Puppeteering the Pixels

Innovative Animation Tools For Stylized Rendering

Lee Kerley
lkerley@imageworks.com
Sony Pictures Imageworks
USA

Joshua Beveridge
joshuab@imageworks.com
Sony Pictures Imageworks
USA

Dylan Reid
dreid@imageworks.com
Sony Pictures Imageworks
Canada

Dylan Gottlieb
dgottlieb@imageworks.com
Sony Pictures Imageworks
USA

David Allen*
effects.animator@gmail.com
Sony Pictures Imageworks
Canada

Stefan Herz
sherz@imageworks.com
Sony Pictures Imageworks
USA

Chris Kulla†
ckulla@gmail.com
Sony Pictures Imageworks
USA

Stirling Duguid
stirling@imageworks.com
Sony Pictures Imageworks
Canada

Doug Smith
dougs@imageworks.com
Sony Pictures Imageworks
USA

Arjun Namdeo
anamdeo@imageworks.com
Sony Pictures Imageworks
Canada

Mike Cole‡
mike.cole17387@gmail.com
Sony Pictures Imageworks
Canada

Figure 1: Examples of *CreaseLines* and *Variable Motion Blur* in action. ©NETFLIX ©2022 CTMG, Inc. All Rights Reserved

*Author now works at Double Negative Animation.
†Author now works at Epic Games.
‡Author now works at Walt Disney Animation Studios.

DigiPro '22, August 7, 2022, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9418-5/22/08…$15.00
https://doi.org/10.1145/3543664.3543673

ABSTRACT

This talk presents two techniques where novel animation tools are used to influence the final render in unusual ways that have been implemented in the Sony Pictures Imageworks pipeline. The first technique, *CreaseLines*, provides animators the ability to dynamically create and control curves to define emotive facial creases that directly drive displacement of the face meshes. *CreaseLines* wide range of control opened new opportunities for animators to quickly hit performances in a very direct way. The second technique, *Variable Motion Blur*, gives the animators more direct control over the motion blur in the render, allowing them to easily direct the audience's attention and highlight the important action. Spheres define regions in the scene where motion blur amplitude can be scaled.

CCS CONCEPTS

• **Computing methodologies** → **Animation**; **Rendering**.

KEYWORDS

animation, pipeline, rendering

ACM Reference Format:

Lee Kerley, Joshua Beveridge, Dylan Reid, Dylan Gottlieb, David Allen, Stefan Herz, Chris Kulla, Stirling Duguid, Doug Smith, Arjun Namdeo, and Mike Cole. 2022. Puppeteering the Pixels: Innovative Animation Tools For Stylized Rendering. In *The Digital Production Symposium (DigiPro '22), August 7, 2022, Vancouver, BC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3543664.3543673

1 INTRODUCTION

Traditionally animators move vertices on meshes. This often requires us to build complex systems of rigs, deformers, IK/FK, and all sorts of magic, but when it boils down to it, the output from animation is usually just a pile of new point locations per frame that can be applied to a mesh. At Sony Pictures Imageworks we have recently broken this basic assumption in an attempt to solve some interesting problems: Giving animators the freedom to define new geometry as they animate, and interpreting that new geometry in novel ways to influence the renderer as it is rendering.

CreaseLines was developed for the movie *The Sea Beast*. Animators can create and animate curves to define facial creases in a topologically independent way, without the use of painted textures. This tool provides the animator a more direct way to achieve the performance they desire, and removes the constraints of previous techniques to solve this problem. These curves are dynamically combined with the facial mesh, and a shader reads the curves and attributes defined by the animator to perform displacement at render time.

Variable Motion Blur was developed for the movie *Hotel Transylvania 4: Transformania*. The Hotel Transylvania series has always had a very dynamic animation style, and previously it has required lots of manual work to hit some of the performance notes. This tool allows animators to use simple primitives in their scene to control the amount of motion blur applied to geometry that overlaps those primitives with a controllable falloff. This is a simple idea, but a very powerful one, as it again puts direct control of the performance into the hands of animators, and provides that control in a fast and intuitive form for them to understand. While the idea behind this tool is a simple one, its implementation involved many departments in our pipeline and several revisions until we found the sweet spot of control, ease of use, and efficiency.

2 CREASELINES

2.1 Background

As we began looking at the initial character designs for *The Sea Beast*, we noticed that many of them had very distinct wrinkles. Captain Crow specifically was designed with several deep, asymmetrical creases that crisscrossed against each other.

Our typical solutions for tackling wrinkles would be to (A) model in the detail which makes the models heavier, the rigs more complex and constrains the animator's freedom; or (B) to use textures, which could have animated displacement depths, but shaping controls are hard and very indirect for the animator.

Our Head of Animation, Joshua Beveridge, had previously worked with the *Inklines* on *Spider-Man: Into the Spider-Verse* [Dimian et al. 2019], and found the flexibility and expressive control extraordinarily useful in conveying emotion. Additionally the designs on *The Sea Beast* had the wrinkles at angles that would have seriously hindered the flow of our topology, and would require a density that would rival a VFX digital double as opposed to a CG feature character.

So the desire for expressiveness, and the need to keep the density of the mesh at a reasonable resolution for speed, led him to the idea of altering the *Inklines* to be able to actually displace the geometry at render time to create the wrinkles.

2.2 Goal

Inspired by the *Inkline* rig work that was done for *Spider-Man: Into the Spider-Verse*, we wanted to create a highly dynamic system for creating and animating facial creases. The system needed to be dynamic enough to not place restrictions on how later parts of the pipeline would construct their data, or require a high level of manual interaction. It was also obvious at the start that because this was an integral part of the characters performance, the animators would need very good interactive visualization to work from, as well as easy access to a highly accurate preview render for QC/approval.

2.3 Animation Tools

A set of Maya NURBS curves were created and aligned to landmarks on the face, such as along the nasolabial fold, around the eye or across the forehead. This set was realigned for each character, with special care to make sure the control vertices (CVs) were running in the same direction for consistency, and published as a model component, separate from the main mesh model component.

The *CreaseLine* rig essentially worked like a simple Bezier. Each curve had 7 CVs, with a tangent control in the middle (3 CVs), and one at each end (2 CVs each). The tangents could be translated, rotated and scaled to shape the NURBS curve as desired. The center tangent also allowed the tangent to be 'broken' to create sharper bends.

Ordinarily a spline based rig like this would be created straight, and then posed into position. For example, it's much easier to rig a tail straight and then curl it up, than to rig it curled and try and straighten it. However these pre-posed input curves were relatively

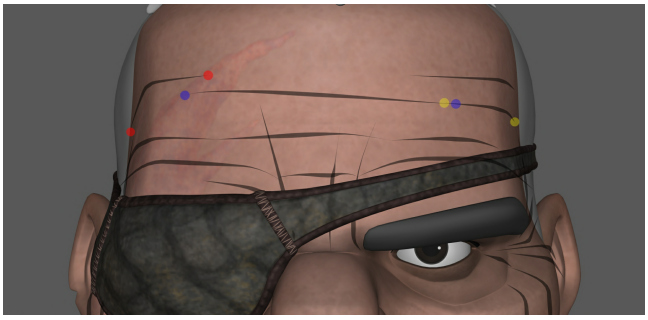

Figure 2: Captain Crow animation curve controls. ©NETFLIX

simple, so the *CreaseLine* rigs were dynamically constructed around them. The middle CV and the end CVs provide the control positions, and form a triangle in space, which creates a plane of rotation. We then calculated the tangent rotation values required to approximate the shape of the input curve. It's not possible to perfectly match every input curve, as the rig has limits on shaping that direct modeling does not, but it was close enough and meant we could quickly iterate and add new curves as needed.

The rigs were also designed to automatically follow the face, adjusting the control positions and orientations to precisely track the underlying deformation. Animation could then tweak and animate on top of this motion. They also had the option to just follow the general movement without changing the curve shape, or turn it off entirely. This tracking saved animation a lot of time, especially on characters where the CreaseLines were used to define static fine lines, rather than dynamic wrinkles.

Attached to the NURBS curve was a Maya Paint Effects stroke, rigged to mimic the shape of the final displacement and provide a visual representation to animation.

The stroke thickness could be adjusted at the start and end of the stroke (min/max), as well as the center. Initially it was thought that having some thickness at the ends would allow for some interesting displacement effects, but it quickly became clear that it was better to lock the min/max thickness to zero so the displacement could taper off smoothly to nothing. The shape of the tapering was also controllable to give either a sharp or blunted shape at the tips.

The tips and center, belly, of the stroke could slide along the curve. This made the animation process smoother. Instead of having to animate the tangent control positions to adjust the stroke length over time, the tangents could remain static and simple slide values adjusted instead.

Finally, the opacity of the stroke was used to define the depth. If the opacity was low it would mean the displacement was shallow. If the opacity was high, the displacement would be deep. The stroke color was black by default, so any transparency would combine the stroke color with the color of the face texture behind it, giving the illusion of a shadow cast by a wrinkle.

Unlike *Spider-Man: Into the Spider-Verse*, where the *Inkline* strokes were converted to a mesh and rendered, these strokes were only used for visualization in Maya. Lookdev only required the underlying curve shape, and the various thickness and positional information along it. Sony Pictures Imageworks' animation publishing system allows for specifically named attributes to be cached out per frame and travel through the pipeline along with the geometry. The rig control values were converted into parameterized values via an internal node network, and output as meaningful lookdev data to these special, cacheable, attributes. Some iteration was required to allow the visualization to accurately reflect the lookdev interpretation of the data, but the result was animation could animate the lookdev data indirectly in an intuitive and user friendly way.

The *CreaseLines* system was also used for other facial features, such as dimples. Additionally each character also had a set of 'extra' *CreaseLine* rigs that were straight by default, and not attached to the geometry. They could be turned on and used to add extra details as needed.

For Captain Crow we created a number of extra long creases with multiple tangent controls. He had wrinkles running all the way across his forehead so the standard *CreaseLine* rigs/curves were too simple to create the desired shape. We could not build this longer rig automatically around an input curve, so it was built straight, posed manually, and the pose values baked into the face rig. Additional lookdev attributes were added to output the correct parameterized data.

2.4 Curve Evaluation in OSL

Creases were expressed as control points of quadratic b-splines (the default for Paint Effects strokes in Maya). An OSL shader was written to measure the distance from the current shading point to the curve and displace it according to an artist-defined profile, both along the length of the curve and across its width. Distance fields to quadratic splines happen to have a closed form solution which was used here to avoid the need to discretize the curves into linear segments. Because the curves were placed by animators in relation to a particular camera view, all point to curve calculations were done in screen space, such that the curve position precisely aligned with the stroke the animators could visualize in Maya. A first shader would output both the distance and parameter value of the closest point along the curve, allowing easy visualization of the generated coordinates and for subsequent patterns to define the actual profile of the curve.

Given that the shader was driving displacement, we found it was critically important that all profiles be driven with expressions with C2 continuity, to avoid discontinuities in the surface normal. We ended up deriving the following polynomial where the user-defined value s controls the slope at $x = 0$, with the slope smoothly changing to 0 at $x = 1$, where x is the distance from the shading point to the curve.

$$f(x) = (s * x)/(1 + (2 * s - 3) * x + 3 * x^2 - x^3)$$

We considered multiple designs to implement this procedure. At first, we contemplated allowing the renderer's displacement system to make queries of other geometric primitives in the scene (in this case the curves). This seemed like it could form the basis for a very powerful set of features. On the other hand, we were worried about scope creep and potentially having to solve a much more

general problem than we were really faced with for this particular application. One concern in particular with the idea of geometries affecting one another during the displacement stage is that modern production path tracers tend to prefer to prepare geometry upfront, before image generation starts. This process is typically parallelized over all objects, and introducing inter-dependencies in this stage would have greatly complicated the overall software architecture.

Instead, we took a much more pragmatic approach and simply copied the curve's vertex data onto the target mesh as user-data. This retained some of the advantage of the idea of querying geometry directly in that we could rely on the renderer to interpolate the user-data to the appropriate time for motion blur (the mesh and *CreaseLine* curves were guaranteed to be exported at the same set of sub-frames, allowing us to have consistent sub-frame motion between both). The *Sony Pictures Imageworks Arnold* renderer already supported time varying displacement which was important in this case. The displacement shader simply executes at every time-sample the geometry is defined on, allowing the CreaseLines to follow the motion of the skin and for the creases themselves to animate quickly as the animators intended.

The initial implementation was prototyped with a single quadratic curve segment and quickly extended to support longer curves with more vertices (piecewise quadratic) and more distinct strokes (representing multiple wrinkles and expression lines). Animators had control over not only the stroke vertices, but their width. A strength setting could also be animated to selectively deepen or fade certain lines on the face. Where multiple lines overlapped, we expected to need some control over the blending behavior, but in the end found that only a plain max of the creases produced a smooth result.

Another surprising fact was that to make the curve falloff into the skin soft and gradual, the region of influence of the curve had to be fairly wide. This in turn could lead to some surprising results if settings were pushed too far. Successful results were strongly dependent on a well tuned initial setup, but once the strengths were dialed in, we found animators had great freedom to place the lines as they saw fit.

Figure 3: Maisie's dimples were also *CreaseLines*, but the high level of curvature posed a problem. ©NETFLIX

In figure 3 the dimple curve has a high curvature, which causes a discontinuity. We ended up solving this pragmatically by using a super-sampling technique in the shader to blur the crease. While this was more expensive, its use was limited, and didn't dramatically impact production rendertime performance.

The *Sony Pictures Imageworks Arnold* renderer already had a fully functional adaptive subdivision tessellation engine. No modifications were necessary to support *CreaseLines*. The adaptive subdivision tessellation engine uses a heuristic to determine the tessellation rate. Each patch is tessellated until the screen-space projected edge length is below a user-defined minimum edge length threshold, this threshold is defaulted to half a pixel in screen-space.

2.5 CreaseLinesPatternLink

It was important that the animators were free to add or remove creases to the faces as the performance required, so we needed a system in Katana that wasn't constrained to a fixed set of curves, and was able to adjust without direct action from the lighters. We developed an extension to *PatternCreate*, our in-house procedural texturing system, called *PatternLink*. This tool allows for pattern graphs to be collected from a number of different 'source' scenegraph locations, and "linked" together in a chain and applied to a destination material. For *CreaseLines*, each of the curves the animators added had a small pattern graph assigned to it that defined the displacement profile of the curve as we describe above. When *PatternLink* was evaluated inside of Katana, as the scene was rendered, whichever curves existed in the scene were dynamically linked to the face mesh, resulting in a chain of displacement profile nodes attached to the displacement terminal of the face mesh.

The *PatternLink* evaluation was also wrapped up with another deferred Katana process that would copy the curve data and the width and strength attributes authored by the animators. Again, this all needed to be dynamic with the number of curves present, and all the user-data was written to the face mesh with name prefixes matching the names of each of the curves, thus lookups could be made as each curve's pattern nodes were evaluated in displacement.

2.6 User Experience

We tried to keep the animation controls and rigs for the CreaseLines as simple as possible. The primary controls were very similar to the setup we used on *Spider-Man: Into the Spider-Verse* which helped as we had some of the same crew. Also, as *The Sea Beast* characters had more texture compared to *Spider-Man: Into the Spider-Verse*, we needed to be more careful about any accidental sliding over the skin. We added an attribute to blend between constraints that were "sticky" to the exact surface area versus simply constrained to the head.

Considering the difference in the graphic nature of what is seen by the animators in Maya compared to the displaced and lit end result, we set up a QC render for visualization. This was trickier than we expected because the depth of the displaced lines can look very different from one lighting scenario to another. The QC renders also helped with the learning curve as animators were adjusting to the concept that the CreaseLines were more of a sculpting tool than a drawing tool.

For look development there was a lot of exploration on the rendering side, both aesthetically for the different shapes that were achievable and also from a technical perspective. We found it useful to render both the final displacement depth of the creases, and also the projected UVs from the curves on the surface to diagnose problems. The red channel represented the length of the curve, and

Figure 4: Captain Crow animation curve controls. ©NETFLIX

Figure 5: Captain Crow animation curve controls. ©NETFLIX

the green channel was the distance to the curve center – defining the falloff. Once we had worked through that initial phase, the setup was pretty straight forward for each character and we ended up with a handful of different curve profiles for different crease types that were used across the entire show.

For shot work animators were responsible for the position, movement and shape of the CreaseLines. The depth of the CreaseLines was controlled by lighting. In shot lighting we found that it was rare that any dialing was needed. It was occasionally necessary to reduce the amplitude of a wrinkle if it appeared too deep, depending upon the light positions. Fortunately, the animators were able to get a good sense of the end result in their QC renders. It was rarely necessary to go back for an animation tweak of a CreaseLine once the shot was in lighting.

2.7 Limitations and Future Work

To state the obvious, real wrinkles don't slide over skin. So even though we had the flexibility to animate the positions of the CreaseLines within a shot, we found that if the motion was visible to the viewer it would look wrong. We could however, hide the movement to a slightly different position within motion blurred frames. It would also depend on how close-up a shot was to the skin – how much skin texture was visible – that could give away the gag.

Another challenge that we ran into was the fact that when real skin is pushed together forming wrinkles, the flesh on either side of the wrinkle will actually bulge out preserving the volume. Our CreaseLines were only displacing in. It's definitely possible that a solution could be found in a future iteration that would give the impression of the preservation of volume.

We had initially implemented *CreaseLines* on Jacob, a lead character in *The Sea Beast*. As we worked on animation tests discovering his personality, the Director wasn't satisfied with the lack of this preservation of volume with the wrinkles on the forehead. We ended up having to adjust the model and facial rig to execute a more traditional route for this character.

An area that could be developed would be adding more organic and natural variation to the displacement of the wrinkles. The profiles that we landed on for *The Sea Beast* were relatively simple, but worked for the desired aesthetic of the filmmakers.

Almost as soon as *PatternLink* was deployed as part of the *CreaseLines* tool it was picked up for other novel uses in the facility. Non-photoreal lighting tools relied heavily on *PatternCreate* to define the look, and *PatternLink* was an obvious complement to that idea. They behave in a very similar way to light linking, but as opposed to lights, they are just procedural texture generators. We can also imagine other applications for *PatternLink* where the dynamic construction of procedural texturing graphs is important, perhaps allowing the FX department provide one part of a material, say for ocean displacement, and dynamically combining it with a separate section delivered by the look dev artist.

In the end, being free of the gridded topology of the edge loops was liberating for the animators. These sorts of details historically have had to be executed in textures, or they had to be sculpted into incredibly dense meshes that would bog down performance. This technique proved very successful on *The Sea Beast*, and has great potential for future applications.

3 VARIABLE MOTION BLUR

3.1 Background

The Hotel Transylvania series' animation style combines pushed poses and fast motion. To ensure that the character performance does not get lost in motion blur animators would give very detailed notes to lighters. The lighting artists were finding the notes challenging and time consuming. Often the lighting artist needed to render the entire shot and each of its passes with full motion blur, half motion blur and no motion blur. Then in compositing they used each as needed to hit the notes given by animation. Blending all the edges to work took time and rendering all the passes took up a lot of resources. For *Hotel Transylvania 4 : Transformania* we were looking for a way to solve this problem, especially as the production schedule was very compressed.

3.2 Goal

On *Hotel Transylvania 4 : Transformania* we aimed to give animators direct control over the amount of motion blur that gets applied to each character (or part of a character) from frame to frame. So that for example the head and shoulder of a character would stay sharp for a given frame while allowing the body to be affected by motion blur in the expected way.

On the lighting side, the goal was to avoid having to do multiple renders to address motion blur notes as outlined above. The motion blur would already be creatively approved when the shot comes into lighting inventory.

Due to the compressed production schedule, this had to be achieved in a highly automated way and with minimal re-computation when upstream components were modified.

3.3 Pipeline Integration

Initially we considered different approaches to define the areas that should have lessened motion blur. Eg. weight maps that could be painted on top of the final character performance. We ended up creating intuitive, lightweight sphere rigs that the animators could use directly in their scene files. The sphere rigs had six controls to deform its shape, as well as attributes to define the falloff and the amount of influence on the motion blur.

Animation could preview the *Variable Motion Blur* by doing lightweight katana renders. We also created color coded debug renders that showed the amount of motion blur reduction as well as the spheres in wireframe.

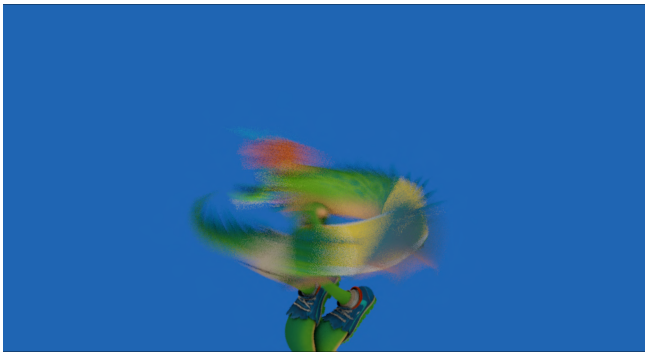

Figure 6: Animation QC render, Johnny Monster with Full Motion Blur. ©2022 CTMG, Inc. All Rights Reserved

Figure 7: Animation QC render, Johnny Monster with Variable Motion Blur. ©2022 CTMG, Inc. All Rights Reserved

When an animation file was processed for lighting, we ran a Houdini post-process that created an OpenVDB volume that stored the information of which areas in 3D space should have reduced motion blur and what the scale factor should be. After that a second Houdini post-process would write out special-purpose velocity vectors for any characters or props that were inside the spheres.

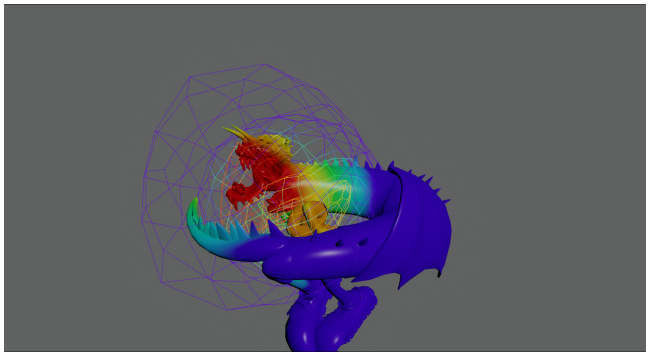

Figure 8: Animation QC render, Johnny Monster with sphere controls, and rainbow render. Red is no motion blur, blue is full motion blur. ©2022 CTMG, Inc. All Rights Reserved

Figure 9: Final render, Johnny Monster with Variable Motion Blur. ©2022 CTMG, Inc. All Rights Reserved

The *Sony Pictures Imageworks Arnold* renderer already knew how to interpret velocity vectors, and acceleration vectors to render motion blur. This technique is often necessary if there is a varying point count of a moving mesh, typical with dynamic geometry often created by an FX simulation. Once we had the modified velocity vectors written to the mesh, we got the variable motion blur we wanted.

One limiting factor was that any character performance or camera change would require re-running the Houdini process that created the scaled velocity vectors. This slowed down the 'Final Layout' process, as even a quick camera update now required a Houdini post-process. A further complication was that character simulations also had to be post processed.

As *Hotel Transylvania 4 : Transformania* had a very short production schedule that required quick iterations from all departments we had to optimize the workflow to reduce cross department dependencies and time spent on the Houdini post-processes. Therefore we deferred the calculation of the scaled velocity vectors until rendering in katana. This allowed animation, final layout and character

simulation departments to adjust the camera and characters without having to re-run any Houdini post-process. The OpenVDB creation post-process was limited to run only when the sphere rigs were added or updated.

AttributeWranglerOSL is a Katana node that is inspired by the AttributeWrangler node in Houdini. It allows artist-written OSL code to be evaluated during scene construction to manipulate attributes inside of Katana. AttributeWranglerOSL was used to read and evaluate the OpenVDB at each point of the mesh, converting the field of motion blur amplitudes into a per-point prim-var on each mesh that intersected the volumes.

A deferred evaluation Katana op was written to compute the final special-purpose velocity vectors. Complete camera descriptions including focal length and aperture information for current and previous frames, point positions of all geometry for the current and previous frames, as well as a per-point attribute representing the desired amount of blur reduction, are read by the op as inputs. Internally, this information is used to compute a vector representing the motion blur the renderer would normally compute, but modified to subtract the blur resulting from frame-to-frame camera changes and geometric point displacements. These calculations are performed in perspective space. We use homogeneous coordinates in order to be able to represent the different types of blur (camera, object, deformation) as a single per-point vector. This vector is then translated from perspective space into world space, and written back to the mesh as a per-point velocity vector.

Switching between the two iterations of the tool-set created its own challenges as we were already in crunch mode, when 'v2' of the *Variable Motion Blur* tool-set was ready to be rolled out. We gradually switched sequences to use the new version of the tool to reduce any impact a newly introduced bug could have on the production and wrote a tool to mass convert shots to the new system.

3.4 Technical Details

From a technical perspective, the approach we took was to provide the renderer with pre-computed per-point velocity vectors representing the object's motion during the frame. Typically, it's the renderer's job to do all the necessary vector math to compute an object's motion. Since we didn't want to modify the *Sony Pictures Imageworks Arnold's* motion blur algorithm, our approach was to generate modified velocity vectors at each point of the geometry being rendered. Providing these modified velocity vectors disables the renderer's own internal motion calculations.

To calculate these modified vectors, there are three types of motion blur that must be considered:

- `Object Blur`: Which is the result of affine transformations of geometry relative to the camera.
- `Deformation Blur`: Which takes into account the motion of each point on the surface of geometry.
- `Camera Blur`: Which is the result of camera motion. This needs to account not only for simple camera translation and rotations, but also for changes in the field of view from one frame to the next (fast zooms for example).

We use homogeneous coordinates to represent previous and current frame's geometry, and their perspective projections. The following is pseudocode which describes the algorithm we used:

```
For each frame of animation:
 For each point in geometry:
  # calculate the regular deformation velocity vector
  deformV = currentP - prevP
  # scale it as requested
  deformV *= FPS * fit(mask, 0, 1, 0, amount)

  # calculate the camera velocity vector
  Pw = (prevP with w component (homogeneous coords))
  cameraP = Pw * invert(camCurrent) * projectionCurrent
  prevCameraP = Pw * invert(camPrev) * projectionPrev
  cameraV = cameraP - prevCameraP
  # scale it as requested
  cameraV *= FPS * fit(1 - mask, 0, 1, 1, amount)

  # finally sum both contributions
  velocityV = deformV + cameraV
```

- `projectionPrev`: Camera projection matrix at previous frame.
- `projectionCurrent`: Camera projection matrix at current-frame.
- `camPrev`: Camera transformation matrix at previous frame.
- `camCurrent`: Camera transformation matrix at current frame.
- `amount`: Multiplier for amount of total effect of blur modification.
- `mask`: Sample pre-computed OpenVDB volume representing strength of effect.
- `currentP`: Current frame's point position.
- `prevP`: Previous frame's point position.
- `modifiedV`: Output velocity vector that should be supplied to the renderer for each point of the geometry.

In practice, it's important that the "mask" attribute, sampled from world-space OpenVDB volumes, has a fairly smooth falloff. Otherwise, the transition from full motion blur to variable motion blur could be too abrupt and look unnatural.

It's important to note that the camera projection is a lossy operation since we work with homogeneous coordinates in perspective space. We found that we needed to do all these calculations at 64 bit floating point precision. Otherwise, noise was introduced into the equation. It's also important to optimize camera clipping planes when performing the camera projection since failing to do so will waste much of the available floating point resolution.

We were able to support hair rendering by making a slight modification to the the way that hair velocity vectors are passed to the renderer. We sample the modified velocity vector at the scalp position and supply that to the renderer to pick up the effect. Please note that we did not calculate a modified velocity vector at each point along the length of each hair. Therefore, the smear resulting from deformation-based velocities of long hair will not be removed in the final render. This was never a problem for us, but could be addressed if needed in the future.

3.5 User Experience

The goal was to make the *Variable Motion Blur* tool implementation as simple as possible for animators to use. We wanted to give them the ability to craft their blur frames but at the same time not bog down the department with an extra step. We achieved this goal by creating a simple UI that gave us full control of the blur. With one click a simple rig would be imported into the scene that was made up of an inner and outer sphere. The inner sphere defines where the motion blur would be reduced, and the distance between the inner and outer sphere defines the falloff. Animators can control three main key-able variables: the scale/shape of the sphere, the falloff, and the amount of blur reduction they would like. The artist could bring as many spheres as required into the scene, which gave us full control to have any part of the body unblurred when needed and let us craft beautiful CG smear frames. This simple solution gave animation all the control needed to get the results we wanted, and best of all it kept our animation files light.

3.6 Limitations and Future Work

We would like to eventually support separating out the control of motion blur that results from camera motion versus blur that results from deformations. This way, an animator could specify parts of the model that should remain sharp when deforming, but still blur from fast camera motion (or vice versa).

We have yet to test the system with volumes. From a technical perspective, voxel-based velocities could be calculated in the same way that we do for surfaces, and this may work fine. However, the way that voxel velocities are interpolated even under normal circumstances sometimes present artifacts, so further investigation is needed.

We would also like to improve the algorithm to support acceleration-based motion blur. However, this may introduce complications in the pipeline, since acceleration vectors are not always available for each data type, and may not be well defined for cameras (for example for zoom/focal length changes).

One of the biggest limitations with the system in its current form is that in order for the animators to see the *Variable Motion Blur* they had crafted, they have to submit a render job to the queue and wait for it to finish. This leads to an iteration feedback loop that is longer than we would desire. We would like to give animation the ability to preview the animated motion blur within Maya, likely by using a Hydra delegate for the *Sony Pictures Imageworks Arnold* renderer. This would make adjusting the motion blur and creating iterations for animation supervisor or client reviews much faster.

4 CONCLUSIONS

This paper has presented two novel solutions to problems that previously would have required a lot more human intervention. By putting animators in direct control of artifacts that control the rendered frame in a very direct way, we have found more efficient solutions to both problems. Both solutions allow users to achieve a higher quality final image. The mesh resolution independent solution for *CreaseLines* gives the animators ultimate freedom to place the facial creases anywhere they need, and the *Variable Motion Blur* tool now allows animators full control to retain sharpness of image in areas that would have previously been lost.

ACKNOWLEDGMENTS

The authors would like to thank the leadership team at Sony Pictures Imageworks for the fantastic projects to work on, the production staff and supervisors for allowing us the opportunity to innovate and believing in us that we could pull it off. Lastly and most importantly we want to thank all the talented artists at Sony Pictures Imageworks without whom this would all be for nothing. Credit: The Sea Beast / Courtesy of Netflix.

REFERENCES

Danny Dimian, Joshua Beveridge, Bret St. Clair, and Geeta Basantani. 2019. Swing into Another Dimension: The Making of "Spider-Man: Into the Spider-Verse". In *ACM SIGGRAPH 2019 Production Sessions* (Los Angeles, California) *(SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/3292423.3313758