Cinematic Animation Effects: Vertex Interpolation, Artistic Rendering, and Post-Processing Filters

LISA DURAND, University of California, Santa Cruz
JENNIFER FULLERTON, University of California, Santa Cruz
JOSHUA NAVARRO, University of California, Santa Cruz

KEYWORDS:
Keyframe, Interpolation, Image Processing, Post-processing, Video Processing, Phong Illumination

1 INTRODUCTION:

Lisa desired to use Three.js as a tool to create complex, nonlinear movements and animations that strayed from standard vertex transformations(scaling, rotating, and translating) and vertex displacements. She wanted to explore animating keyframes and tweening in code. While finding different solutions to this goal, she discovered grouping meshes and using a hierarchy to create full figures with independently operating parts. She also researched splines and curves to move objects along a path. After investigating several ways to animate, she chose to experiment with Three.js's Animation System.

Jennifer's goal was to use shaders to render basic geometry in a way that mimics her own art style, rather than fully realistic rendering. While most "painterly" effects that artistic rendering usually use at least one post-processing pass with external input[5][6], Jennifer wanted to perform as much of this computation as possible in the initial rendering phase. Also, if possible, she wanted to limit external input to only the most essential uniforms and attributes, such as those involving geometry positions, surface normals, and camera uniforms.

Joshua's desire was to create and implement interesting shader image-processing effects for videos/pictures, such as black and white, sepia, and image distortion graphical effects. His aim was to create common graphical effects while trying to create some more interesting effects such as the pixel grid.

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 Rigging, Animating, and Exporting (Lisa)

The majority of the work involved in animating was produced using Blender Animation Software. The process included modeling a character, rigging the character, and animating the character using multiple NLA tracks in one Blender file. The character had to then be exported and loaded into the code to be animated in the browser. This series of steps consisted of multiple experimentations in order to properly rig and animate in such a way that the exported JSON file would be well-prepared for manipulating in the code.

2.1.1 Animation Design

The intent of working with keyframe interpolation instead of using transformations on a mesh's vertices was to simulate complicated, nonlinear actions. Lisa experimented with different actions to animate a non-standard movement, studying Muybridge frames in order to accurately keyframe each motion. The animations had to be properly rigged and parented as well as placed into separate animation actions using the Dope Sheet's action editor. Each of these actions had stored keyframes which were pushed onto the NLA stack to organize the motion strips. This allowed for multiple animation sequences to be exported in one .blend file.[7][8]

2.1.2 Exporting Multiple Animations

One of the most complicated parts of the keyframe interpolation portion of the project was exporting multiple animations in Blender. Documentation on this process seemed to be outdated since Blender created a new exporter, so the animations were finally successfully exported by setting the type to Geometry, selecting Pose and Keyframe animation for animation, and Bones and Skinning for armature. Lisa also selected the vertices, faces, UVs, and normals for Geometry and the Colors and Face Materials for Materials.[9]

2.2 Illumination and Mathematical Drawing Models (Jennifer)

Standard illumination models in conjunction with a couple trigonometric identities form the backbone of the painterly shader. Experimenting with these fundamental equations lead to a fairly simple combination of the two in order to create the basic halftone dotting pattern to color and light the scene, which could then be further altered to increase or decrease highlights or change the basic dotting pattern. The fundamental concepts, Phong Illumination and graphing a Unit Circle, are detailed in the following section.

2.2.1 Phong Illumination

The Phong illumination model[4] calculates the light for a surface based on the diffuse (color of light hitting a surface) and specular (highlight color) for each light in the scene, and a general ambient light constant. The general formula is as follows:

$$I = Ka * Ia + \Sigma(Kd * Id_m * (L_m \bullet N) + Ks * Is_m * (R_m \bullet V)^{\alpha})$$

Where I is the total illumination for that fragment; Ka, Kd, and Ks refer to ambient, diffuse, and specular lighting intensity constants, respectively; Ia, Id, and Is refer to the ambient, diffuse, and specular light colors, respectively; L is the normalized vector from the surface to the light, m; N is the normalized surface normal; V is the normalized viewing vector; R is the normalized reflection vector of the Light vector (L) across the surface normal (N); and finally, α is the glossiness of the surface.

2.2.2 Unit Circle and Trigonometric Identities

The basic equation for graphing a unit circle was used as a basis for computing circular, halftone-dithering highlights based on fragment coordinates. The basic equation for graphing a circle is:

$$(x-h)^2 + (y-k)^2 = radius^2$$

Where (k,h) refers to the center of the circle. From this, we can use the trigonometric identity

$$(cos\theta)^2 + (sin\theta)^2 = 1$$

As a basis for drawing the graph of a unit circle in the fragment shader..

2.3 Post-Processing Experiments (Joshua)

The post processing effects were an experiment of trial and error. After learning the basics on how to get a single pixel from the screen and edit its output color, it was a matter of time until different effects were figured out. First it started with simply overlaying a color modifier of a bluish tint to the texture, and then it when to applying common effects like sepia and grayscale. After some more research, some more ambitious effects were created, such as the pixel dot matrix, the scrolling text effects, and what are known as the hot/warm/cold effects that only draw certain pixels if their red/green/blue values are respectively above certain thresholds. Once the final effects were calculated with some simple Three.js libraries, we were able to apply the shader effects to a video with satisfying results.

3 RESULTS AND DISCUSSION

3.1 Lisa

Once an animated 3D object has been exported in Blender and loaded into a scene using the JSONLoader, the mesh will contain an array of animations. These are animation clips that contain keyframes for each action. The animations can be manipulated and played with Three.js functions.[10]

3.1.1 Using the AnimationMixer

The following code depicts an array of animation clip names:

```
var arrAnimations = [
      'flip',
      'punch',
      'wave',
];
```

After loading the character into the scene using THREE.SkinnedMesh(), Lisa used the AnimationMixer to allow for certain animation clips to be played, paused, etc. Using clipAction, she created an action object with the different actions as keys; in this case, those actions are flip, punch, and wave. Using .play() activates an action to animate.

```
      mixer = new THREE.AnimationMixer(character);
      action.flip = mixer.clipAction(geometry.animations[0]);
      action.punch = mixer.clipAction(geometry.animations[2]);
      action.wave = mixer.clipAction(geometry.animations[3]);
      action.wave.play();
```

The code below is an example of the animation clip for flip containing the name, duration, and keyframes tracks. Also in the JSON file were vertices, normals, and metadata for materials. Because this information was stored with the exported object, there was no need for a vertex shader to provide this data. Due to the fact that the object was loaded using THREE.MeshFaceMaterials(), Lisa was unable to use a vertex shader anyway as the vertices could not be customized.

```
"animations":[{
      "name":"flip",
      "fps":24,
      "length": 9.45833,
      "Hierarchy":[{
            "parent": -1,
            "keys": [{
            "scl":[0.55591, 0.55591, 0.55591],
            "rot": [0.770004, 9.17916e-08, -7.60602e-08.9.63804],
            "pos": [2.15409e-08, 3.79939, 0.605286],
            'time": 0
            },
```

3.2 Jennifer

The halftone dithering effect is by using fragment coordinates to determine which specular or diffuse calculations should be added to the final output color per each fragment. The exact pattern of dots and intensity of illumination can then be altered with a few key constants.

3.2.1 Specular Formula

Artistically, Jennifer found the specular to be the most important part of the lighting computation, as it results in the most interesting and eye-catching part of the effect. The general algorithm is as follows:

Compute Phong Specular:

```
Reflection = compute reflection between Light and Surface Normal;
Normalize Reflection;
RdotV = dot product Reflection and View Vector;
      If RdotV is negative, then RdotV = 0.0
Specular = (dot product Reflection and View)^Glossiness;
```

Create Halftone Pattern:

```
// increase unit circle radius with specular intensity
SpecRadius = BaseRadius * Specular
If (Specular > 0){       // if specular is visible
      CosX = cos(XCoordinateForThisFragment)*SpecRadius;
      SinY = sin(YCoordinateForThisFragment)*SpecRadius;
      // use unit circle function to determine fragment visibility
      If( CosX^2 + SinY^2 >= SpecRadius )
            FragmentColor += (Specular * SpecularLightColor);
      } Else {
      // this fragment is not visible in the halftone
}
Return FragmentColor;
```

Additionally, you can calculate negative specular with just a quick check to RdotV and a second if/else statement. Essentially, you are computing specular in the "negative" direction, on the geometry facing the exact opposite direction of the light.

Negative Specular Check and Calculation:

```
RdotV = dot product Reflection and View Vector;
If RdotV >= 0, then
      Specular = (dot product Reflection and View)^Glossiness;
Else
      Specular = (-(dot product Reflection and View))^Glossiness;
      Specular = -Specular
// . . . Do normal specular calculation
If (Specular is < 0)
      Radius = -Radius
// . . . Same calculation as above
```

3.2.2 Diffuse Formula

The diffuse formula follows the same principles. To enhance the limited-color halftone effect, a diffuse threshold was used to force the diffuse to be in a specific range of colors--similar to a standard toon shader.

Compute Phong Diffuse:

```
Diffuse = dot product of Normal, Light direction
      If Diffuse < 0, then Diffuse = 0

// create shading thresholds
ThresholdValue = 0.1;
For( i=0; i<1; i += ThresholdValue ){
  // determine halftone pattern
  If (Diffuse >= i ){
      CosX = cos(XCoordinateForThisFragment);
      SinY = sin(YCoordinateForThisFragment);
      // use unit circle function to determine fragment visibility
      If( CosX^2 + SinY^2 >= Diffuse )
            FragmentColor += (Diffuse * DiffuseLightColor);
  }
}
Return FragmentColor;
```

3.2.3 Altering the Halftone Pattern

The two best ways to alter the halftone pattern in a controlled manner are to change the Specular BaseRadius, and to divide the X and Y Fragment Coordinate before computing sin or cos, respectively. A larger BaseRadius results in a stronger, smoother specular (until eventually it becomes normal Phong Illumination), and smaller, more diluted halftone with a smaller radius (until eventually it disappears). When dividing the X or Y FragCoord by a constant, it is helpful to stay within the range of 0-6.28; dividing by pi in particular creates a solid circle of color.

3.3 Joshua

For the post-processing effects a number of mathematical equations were used to reproduce common post processing effects such as grayscale and sepia as well as more uncommon effects such as a pixelated grid look and video scrolling.
For the sepia effect the computational equation to do so is as follows.

```
     vec4 sepia(vec4 c1){
            float inputRed= c1.x;
            float inputGreen=c1.y;
            float inputBlue=c1.z;
            float outputRed = (inputRed * .393) + (inputGreen *.769) +
(inputBlue * .189);
            float outputGreen  = (inputRed * .349) + (inputGreen *.686) +
(inputBlue * .168);
            float outputBlue = (inputRed * .272) + (inputGreen *.534) +
(inputBlue * .131);
            return vec4(outputRed,outputGreen,outputBlue,1.0);
      }
```

Where the function would take in a vector4 full of color components and then calculate different weights for the output color based on the input to give us the desired Sepia look.
A similar calculation is used for grayscale which is calculated as follows.

```
     vec4 greyScale(vec4 c1){
            float Red=c1.r;
            float Green=c1.g;
            float Blue=c1.b;
            float Gray = (Red * 0.3 + Green * 0.59 + Blue * 0.11);
            return vec4(Gray,Gray,Gray,1.0);
     }
```

Where here we take in a similar vector4 color component and then calculate a combination color for gray based on the different amount each color has going in.

Different effects can be calculated too that don't just affect the color output of a fragment or pixel as well.
For example one of the calculations includes a horizontal scrolling effect, where based off of the image/video dimensions and a time uniform we are able to offset the pixel position to be a different pixel on the same y coordinate.

```
     vec4 sidewaysScroll(vec4 c1){
            float grabPos=gl_FragCoord.x+horizontalScrollAmount;
            if(grabPos>screenResolution.x) grabPos=grabPos -
screenResolution.x;
            vec2 pt=gl_FragCoord.xy;
            float cx = grabPos/screenResolution.x;
            float cy = pt.y/screenResolution.y;
            vec4 col = texture2D( t1, vec2( cx, cy )); //get the pixel from the
offset amount.
            return col;
      }
```

Similar interesting effects can occur with some more complex mathematical operations such as using the modulus operator for float arithmetic to create an effect where we are able to make a pixel grid effect where only small squares of pixels are visible at a time over the entire texture used and set the rest of the unused area to be black. This creates an effect similar to a dot matrix where images are rendered using a variety of colored dots. The algorithm for that follows.

```
      vec4 dotMatrix(vec4 c1){
            float radius=effectRadius; //The width and height of each square in
pixels.
            vec2 pt=gl_FragCoord.xy;
            //mod means every x lines different color happens.
            vec2 newPos=pt.xy *1.0; //float conversions to be safe.
            vec4 pixCol=vec4(0.0,0.0,0.0,1.0);
                  if(mod(newPos.x,radius)>=radius/2.0 &&
mod(newPos.y,radius)>=radius/2.0)
                  pixCol=c1;
            else pixCol=vec4(0.0,0.0,0.0,1.0); //If this pixel is not in a
square it is black.
            return pixCol;
      }
```

All of these algorithms contribute to the plethora of effects released in the project and some of the key ones that were chosen to be showcased in the paper.

---

4 CONCLUSIONS

Lisa came across several issues with outdated documentation and sources. She was able to successfully animate her character by using a work-around.[9] to export the 3D animated object, but this resulted in some odd changes to the vertices of the character as it was animating. She also had to import an outdated version of the Three.js library, which made her code incompatible with the others when combining.

Jennifer discovered that for a lot of complex artistic effect, you really do need that second rendering pass with additional stroke, depth, or specular information fed into the shader. Her effect doesn't exactly look like her art; however, the resulting effect is interesting, and may form the basis of some other first-pass rendering effects, like cross hatching or other shapes.

Joshua accomplished all his goals regarding making common shader effects. He was able to achieve some interesting effects such as a pseudo-dot matrix, a sepia, and a image scrolling effect. It all worked for the most part aside from the glitch shader effect due to lack of time. The complexity of some of the image effects were as simple as an if statement, while others required more intensive calculations that require the use of some math to determine neighboring pixels.

Finally, our general plan to link together the code became fairly difficult because of the difference in Three.js libraries being used. We were unable to combine the code as we had expected since Lisa's code needed to use an older version of Three.js as her code would not render properly using the current version that Jenny and Josh's code needed. We ended up using an unanimated version of Lisa's code to be able to apply the post-processing effects on the scene. However, the work was naturally segmented because each person worked on one part of the rendering process; had the library not had an issue, the code all would've fit together quite nicely.

ACKNOWLEDGEMENTS

Big thanks to Angus and Lucas for all they've done, it's been an amazing quarter!

REFERENCES

[1] Incin. Simple Dot Matrix, Shadertoy (2016). Retrieved from https://www.shadertoy.com/view/4sySDd

[2] Tanner Helland. Seven grayscale conversion algorithms (with pseudocode and VB6 source code), tannerhalland.com (2011). Retrieved from http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

[3] user83358. How is a sepia tone created? Stackoverflow, 2009. Retrieved from https://stackoverflow.com/questions/1061093/how-is-a-sepia-tone-created

[4] Bui Tuong Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975), no. 6, 311–317.

[5] Pal, Kaushik, A shader based approach to painterly rendering. Master's thesis, Texas A&M University. Texas A&M University.

[6] Domingo Martin, Vicente del Sol, Celia Romo, and Tobias Isenberg. Drawing Characteristics for Reproducing Traditional Hand-Made Stippling. Conference Paper, EXPRESSIVE 2015: Non-Photorealistic Animation and Rendering. EXPRESSIVE, Istanbul, Turkey.

[7] M. (2017, April 27). Rigging People | Blender | Quick | Beginner. Retrieved March 25, 2018, from https://www.youtube.com/watch?v=cp1YRaTZBfw

[8] M. (2013, April 28). Animating a Rigged Character in Blender. Retrieved March 25, 2018, from https://www.youtube.com/watch?v=3gJaSl2jd_M

[9] Blender: Export multiple animations · Issue #6326 · mrdoob/three.js. (n.d.). Retrieved March 25, 2018, from https://github.com/mrdoob/three.js/issues/6326

[10] Workflow: Animation from Blender to three.js. (n.d.). Retrieved March 25, 2018, from http://unboring.net/workflows/animation.html