Volumetric Interaction: Real-Time Dynamic Volume Textures

Jordan Hamlin
Faculty of Engineering and IT
University of Technology Sydney
Sydney, Australia
Jordan.T.Hamlin@student.uts.edu.au

***Abstract*—This paper presents a prototype system for generating dynamic volume textures for reactive shaders and assesses its performance on hardware of varying capabilities to conclude the viability of dynamic volume textures in games.**

Keywords—volume texture, vector field, procedural texture, benchmark, jump flooding algorithm, real-time

I. Introduction

Textures can be used to store useful information, such as the vectors defined by a vector field [2]. A dynamic texture can represent a changing vector field by updating in real-time. Many modern games use dynamic textures to create reactive environments – environments where entities of the game world can influence the rendering of materials. Modern games are incorporating greater verticality in their environment design, which highlights some limitations of current solutions for creating reactive environments using 2D dynamic textures.

This paper presents a prototype system for generating dynamic volume (3D) textures as a solution for creating reactive environments. Modern Graphics Processing Units (GPU) can render dynamic volume textures which can represent vector fields more realistically.

Section II will present related work in existing reactive environment solutions, volume texture generation and visualisation. Section III will discuss the architecture of the prototype system and the approach to assessing the system's performance on multiple devices. Section IV will analyse the benchmark results and identify the performance impact of different aspects of the prototype. The current viability of the prototype will be concluded, and future improvements will be proposed.

II. Background

A. Related Work

1) Vector Fields

Vector fields are regions in space containing a set of vectors representing bounded points [3]. They are often used in games to define movement at points in space, such as AI pathfinding and particle system flow. Vector fields can be represented by textures, where each bounded point is represented by a pixel, with the colour channels storing the vector [2]. This can be done by remapping each normalised component of a vector from the range -1 to +1, to the range 0 to +1. For example, a vector (-1, 1, 0) can be encoded as the colour (0, 1, 0.5).

2) Reactive Shaders

Reactive shaders refer to shaders with parameters that are determined by the state of objects in the game world. Reference [1] demonstrates a fluid shader where the surface normals are affected by projectiles and the player. This example uses the method of encoding vector fields as a texture to store the ripple information of the fluid surface. One notable issue with this technique is that a separate series of textures for each instance of a fluid surface must be created, resulting in exaggerated file sizes and slower performance for each surface in the world. This could be avoided by rendering the texture with an axis-aligned camera; however, this raises the possibility of issues caused by occlusion. Another example of a reactive shader is shown in [4]. This shader renders grass blades that bend according to collisions with defined spheres. Rather than encoding the collision information in a texture, each sphere is represented by a vector that is passed directly into the shader. This approach gets exponentially slower as the number of spheres increases because each vertex needs to iterate over every sphere.

3) Jump Flooding

The Jump Flooding Algorithm (JFA) produces Voronoi diagrams and distance fields in logarithmic time and alleviates some issues present in existing reactive shaders [7]. JFA avoids the problem of an exponential decrease in speed with an increase in effectors previously mentioned by sacrificing the consideration of multiple effectors by each vertex for only considering the closest.

Further, one proposed variant to JFA [8] details a method of generating volumetric diagrams. By encoding a vector field as a volume texture, the vector field can be disassociated from individual objects while still avoiding occlusion issues that could occur when using an axis-aligned camera. However, the proposed method of rendering to a series of 2D slices is outdated, as most GPUs now support writing to volume textures.

4) Ray Marching

Volume textures cannot be visualised as simply as 2D textures. The ray marching algorithm proposed in [6] can be used to render volume textures efficiently. By marching a ray outward from each screen pixel and performing an intersection test against the bounding box of the volume, a path from the entry point to the exit point can be found, along which to sample the texture.

The more efficient ray-box intersection algorithm proposed in [5] can further increase the performance of the ray marching algorithm.

B. Proposal

The proposed prototype system will dynamically generate volume textures representing vector fields. These vector fields will be affected by a variable number of sources that exert an outward force from a given point in space. JFA will be utilised to flood-fill the volume textures in a scalable manner allowing for a larger number of sources, and more complex shapes constituting multiple sources like the solution in [4]. Ray marching and ray-box intersection testing against the volume bounds will allow for rendering a debug visualisation of the volume texture. The system will provide a simple API for developing reactive shaders relying on the prototype system.

The viability of rendering dynamic volume textures in games will be assessed using a benchmarking program. This program will assess the performance impact of various parameters of the prototype system.

III. Prototype Development

A. Tools

The following tools were used to develop the prototype system and benchmarking program.

1) Unity Game Engine

- *Compute Shader API:* The API allowed for simple assignment of global and per-kernel compute shader parameters. The process of creating compute buffers enabled seamless conversion between C# and HLSL user-defined structs.
- *HLSL Shaders:* A shader include file (.cginc) was created as the interface between the prototype system and reactive shaders relying on it. Multiple sampling functions were defined that allowed for flexible use of the system in sample use cases, such as ray marching and vertex displacement.
- *Profiler:* Unity's profiler allowed for observing and recording frame times for specifiable chunks of code which was used to verify the integrity of the benchmarking application.
- *Web Request API:* Benchmark data could be sent from the application to an externally-hosted Google Form (see below) allowing for a wider range of surveyable devices.

2) Google Forms:

Benchmark results could be collected and sorted using Google Forms, which could then be exported to various useful file formats.

B. Entity Management System (EMS)

The EMS is responsible for managing the relationships between the vector fields and the entities affecting them.

1) Core

The Core class of the EMS is a static class containing the base logic for storing and managing the various scene components. It also defines the logic for the texture generation process (see below).

2) Scene Components

Scene components refer to instances of classes that exist in the game world and adhere to Unity's global event execution order.

- *Controller:* Only one instance of Controller needs to exist in each scene. Controller is responsible for initialising and updating Core at a fixed timestep.
- *Volume:* A Volume is an independently orientable cuboid that represents the bounds of a volume texture.
- *Source:* A Source represents a point and radius in world space and acts as a "site" for generating Voronoi diagrams [7].
- *Visualiser:* The Visualiser uses the ray marching algorithm proposed in [6] and the ray-box intersection algorithm proposed in [5] to present a volumetric visualisation of the vector field. The density of each voxel is determined by its alpha channel.

C. Texture Generation Process

1) Settings: The Settings class exposes the following generation parameters:

- *Resolution:* The 3D resolution of the volume texture.
- *Use Decay:* Whether the system interpolates between the previous and current frames of the volume texture.
- *Decay Speed:* The speed at which voxels of the texture decay if "Use Decay" is true.
- *Use Brute Force:* Whether the system uses the Brute Force Algorithm (BFA) of JFA to render the volume texture (see "Vector Field Generation" below).
- *Time Step:* The fixed time between render passes of the volume texture.

2) Initialization: The volume textures are initialized with the default parameters as assigned by Settings. Two volume textures are allocated in memory – one for the current render pass and one to store the previous render pass for decay.

3) Generation: This is the central process for rendering the volume texure, and can be divided into the following compute shader passes:

a) Sentinel Pass: The first pass over the volume texture applies a default colour of (0.5, 0.5, 0.5, 0) to each voxel.

b) Vector Field Generation: This is the main process of drawing the volume texture for the current frame. There are two algorithms for rendering the texture, determined by the "Use Brute Force" bool in Settings. First each Source is converted to a seed – a struct storing the world-space position and effect radius of the source. Regardless of which algorithm is used, a compute buffer is created containing these seeds and is passed to the GPU. The "Vector Field Generation" step is skipped if there are no Sources bounded by the focal Volume to avoid passing a zero-length compute buffer to the GPU.

- *Brute Force Algorithm:* For each voxel, the closest seed's world-space position s is found for the voxel's position p where the distance between p and s is less than the seed's radius r. The vector v is calculated:

$$v = p - s \tag{1}$$

The vector n is encoded as a colour and stored in the voxel's RGB channels:

$$n = \frac{v}{\|v\|} \tag{2}$$

The scalar d is stored in the alpha channel:

$$d = 1 - \frac{\|v\|}{r} \tag{3}$$

- *Jump Flooding Algorithm:* The jump flooding approach to rendering the volume texture requires multiple passes. First, the largest radius r_{max} of the seed buffer is found so that each seed's radius can be represented as a fraction (0 to 1) of r_{max}. Fig. 1 demonstrates the jump flooding process for a volume texture with a resolution of 64^3.

  Seeding pass: The seeding pass iterates over each seed in the compute buffer, calculates the nearest voxel coordinate and assigns the voxel coordinate as a fraction of the volume texture's resolution to the RBG channels. The seed's radius as a fraction of r_{max} is stored in the alpha channel.

  JFA passes: JFA is applied to the texture at this step. Beginning with a step size of half the texture's resolution, a JFA pass is applied and the step size is halved again until it reaches (1, 1, 1).

  Conversion pass: This pass iterates over every voxel of the volume texture. If the alpha value is greater than 0, the local position of the nearest seed is decoded from a colour and converted to world space. An identical operation as shown in "Brute Force Algorithm" is performed using the decoded vector as the world-space seed position to calculate the final voxel colour.

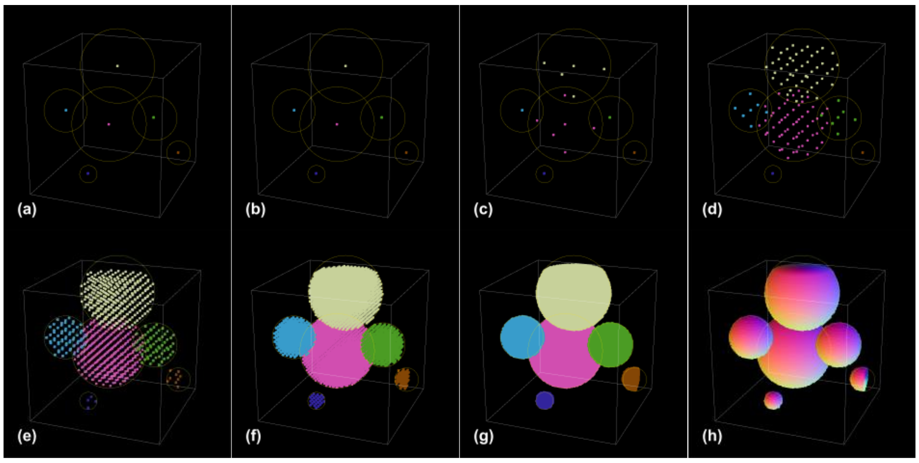


Fig. 1. Visualisation of the JFA vector field generation process for a volume texture with a radius of 64^3. (a) The seeding pass. (b) – (g) JFA passes with iteratively halved step sizes of 32, 16, 8, 4, 2, 1 respectively. (f) Conversion pass.

c) Decay Pass: The alpha value of each voxel in the previously rendered volume texture is interpolated towards 0 as a function of "Decay Speed" and the time since the last frame.

d) Blend Pass: The RGB values stored in the current volume texture and the texture generated in the previous frame are interpolated between as a function of the alpha values to smooth the vector field and allow Sources to produce residual forces.

D. Programmable Shader Interface

A shader include file (.cginc) was created to provide developers with an accessible interface for creating reactive shaders. The file contains functions that perform necessary matrix conversions and sample raw colours or decoded vectors from the volume texture. The interface was used for creating the Visualiser and a simple vertex displacement surface shader.

E. Benchmarking

An executable Windows program was developed to stress-test devices running the prototype volume texture generation system. The benchmark program conducts several tests, each consisting of multiple phases. Each test defines independent and constant variables (see Table 1). The independent variables would be incremented by a given function after each phase. During each phase, several Sources ("Source Count" in Table 1) of arbitrary sizes and velocities would travel through a Volume for the prototype system to render.

TABLE I. TABLE 1. TEST VARIABLES (SEE "SETTINGS" FOR DESCRIPTIONS).

Test	**Variable**				
	Resolution	***Source Count***	***Time Step***	***Use Brute Force***	***Use Decay***
Fidelity	I	20	0.0	False	True
Decay	I	20	0.0	False	I
JFA (Source)	64, 64, 64	I	0.0	I	True
JFA (Resolution)	I	50	0.0	I	True
Time Step	64, 64, 64	20	I	False	True

Quarter-Resolution	I	20	0.0	False	True

[a] I = Independent, constant variables display the value.

Over the course of the benchmark process, several Sources would travel through the focal Volume as the prototype system ran. This process would be repeated for each incremented independent variable. At the end of each phase, the test name and variable values are written to a spreadsheet. The benchmark program was executed on several devices and the following device-specific variables were also collated:

- *Device Unique Identifier:* An identifier used to identify the specific device for each set of phase results.
- *Graphics Device Name:* The name of the GPU used to execute the benchmarking program.
- *Processor Type:* The name of the CPU used to execute the benchmarking program.
- *Time:* The average time to execute the volume texture rendering function for the phase.
- *FPS:* The average frames per second of the program during the phase.

IV. Discussion

A. Results

The following benchmark tests were conducted on 4 devices with the following specifications:

TABLE II. Device Specifications.

Device 1	
Graphics Device Name	NVIDIA GeForce RTX 2080 with Max-Q Design
Processor Type	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Device 2	
Graphics Device Name	NVIDIA GeForce GTX 970
Processor Type	Intel(R) Core(TM) i5-4690 CPU @ 3.60 GHz
Device 3	
Graphics Device Name	AMD Radeon(TM) Vega 6 Graphics
Processor Type	AMD Ryzen 3 2300U with Radeon Vega Mobile Gfx
Device 4	
Graphics Device Name	Intel(R) HD Graphics 520
Processor Type	Intel(R) Core(TM) i5-6300U CPU @ 2.40Ghz

1) *Fidelity*

The fidelity test recorded the frame times for each device to assess the impact of increasing resolution on the overall performance of the prototype system. A resolution value of 0 indicated that the entity management system was operating, but the texture generation process was skipped. This acts as the “game” running without the volume rendering system operating and works as a baseline value for comparison.

The results are shown in Fig. 2. On the higher-end devices (1 and 2), increasing the resolution has little impact on performance. However, on the lower end devices (3 and 4), changes to the resolution resulted in a non-uniform increase in frame time.

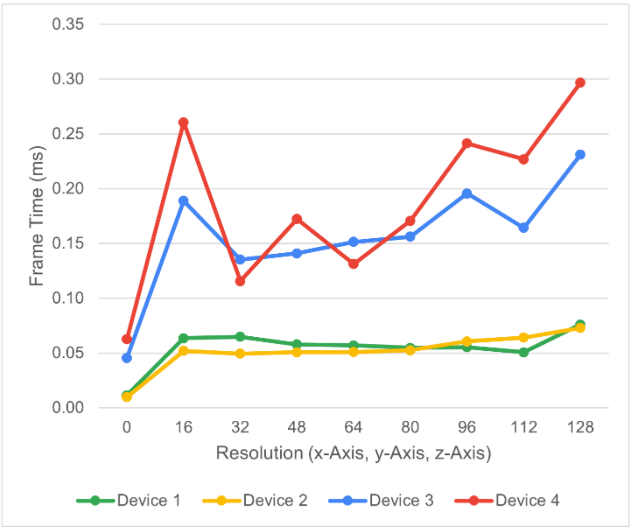


Fig. 2. Fidelity test results.

2) *Decay*

The decay test recorded the frame times for each device at a steadily increasing resolution with “Use Decay” set to both true and false. When set to false, two rendering passes of the volume texture are skipped.

The results are shown in Fig. 3. The results showed a similar trend to the fidelity test regarding stability on the lower-end devices. However, for many resolutions, decaying the texture caused considerable spikes in frame time. The impact for the higher end devices were largely negligible.

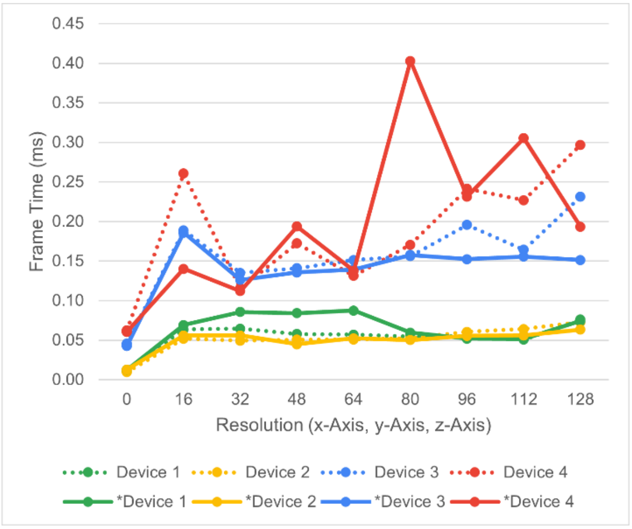


Fig. 3. Decay test results.

3) *JFA (Source)*

This test compared the performance of BFA and JFA vector field generation as the number of Sources increased.

The results are shown in Fig. 4. Device 4 showed noticeably better performance using BFA, while the other devices showed no notable difference in frame time.

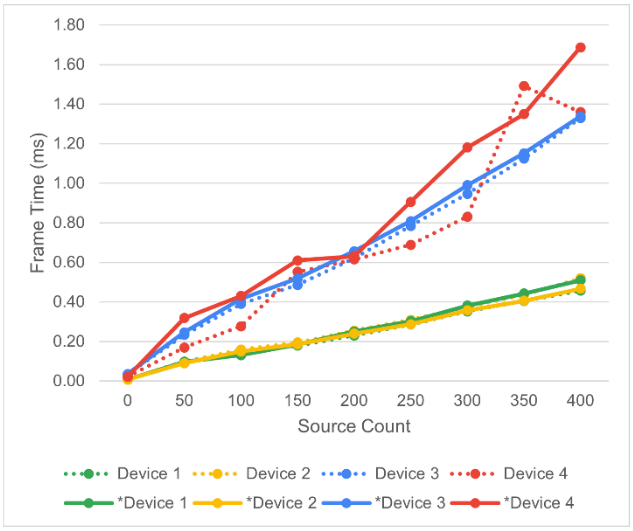


Fig. 4. JFA (Source) test results.

4) *JFA (Resolution)*

This test further compared the performance of the two vector field generation algorithms at varying resolutions.

The results are shown in Fig. 5. Similar spikes in performance for the low-end devices appeared, however the impact of using JFA show the same result as the JFA (Source) test.

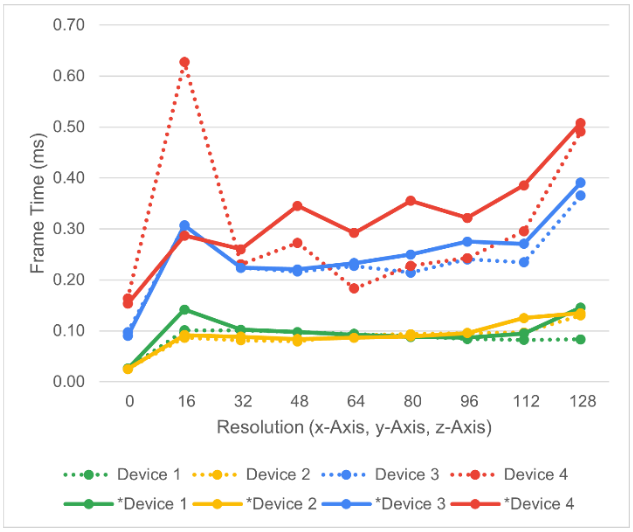


Fig. 5. JFA (resolution) test results.

5) *Time Step*

The time step test recorded the overall performance of each device while varying the time between volume texture rendering processes. Because the frame times should not be impacted, the dependent variable was FPS.

The results are shown in Fig. 6. The most noticeable feature of the set is the spike in performance when device 2 has a time step greater than 0. This device is also the only desktop system of the benchmarked devices. The other devices showed a less significant but similar trend, where having a time step greater than ~0.5 gave diminishing returns.

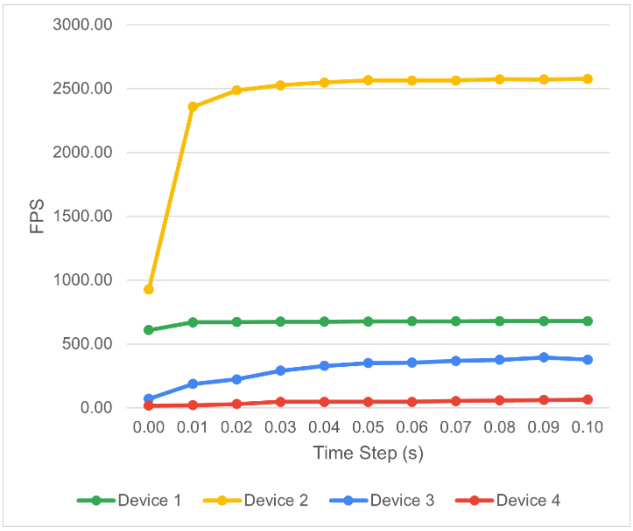


Fig. 6. Time step test results.

6) *Quarter-Resolution*

Some games emphasise the importance of some axes over others. The quarter-resolution test aimed to find the impact of reducing the resolution of a single axis of the volume texture to ¼ the value of the remaining two to optimise the performance of such games.

The results are shown in Fig. 7. As with the fidelity test, the lower-end devices demonstrated an unstable trend with resolution variance. All devices failed to show an increase in performance relatively equal to the reduction in the total number of voxels rendered.

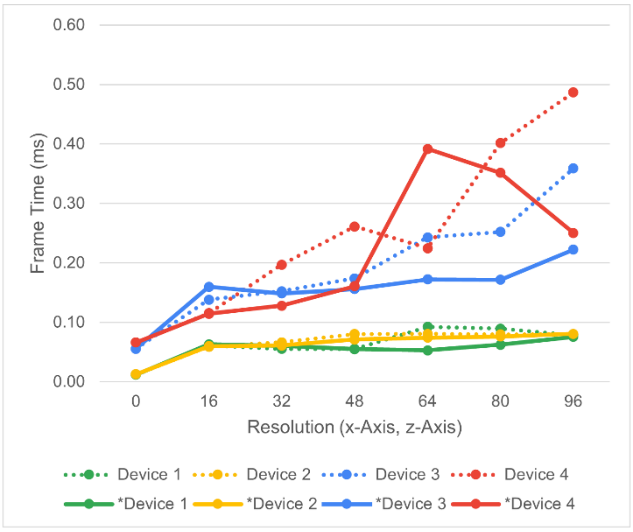


Fig. 7. Quarter-resolution test results.

From the data gathered from the benchmarking process, settings profiles were created for each device, listed in Table 3. These custom profiles were designed for the prototype system to generate textures of optimal quality while still maintaining reasonable frame times.

TABLE III. DEVICE-SPECIFIC SETTINGS PROFILES

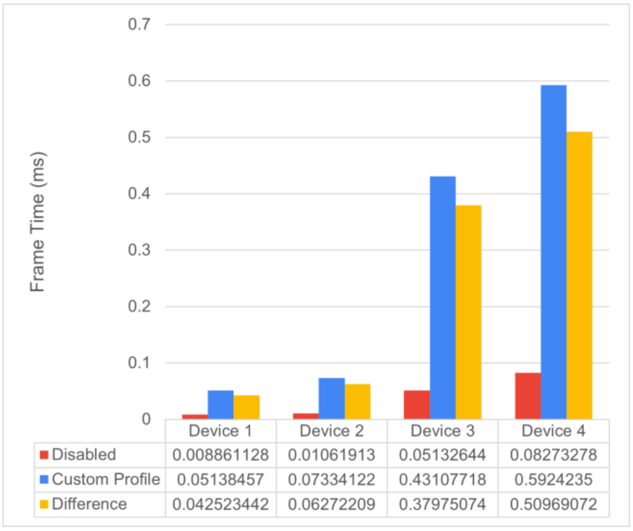
Device 1	
Resolution	112, 112, 112
Use Decay	True
Use Brute Force	False
Time Step	0.01

Device 2	
Resolution	80, 80, 80
Use Decay	True
Use Brute Force	False
Time Step	0.02

Device 3	
Resolution	80, 80, 80
Use Decay	True
Use Brute Force	True
Time Step	0.03

Device 4	
Resolution	64, 64, 64
Use Decay	True
Use Brute Force	True
Time Step	0.1

The frame time for each of these profiles were recorded running the benchmark program on the respective device. The results are displayed in Fig. 8.



	Device 1	Device 2	Device 3	Device 4
🟥 Disabled	0.008861128	0.01061913	0.05132644	0.08273278
🟦 Custom Profile	0.05138457	0.07334122	0.43107718	0.5924235
🟨 Difference	0.042523442	0.06272209	0.37975074	0.50969072

Fig. 8. Device-specific settings profile benchmark results.

The frame time difference between “all-features” and “collision disabled” for a similar scene in [4] is 0.388ms. Considering that enabling collisions on the grass is similar in effect to enabling the prototype system proposed in this paper, a frame time difference of near or less is a positive result at this stage of the project. Additionally, this project offers expandable environment interaction and can be used for multiple shaders with near-identical performance. The frame times (of the higher-end devices in particular) indicate the integrity of the prototype system as a solution for developing reactive shaders for game environments.

B. Limitations

As the proposed system is in the prototype stage, most underdeveloped aspects can be iterated upon in future (see “Future Improvements” below).

There were several limitations with the benchmarking process. The main limitation was the accessibility of devices for benchmarking. Four devices ran the benchmark program, however, that is not a large enough volume of responses to accurately portray the performance variations between different devices. Further, because of limited development resources, the benchmarking program was only built to run on devices with Windows OS. Other common operating systems such as Mac OS and Linux were absent from the benchmark responses.

C. Future Improvements

The most significant issue identified through the benchmarking process was the near identical performance of the brute force and JFA vector field generation processes despite the difference in time complexity outlined in [7]. Future work should be focused on exploiting the greatest efficiency from JFA. Possible directions outlined in [8] should be investigated, specifically variants 1 and 2.

It is possible that the potential of JFA was not reached due to unoptimized use of parallelism offered by the GPU. All compute shader kernels passing over the entire volume texture had a constant thread count value of (8, 8, 8). Future benchmarking should focus on the performance impact of different thread count values and work group sizes.

The brute force algorithm for generating the vector field assesses each Source sequentially to find the nearest where the current voxel lies within its radius. Because of this, it is possible to simply substitute the Source’s position with the closest point on the line segment between the Source’s current position and position in the previous frame. This prevents empty space being rendered between the previous and current positions of a Source if it is moving quickly. However, because the JFA process only initially iterates over the pixels containing a seed, this same approach is not possible. Finding a method to seed the pixels along a Source’s velocity since the previous frame would create more continuous trails while maintaining the benefits of the JFA approach.

The benchmarking indicated that devices with similar specifications to device 4 run relatively poorly with the prototype system. Further work should also aim to introduce more performance enhancing settings, such as using BFA with a capped maximum number of Sources.

V. Conclusion

Many methods exist for creating reactive real-time environments for games. The proposed prototype system aims to utilise modern GPUs to efficiently generate and store vector fields as volume textures. This solves many issues with current approaches and streamlines development by offering a central API for creating reactive shaders. The various factors contributing to the performance of the prototype were assessed on different devices. The benchmark results showed that the prototype system offered greater scalability and lower frame times than identified existing solutions. The potential improvements proposed can potentially solve most of the weaknesses that emerged during testing. The prototype system shows clear potential as a comprehensive and effective solution for creating reactive environments.

References

[1] Epic Games, Inc. n.d., *Creating a Fluid Surface with Blueprints and Render Targets*, Unreal Engine 4 Documentation, viewed 3 February 2021, <https://docs.unrealengine.com/en-US/RenderingAndGraphics/RenderTargets/BlueprintRenderTargets/HowTo/FluidSurface/index.html>.

[2] Epic Games, Inc. n.d., *Vector Fields*, Unreal Engine 4 Documentation, viewed 10 February 2021, <https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/VectorFields/index.html>.

[3] Galbis, A. & Maestre, M. 2021, *Vector Analysis Versus Vector Calculus*, 1st edn, Springer Science+Business Media, LLC, New York, NY.

[4] Jahrmann, K. & Wimmer, M. 2017, 'Responsive real-time grass rendering for general 3D scenes', *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, New York, NY, pp. 1–10.

[5] Majercik, A., Crassin, C., Shirley, P. & McGuire, M. 2018, 'A ray-box intersection algorithm and efficient dynamic voxel rendering', *Journal of Computer Graphics Techniques*, vol. 7, no. 3, pp. 66–81.

[6] Perlin, K. & Hoffert, E.M. 1989, 'Hypertexture', *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, vol. 23, no. 3, pp. 253–262.

[7] Rong, G. & Tan, T. 2006, 'Jump flooding in GPU with applications to Voronoi diagram and distance transform', *Proceedings of the 2006 Symopsium on Interactive 3D Graphics and Games*, Association for Computing Machinery, New York, NY, pp. 109–116.

[8] Rong, G. & Tan, T. 2007, 'Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams', *4th International Symposium on Voronoi Diagrams in Science and Engineering*, pp. 176–181.