DYNAMIC KELVINLETS: PROCEDURAL ELASTIC DEFORMATIONS

HOW TO MAKE YOUR GAME OBJECTS STRETCH, RIPPLE AND SCALE ON THE FLY

Animating a mesh deformation usually requires a pre-defined animation to be authored by an artist before it can be applied in a game. Procedural deformation techniques, such as constraint-based dynamics, position-based dynamics (PBD) and finite element methods (FEM) warrant a non-trivial pre-processing of the mesh during the game's initialization. If we wish to tweak the deformation, changes must be made offline: either the animation has to be rewritten by the artist or the mesh must be pre-processed with different parameters. Here, we present a novel technique for customising and applying procedural elastic deformations to any mesh at run-time.

WHAT ARE DYNAMIC KELVINLETS? In the theory of linear elastodynamics, elastic disturbances propagate as waves throughout the material, radiating away from the source. We can decompose these waves into two types: pressure (P) waves, which travel as longitudinal compressions, and shear (S) waves, which are transverse displacements. Assuming that the material is isotropic (has the same properties throughout), these waves will radiate spherically, depending only on the distance r from the epicentre, and the time t since the disturbance began.

The disturbances $\mathbf{u}(\mathbf{r}, t)$ caused by a force field $\mathbf{F}$ obey the elastic wave equations,

$$\frac{\partial^2}{\partial t^2}\phi = \alpha^2\nabla^2\phi + \psi, \qquad \frac{\partial^2}{\partial t^2}\boldsymbol{\Phi} = \beta^2\nabla^2\boldsymbol{\Phi} + \boldsymbol{\Psi},$$

where we have applied the Helmholtz decompositions $\mathbf{u} = \nabla\phi + \nabla\times\boldsymbol{\Phi}$ and $\mathbf{F} = \nabla\psi + \nabla\times\boldsymbol{\Psi}$. The constants α and β refer to the speeds at which P-waves and S-waves travel through the material respectively. These equations are difficult to solve, but they are linear, which means that we can find fundamental solutions and add them together to build any number of solutions.

A fundamental solution considers the simple case where there are no boundary conditions (the entirety of R^3 is treated as an isotropic elastic medium), and the force $\mathbf{F}$ is nothing more than a point impulse applied to the material at time $t = 0$. Mathematically, we write this as $\mathbf{F}(\mathbf{x}, t) = \mathbf{f}\delta(\boldsymbol{x} - \boldsymbol{c})\delta(t)$, where $\mathbf{f}$ is a constant vector and δ is the Dirac-delta function. Solving the elastostatic wave equations (without time evolution) using fundamental solutions was first attributed to Lord Kelvin in 1848 [1] and the resulting solutions are named Kelvinlets [2] in his honour.

A point impulse, as described above, is numerically unstable and is not amenable to computer simulations. Furthermore, such an instant load of infinite magnitude, whilst mathematically sound, is physically implausible. To overcome this, we can use the technique of regularization [3], to smooth out the impulse and avoid singularities at the epicentre. In the case of regularized Kelvinlets [4], [5], we choose an initial impulse of the form $\mathbf{F}(\mathbf{r}) = \mathbf{f}\rho(r)\delta(t)$, where $\rho(r) = 15\varepsilon^4/8\pi r_\varepsilon^7$, using the regularized radius $r_\varepsilon = \sqrt{r^2 + \varepsilon^2}$. Figure 1 shows how increasing the regularization scale ε smooths out the initial impulse.

Author: Ismail Movahedi

Supervisor: David Moore

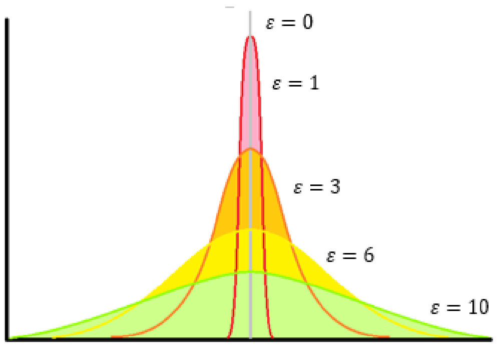

Figure 1: Increasing the regularization scale ε decreases the amplitude of the impulse and increases its area of effect. The degenerate case $\varepsilon = 0$ collapses down to the Dirac-delta point impulse.

Once we have constructed a regularized fundamental solution, we can add together any linear combination of Kelvinlets to produce more interesting solutions, in particular, affine loads. These consist of the more familiar elastic disturbances, specifically pinching, scaling and twisting. The closed form solutions for these disturbances can be found in Appendix A.

DEVELOPMENT METHODOLOGY The main aim of this project was to explore the theory of dynamic Kelvinlets and how they can be integrated into a game engine. Using Feature-Driven Development (FDD), we incrementally developed a program that allows users to schedule, edit and play any number of Kelvinlets to a mesh, up to an arbitrary, pre-defined maximum. All programs were developed using the *Debug Draw* rendering framework written in C++ and DirectX 11 by Guilherme R. Lampert. The GUI was implemented using the lightweight *Dear ImGui* library.

As we are exploring a fairly novel technique, there is, at the time of writing, no reference implementation for applying Kelvinlets to a game engine. Previous work in [4], [5] used the theory to develop sculpting tools in *Houdini* for authoring animation frames, but their performance was not measured and the final animations were rendered offline. Due to the high risk associated with applying Kelvinlets to a game environment, where performance takes precedence, we adopted a Spiral Model to manage the program's lifecycle. The iterative nature of the spiral model suited our project, where the each feature developed is prototypical and informs the development of the feature after it.

During each iteration around the spiral (typically one to two weeks), a new feature would be developed using four principal stages. First, we would identify the requirements of the feature to be implemented. This would be followed by identifying the risks associated with developing the feature. The remainder of the iteration would then be used to design, implement and test the code for the new feature, before finally evaluating the program. For our purposes, evaluation will comprise of performance profiling, and a brief assessment of the visual quality of the results. The table in Figure 2 outlines the development schedule over the course of the project.

FEATURES TO DEVELOP	ESTIMATED TIME	TIME SPENT
Proof-of-concept: Implement a single Kelvinlet on a single mesh. Assess different types of Kelvinlet.	2 weeks	3 weeks
Apply multiple Kelvinlets to a single mesh.	2 weeks	2 weeks
Create a simple game object to which we can add Kelvinlets at run-time.	2 weeks	3 weeks
Provide an interface for the user to construct a timeline of Kelvinlets.	2 weeks	1 week

Figure 2: Each development phase aims to assess, design, implement and test a feature every two weeks.

PHASE 1: SINGLE KELVINLET, SINGLE MESH The first feature implemented was a simple proof-of-concept, simulating a single Kelvinlet applied to a single mesh. Solutions based on dynamic Kelvinlets require the position vector $\mathbf{r}$ in the elastic material and the point in time t. Unlike mass-spring systems, there is no dependence between the displacements of neighbouring vertices, so these calculations can be performed in parallel on a per-vertex basis. Hence, the first attempt to simulate dynamic Kelvinlets was implemented using a vertex shader in HLSL. Whilst this was intended to be one of the shorter development phases, there were a number of design problems to consider before this could be implemented.

Defining each type of Kelvinlet requires different parameters (see Appendix A for a more detailed discussion), and this information needs to be sent from the CPU to the GPU every frame. For a single Kelvinlet, this data can be passed in a constant buffer, but for multiple Kelvinlets, this proves a very inflexible design. Furthermore, affine and impulse Kelvinlets, as described in [4], [5], require different data structures, meaning a hard-coded change to the constant buffer every time we want to simulate a different type. In spite of this, the developed prototype demonstrated that all Kelvinlets, with the exception of twist Kelvinlets, can be rendered and simulated in real-time. Figure 3 shows an example of an impulse Kelvinlet being simulated.

Twist Kelvinlets create visual artefacts as the resulting torsion causes large separations between mesh vertices, leading to an under-sampling of the mesh. To work around this, the deforming object must be dynamically re-meshed [6], but the implementation of this is beyond the scope of the features intended for development.

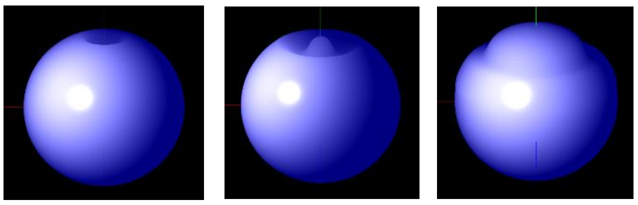

Figure 3: An impulse Kelvinlet propagates radially from the epicentre as time progresses.

Because of this, future iterations do not implement twist Kelvinlets and our focus is restricted to impulse, pinch and scale Kelvinlets.

PHASE 2: MULTIPLE KELVINLETS, SINGLE MESH. For a single Kelvinlet, placing the parameters within a constant buffer was sufficient. From the structure shown in Figure A1, we see that the 16-byte alignment imposed on constant buffers leads to 4 bytes of memory being wasted per Kelvinlet. For efficiency and spatial locality when applying multiple Kelvinlets, all of the Kelvinlet data is now packed into a structured buffer that is sent to the GPU.

The vertex shader is invoked on a per-vertex basis, so we must iterate over the entire structured buffer of Kelvinlets for each vertex, collecting all of the displacements that affect that vertex. In a game engine, the vertex shader is often responsible for performing work that is tied to the rendering pipeline, such as lighting calculations and mesh skinning. The fact that the elastic wave equations are linear means that the displacements due to multiple Kelvinlets can be added together and stacked in any order. We are not restricted to performing Kelvinlet calculations in the vertex shader, although vertex positions are required.

Using a compute shader, we can remove some of the burden from the vertex shader and extricate Kelvinlets from the rendering pipeline. Results from Kelvinlet calculations are output from the compute shader into a structured buffer, where each element holds a cumulative displacement vector for one of the mesh's vertices. This means that we can use the same vertex/pixel shader pair to render meshes regardless of whether or not they are being deformed. The vertex shader simply has to read from the displacement buffer – static meshes will have a buffer of null displacements.

The aim of this phase was to extend the proof-of-concept from the last phase, to demonstrate that a typical real-time rendering framework could support multiple Kelvinlets. These first two phases were very high risk, as failure to implement Kelvinlets would mean that the artefact could be developed no further. As such, these first two prototypes were developed quickly, and the code written in a procedural fashion. Before developing further features, the code would have to be refactored with a more object-oriented approach.

PHASE 3: ADDING NEW KELVINLETS AT RUN-TIME
Once the code was successfully redesigned, the next feature to develop was a user interface for adding Kelvinlets whilst the application was running.

Structured buffers, once created, cannot change size, so the number of Kelvinlets we pass to the GPU every frame must remain fixed. To create the illusion of a dynamic number of Kelvinlets, the structured buffer is treated as an object pool [7], where free blocks in the Kelvinlet buffer are treated as 'null Kelvinlets', which cause no displacement. When the user requests to add a Kelvinlet, the program checks that the buffer has free space, before filling the first available space with new Kelvinlet's data. As a large portion of this phase was spent refactoring code, the user interface for this program was kept minimal. The user could click on a button that would generate an impulse Kelvinlet applied to a random point on the surface of a spherical mesh with each click.

In the prototype, a typical game object is represented by the `MeshInstance` class (see Figure 4). As well as holding the minimum data needed for rendering (a pointer to a mesh, a pointer to a texture and world space positional coordinates), it also contains a `KelvinletManager` class, which deals with all of the logic necessary for applying Kelvinlets to that game object. In attaching Kelvinlets as a component that is decoupled from the rendering pipeline, we aim to demonstrate that they can be integrated into most game engines. Here, we have demonstrated an implementation in an object-oriented framework, but their self-contained nature shows promise for a future direction of research to add Kelvinlets to an entity-component-system architecture, such as the *Unity* engine.

The `KelvinletManager` class contains an array of Kelvinlets and subscribes to the game loop's `Update()` method. During the update portion of the game loop, this class will iterate over all of its valid Kelvinlets, incrementing their age and retiring them once they have exceeded their lifespan. When requested, this class also provides access to the underlying array, so that it the updated data can be copied into a structured buffer and pushed to the GPU every frame.

```
class MeshInstance
{

//...
private:
        size_t m_id;    // Unique ID
        v3 m_position;
        Mesh* m_pMesh;
        Texture* m_pTexture;
        KelvinletManager m_kManager;
};
```

Figure 4: A game object owns the Kelvinlets that are applied to it. The Kelvinlet manager component is responsible for a container of Kelvinlets, providing an interface for adding, removing and updating them.

Upon completing this developmental phase, the prototype could now apply a dynamic number of multiple Kelvinlets to a single mesh. However, the user interface left much to be desired. Building upon the foundation laid in this phase, the next feature to implement was a timeline functionality. This would allow the user to specify any number of any type of Kelvinlets to apply to the mesh (up to a hard-coded maximum), along with their individual starting time and lifespan before they are retired.

PHASE 4: CREATING A KELVINLET SCHEDULE

To avoid excessive branching during simulation, the first design choice was to split the application's flow of logic into two modes. Either the application is in *Edit Mode*, where the user can add or remove custom Kelvinlets from the timeline, or the application is in *Play Mode*, where the specified timeline is locked from editing and the simulation is rendered in real-time.

As the `KelvinletManager` component now has additional responsibilities, it has been split across different classes. To apply distortions to its mesh, a `MeshInstance` now holds two components: a `DisplacementManager`, which is responsible for the vertex displacement buffer output from the compute shader each frame, and an updated `KelvinletManager`. This class now contains a `Timeline` wrapper around the Kelvinlet container, and also manages the parameters α and β that characterise the mesh's physical properties.

During *Play Mode*, a slider bar is exposed to the user that allows them to select any point from within the timeline to view the animation. Tests with the application indicate that performance is smooth, with no drop in performance when moving along the timeline, and no issues when switching between *Edit Mode* and *Play Mode*. In addition, users can also pause the timeline at any point to closely observe the behaviour of the animation.

As the development of this phase was faster than anticipated, the extra time was used to implement a normal-correction scheme for lighting. As discussed in [6], distorting the mesh by moving the vertex positions alters the mesh's shape, but the vertex normals remain the same and still correspond to the original mesh. This can produce incorrect lighting for sufficiently large distortions. Our simple normal correction scheme, as outlined in [6], uses the normal vector $\mathbf{N}$ and tangent vector $\mathbf{T}$ of a vertex to construct a bitangent vector $\mathbf{B} = \mathbf{N} \times \mathbf{T}$.

Using $\mathbf{B}$ and $\mathbf{T}$ as the basis of the vertex's tangent plane, we can define construct two local points in the neighbourhood of $\mathbf{r}$: $\mathbf{r}_{\mathbf{T}} = \mathbf{r} + \lambda\mathbf{T}$ and $\mathbf{r}_{\mathbf{B}} = \mathbf{r} + \mu\mathbf{B}$, for constants $0 < \lambda, \mu \ll 1$. As well as computing the displaced vertex position $\mathbf{r}'$, we also compute the displaced local points $\mathbf{r}'_{\mathbf{T}}$ and $\mathbf{r}'_{\mathbf{B}}$ to account for the distortion of the surface's local tangent plane. Then, we can estimate the new vertex normal post-displacement by taking the cross product of the distorted tangent and bitangent:

$$\mathbf{N}' = \mathbf{T}' \times \mathbf{B}' \approx (\mathbf{r}'_{\mathbf{T}} - \mathbf{r}') \times (\mathbf{r}'_{\mathbf{B}} - \mathbf{r}') = \mathbf{N}'_{A}.$$

Figure 5 shows a visual representation of this method.

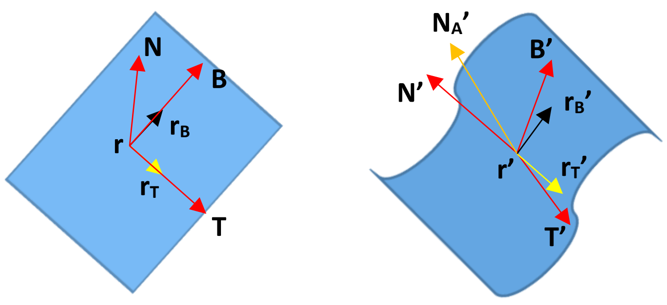

Figure 5: On the left, we have a segment of the original mesh, and on the right, we have the distorted segment once Kelvinlets have been applied. By calculating the displacement of points in the neighbourhood of the displaced vertex, we can estimate the distorted tangent plane and produce an estimated normal N_A'.

FUTURE WORK A potential extension to the timeline functionality would be the ability to save arrangements of Kelvinlets to file, so that once a desired animation sequence had been produced, it could be readily reused. A single Kelvinlet structure is very lightweight (44 bytes), and, once plugged into the framework, can produce a rich set of animations without the need of a vertex cache or animation file, saving a lot of memory.

Another area to research would be to implement twist Kelvinlets by dynamically re-meshing the deforming mesh to accommodate large deformations. However, it is unknown how this would impact textured meshes, as the methods for re-meshing discussed in [6] do not address UV unwrapping the new mesh.

RESULTS Throughout the application's development, performance was profiled using two main methods. *Visual Studio 2017* has a built-in CPU profiler for monitoring function calls and execution time each frame, and a built-in GPU profiler which provides a detailed breakdown of GPU tasks every frame. The debugging application *RenderDoc* was also used extensively to provide valuable insights into resources allocated on the GPU, and to keep track of the commands being executed by the GPU each frame. By comparing the results from these two profilers, we were able to ensure that the application was performing as intended after every feature phase, and identify any potential bottlenecks to be resolved in the next iteration. Figure 6 shows the frame times recorded in the final application on each profiler, along with the frame delta time recorded using *Query Performance Counter* from the Windows API.

Profiler	VS 2017	RenderDoc	Query Performance
Frame time (Edit Mode)	1.9ms	1.8ms	1.7ms
Frame time (Play Mode)	2.8ms	2.8ms	2.9ms

Figure 6: Edit Mode only renders the mesh, taking roughly 1.8ms. Switching to Play Mode dispatches the compute shader, adding an extra microsecond to the frame time. Results were measured on an *Intel i5-4210U* integrated processor.

We can see that the application takes roughly 2.8ms to render a frame, with around 1ms of this time taken to simulate Kelvinlets. A typical console game aims to run at 60FPS (16.6ms per frame), so Kelvinlets show promise as a viable option for procedural animation in a game engine.

Dynamic Kelvinlet simulations have also proven to be versatile within a rendering framework. They can either be included in the rendering pipeline with calculations performed and applied in the vertex shader, or they can be handled externally using the compute shader capabilities of modern graphics APIs. If applied in the vertex shader, Kelvinlets can take advantage of the vertex data that is already on the GPU.

When using the compute shader, we must first copy the vertex positional data into a structured buffer, since the vertex buffer cannot be accessed from the compute shader. Then, we must create another structured buffer for storing the net displacements at every vertex, to be read by the vertex shader. What we gain from decoupling the calculations from the vertex shader, we must make up for with memory consumption, creating two additional structured buffers per mesh instance, whose elements number the mesh vertices.

DISCUSSION One issue that could prevent Kelvinlets from being added to a game engine is the need for high-poly meshes to produce convincing effects. It is a common technique to use low-poly meshes in games, and use normal maps and textures to disguise their low resolution. Unfortunately, displacing vertices on a low poly mesh breaks this illusion and does not produce pleasing results.

As shown in Figure 7, an undistorted low-poly sphere appears smooth due to normal interpolation in the lighting calculations. Once we throw Kelvinlet displacements into the mix, the distorted regions of the mesh appear blocky and angular. Comparing these results with the images in Figure 3, we also lose the appearance of a smooth, continuous elastic wave propagating through the mesh. The sphere used in Figure 3 is made up of 61440 vertices, whereas the mesh in Figure 7 contains 960. Due to the large level of detail required for convincing distortions, Kelvinlets are not recommended for low-poly game models.

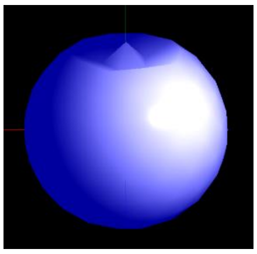

Figure 7: Meshes with a low resolution are not suitable for convincing elastic deformations. The blocky appearance that results negates any efforts (normal maps, textures, etc.) to make the mesh appear smooth.

Another important consideration is the topology of the mesh we are deforming. For a simple mesh with no breaks, the deformation appears to propagate through the shape. However, for shapes whose ends are topologically distant but spatially close (see Figure 8), the effects can be noticeable. This is because Kelvinlets are fundamental solutions, treating the entire space as if it were deforming and ignoring any shape-specific boundary conditions. The simulated distortions propagate radially through space, unaware of the topology of the mesh that is being deformed. To the viewer, the distortion is propagating through empty space, reaching seemingly distant points in the shape prematurely. To avoid this artefact, we must only apply Kelvinlets to meshes whose vertices have similar spatial and geodesic separations.

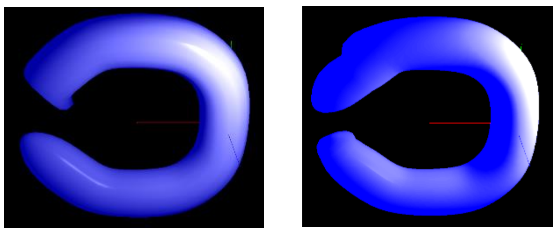

Figure 8: A Kelvinlet is a fundamental solution, warping all of space as if it were elastic. We expect the distortion to travel from one end of the shape to the other. However, as the two ends are close together spatially, the elastic wave starts at one end and affects the other end **before** travelling around the shape.

In spite of these physical inaccuracies, Kelvinlets do have their advantages over other contemporary methods of deforming meshes. Finite element methods (FEM), such as those used in *Star Wars: The Force Unleashed* [8] cannot be readily applied to any game object. The mesh must be specially processed and divided into tetrahedra before it can be used in the game. The way in which the mesh is divided affects the physical parameters in the simulation, and this cannot be changed whilst the game is running, unlike dynamic Kelvinlets.

Another popular alternative method is constraint-based dynamics, which treat the mesh's vertices as a system of particles and apply a series of constraints (e.g. volume-preservation, conservation of momentum). These constraints are typically non-linear and cannot be solved exactly, requiring numerical methods which introduce errors.

Position-based dynamics (PBD) [9] is a popular variant of constraint-based dynamics, used in Nvidia's *PhysX* library for cloth simulations. However, if we wish to apply PBD to simulate elastic solids, we require a tetrahedral division of the mesh's volume, as with FEM. As we have shown, Kelvinlets can be applied to most meshes, without any prior preparation of the mesh. Furthermore, Kelvinlets can be easily customised, added, removed and simply turned off completely at run-time, providing great flexibility.

Lastly, PBD is framerate dependent, as the iterative solutions depend on the time-step every frame [10]. On the other hand, the closed form solutions that underpin Kelvinlet simulations are exact, deterministic and do not depend on framerate. We simply advance the time parameter with the measured frame time, so that all platforms exhibit similar results.

CONCLUSION To summarise, we have presented a novel framework for applying procedural, physically based elastic deformations to any mesh. Whilst the results may not be entirely physically accurate, the calculations are closed-form, frame-rate independent and deterministic. We have outlined an implementation that is amenable to both object-oriented and entity-component game engine architectures. Dynamic Kelvinlets are flexible, lightweight and can be integrated outside of the rendering pipeline.

In the future, we hope to implement twist Kelvinlets and overcome some of the artefacts that plague large mesh deformations.

APPENDIX A: DEFINING A KELVINLET All Kelvinlets share common parameters (epicentre $\mathbf{c}$, time-point t, regularization scale ε, material parameters α, β), but in the theory developed in [4], [5], each type of Kelvinlet is distinguished by different sets of defining parameters. For point impulse Kelvinlets, we need the direction of the impulse $\mathbf{f} \in \mathrm{R}^3$, but to define an affine Kelvinlet (pinch, scale or twist) we require a 3x3 matrix.

It is tempting to create an inheritance hierarchy, since a scale Kelvinlet *is a* Kelvinlet. If we proceed in this manner, then we run into problems for simulating multiple Kelvinlets. To store them in the same container to be passed to the GPU, we must store them as an array of Kelvinlet pointers. Firstly, CPU pointers are meaningless to the GPU, and even if the memory could be accessed, the GPU has no means of identifying the type of Kelvinlet from a pointer to the base class. In practise, it turns out that the code is easier to implement if we eschew inheritance altogether, and opt to use an integer enumeration to identify the Kelvinlet's type on the GPU. In addition, we also do not need a full 3x3 matrix to define an affine Kelvinlet. In fact, all types of Kelvinlets can be defined and identified by subscribing to a single data structure, shown in Figure A1.

When defining a scale Kelvinlet, the matrix is simply a scalar multiple of the identity matrix $s\mathbf{I}$, where s is the scale factor. In fact, all we really need is the scale factor s for the definition. For pinch Kelvinlets, the eigenvalues of the matrix correspond to the magnitude of the expansion or contraction along the principal axes of the matrix. Negative eigenvalues correspond to contraction, whereas positive eigenvalues represent an expansion along that axis. If we restrict the distortions to being aligned along the x, y, z world space coordinate axes, the matrix becomes a diagonal in world space. Then, we only need the three eigenvalues (scale factors) to define expansions and contractions, instead of an entire 3x3 matrix.

Twist Kelvinlets work by rotating points r in the material about a common axis $\mathbf{q}$. Part of the fundamental solution in [5] requires the 3x3 matrix $[\mathbf{q}]_\times$, which is defined such that $[\mathbf{q}]_\times \mathbf{r} \equiv \mathbf{q} \times \mathbf{r}$. However, the sole purpose of the matrix $[\mathbf{q}]_\times$ is computing a cross-product, so we only need the vector $\mathbf{q}$ to define a twist Kelvinlet instead of the 3x3 matrix. This means that all types of Kelvinlet, both impulse and affine can be described using the same data structure. Each type differs by how the vector of three floats, `forceParams`, is interpreted. For impulse Kelvinlets, it defines the direction of the initial impulse; for pinch and scale Kelvinlets, we interpret the vector's elements as scale factors of expansion and contraction in corresponding x, y or z direction.

```
struct Kelvinlet
{
	float3 loadCentre;  // Epicentre
	float epsilon;      // Regularization
	float3 forceParams;
	float age;          // Evolution time
	float startTime;
	float lifespan;
	int type;
};
```

Figure A1: Animating a Kelvinlet requires the data in the above structure (44 bytes), the mesh parameters α, β (8 bytes) and the mesh vertex data, which we get for free in the rendering pipeline.

REFERENCES

[1] – W. Thompson (Lord Kelvin), (1848). *Note on the integration of the equations of equilibrium of an elastic solid,* Cambr. Dubl. Math. J., 3, 87-89.

[2] - N. Phan-Thien, & S. Kim, (1994). *Microstructures in Elastic Media: Principles and Computational Methods.* Oxford University Press, Oxford, UK.

[3] – R. Cortez, (2001). *The Method of Regularized Stokeslets*. SIAM J. Sci. Comput. (USA), 23(4), 1204-1225.

[4] – F. de Goes, & D. L. James, (2017). *Regularized Kelvinlets: Sculpting Brushes Based on Fundamental Solutions of Elasticity*. ACM Trans. Graph. 36(4), Article 40.

[5] – F. de Goes, & D. L. James (2018), *Dynamic Kelvinlets: Secondary Motions based on Fundamental Solutions of Elastodynamics*. ACM Trans. Graph. 37(4), Article 81.

[6] – W. von Funck, H. Thiesel, & P. Seidel, (2006). *Vector-Field Based Shape Deformations*. SIGGRAPH '06, ACM Trans. Graph. 25(3), 1118-1125

[7] – R. Nystrom, (2014). *Game Programming Patterns*. Genever Benning.

[8] – E. G. Parker, & J. F. O'Brien, (2009). *Real-Time Deformation and Fracture in a Game Environment.* Eurographics/ACM SIGGRAPH Symposium on Computer Animation 2009.

[9] – M. Müller, B. Heidelberger, M. Hennix, & J. Ratcliff, (2007). *Position-Based Dynamics*. J. Vis. Comun. Image Represent. 18, 2 (Apr.), 109–118

[10] – M. Macklin, M. Müller, & N. Chentanez, (2016). *XPBD: Position-Based Simulation of Compliant Constrained Dynamics*. MIG '16 Proceedings of the 9th International Conference on Motion in Games. 49-54.