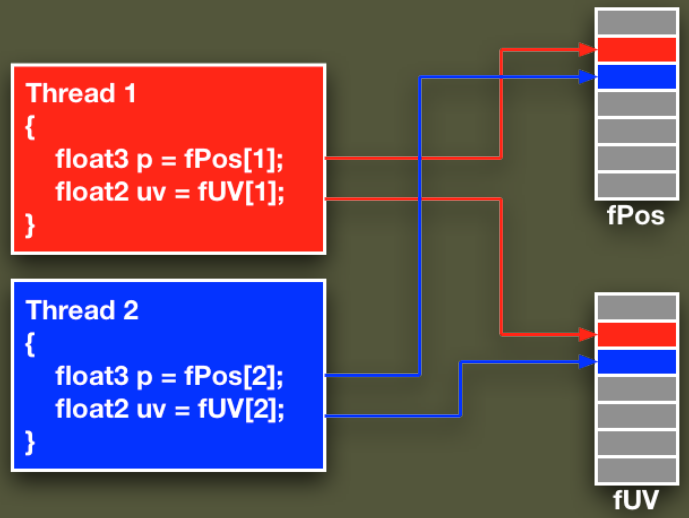


Main families of operators

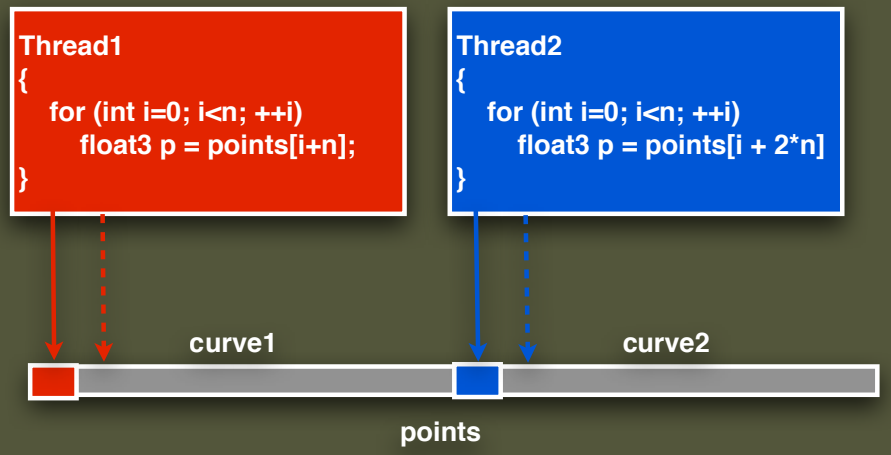
- **Per-point:**
  - Each point in a separate thread
  - No need for info about neighbors
  - Example: scale
- **Per-curve:**
  - Compute a whole curve in a single thread
  - Accumulate constraints walking along the curve
  - Example: curl
- **One-curve-to-many:**
  - Relationships between one curve and a set of curves
  - Per-curve kernel with information about neighbors
  - Example: guide interpolation
- **Many-curves-to-many:**
  - Potentially constraints between all curves in a set
  - Example: hair-hair collisions

Memory Layout

- Follicles sorted per-patch
- Curves sorted per patch, same order as follicles
- Curve points are ordered per curve, root to tip
- Each attribute to separate compact array
- Can split components to separate arrays to maximize memory access efficiency
- 1 million curves, 32 segments: **36ms** on per-point operator, **96ms** on per-curve operator



Coalesced access



Uncoalesced access

Caching

- **Problem**: Caching occupies memory resources
- Must cache on Host: Need transfer H->D when reading cache (slow)
- Can't use too much pinned memory, or system performance will degrade
- **Solution**: cache follicles
- Limited data set: no curve points
- Can build kdtree and cache it along
- Transfer of follicles data is quick, smaller data set so we can use pinned memory
- Recompute hairs on the device

- 1 million curves, 32 segments per curve:
  - **Follicles and hairs on host, non-pinned memory,** Size: **420MB**, H->D:**120ms**
  - **Follicles on host, hairs on device, pinned,** Size: **50MB**, H->D: **10ms**, Hair Generation: **14ms**

Test Computer

Mirrors current artists' computers:

Xeon X5690 @ 3.47 GHz
6 Cores
48 GB RAM
Quadro 4000

CPU - Single threaded using STL containers
CUDA - compute 2.0, using thrust

Soon to test multi-threaded CPU and CUDA on Tesla K20

Filter Frizz

Inputs:

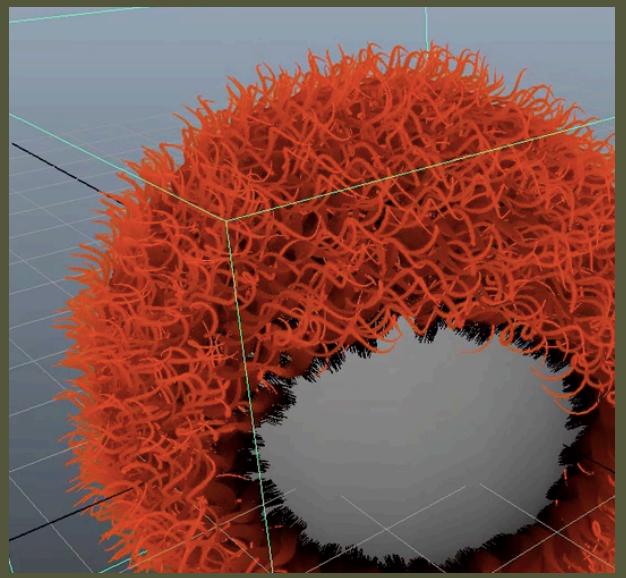
Hairs
Ramp
Mask
Randomization

Steps:

1) Generate random sequence per-hair
2) Generate random sequence per-point
3) Apply random displacement to each curve point
4) Weigh the effect of the frizz by mask value, ramp value and random sequences

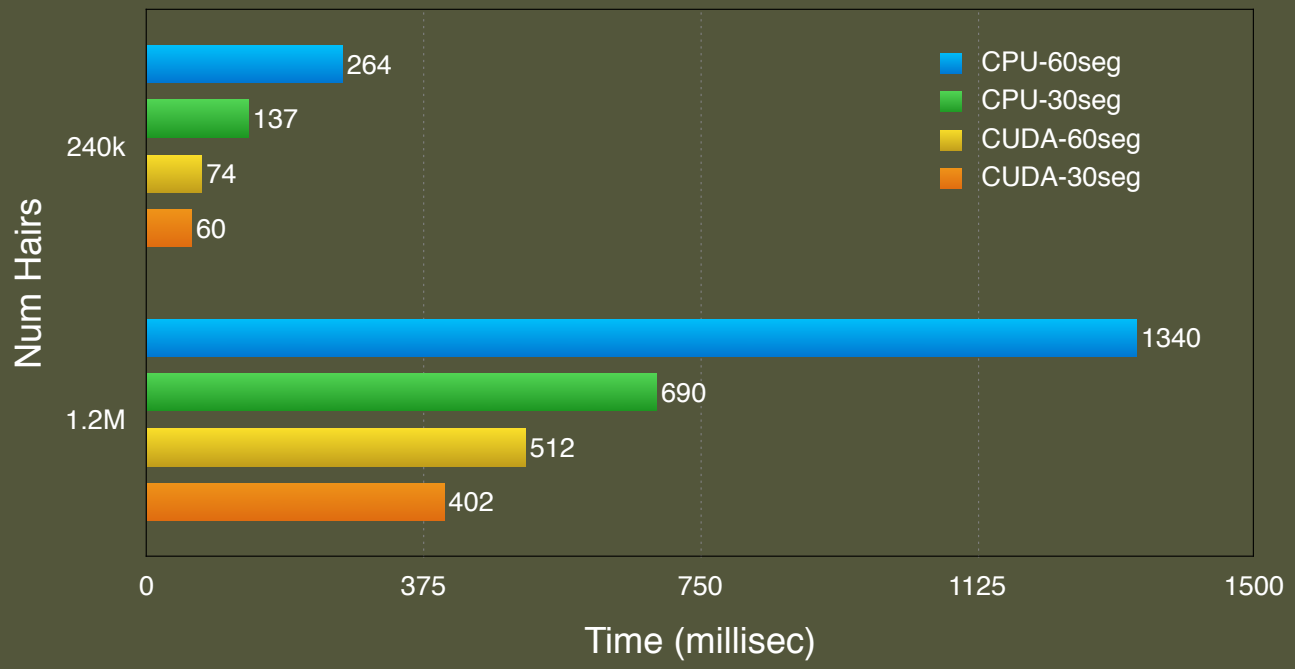
Improvement:

- Combine mask and random values per-curve before launching main kernel
- Reduce texture accesses from (numSegments x numCurves) to numCurves
- 10% performance gain

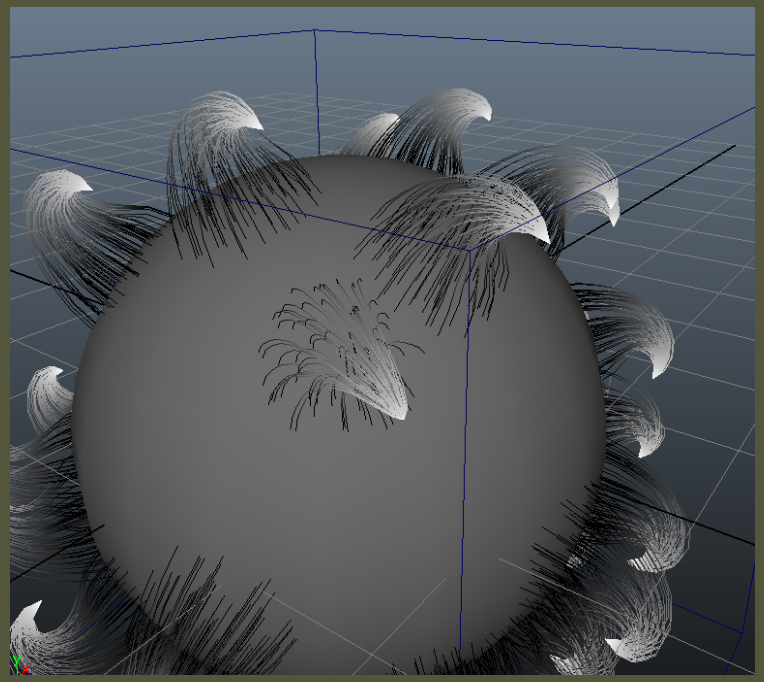


FilterFrizz

- Total 4-5x speedup
- More data -> more performance gain



Wisps



Inputs:

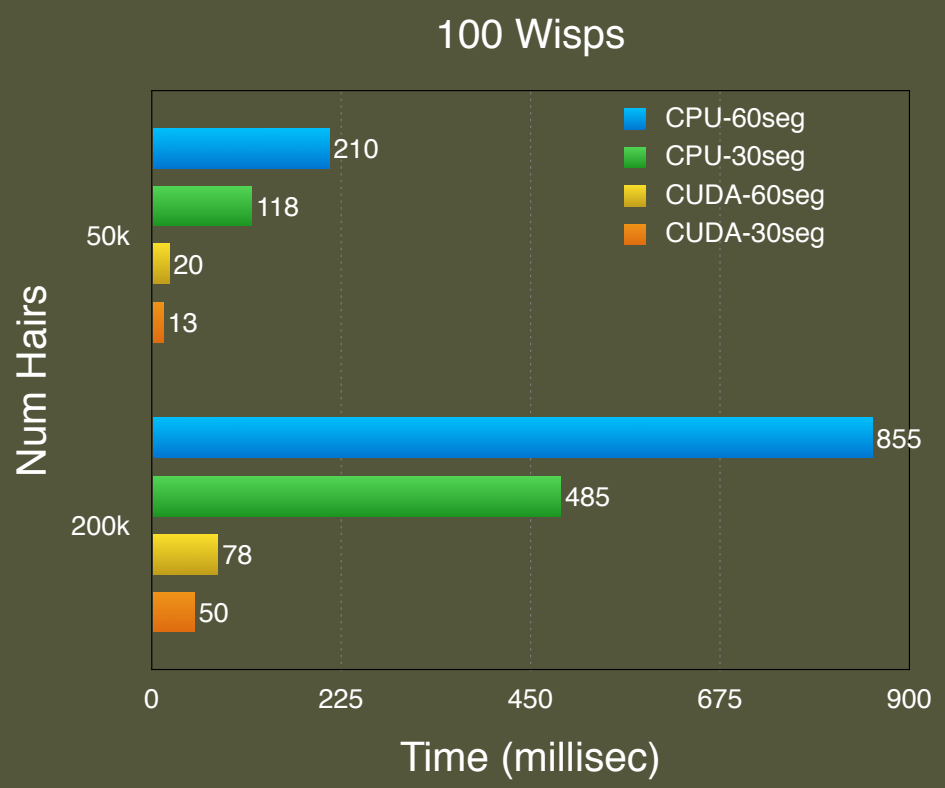
- Hairs
- Wisps center curves
- Envelope profiles
- Masks
- Randomization

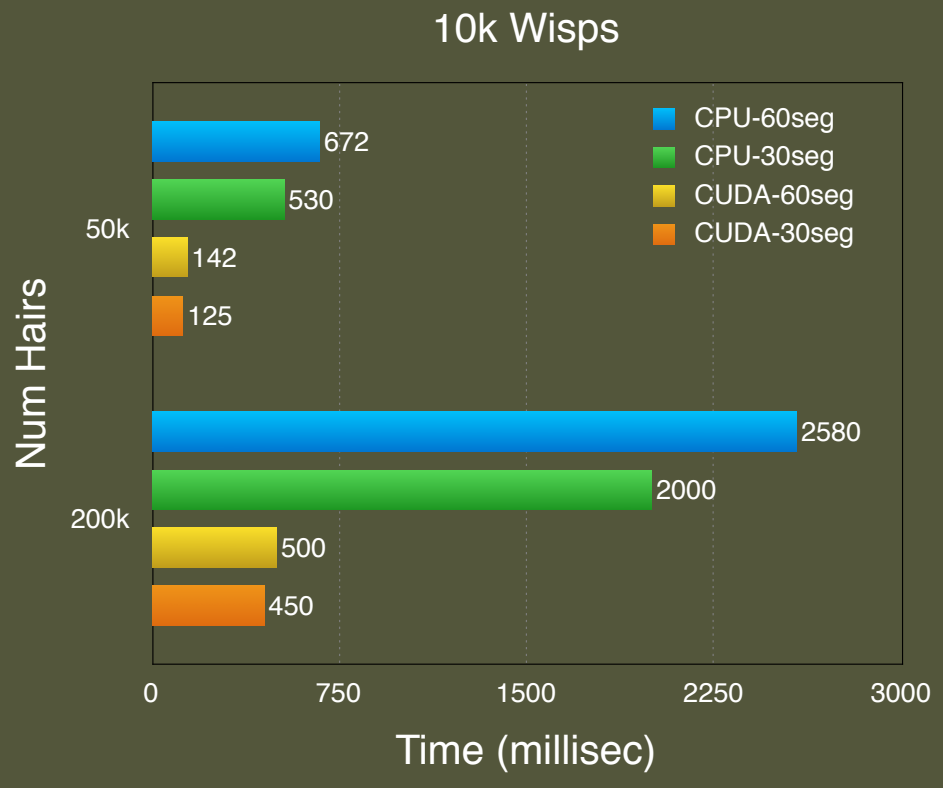
Steps:

1) Generate envelope for each wisp
2) Distance computation hair follicle - wisp root
3) Randomly pick one of the overlapping wisps for each hair
4) Parallel transport of distance vector along the curve
5) Rescale vector so that it fits the envelope

Wisps

CPU kdtree - CUDA brute force
10x speedup





Future Work

- Try Kepler cards
- Multiple streams
- Multi-threaded CPU
- Compile portions of graph
- Kernel fusion
- GPU k-d trees
- Try CPU backends (OpenMP, TBB)
- Include dynamics simulation system inside Furball

Questions?

Francesco Giordana - fg@dneg.com
Sarah Mcdonald - svm@dneg.com
Gianluca Vatinno - gv@dneg.com