Lavie Shaked Golan

Independent Developer, July 7th, 2024

Using the Jump Flood Algorithm to Dilate Velocity Maps in the application of Believable High Range High Fidelity Motion Blur

Abstract

This paper describes a method for simulating motion blur phenomenon for real time application by making use of the jump flood algorithm for velocity map dilation. I demonstrate results on video game scenes rendered and reconstructed in real time on NVIDIA rtx 2070. The technique is faster than previous known methods and is equally robust, while providing higher fidelity. This technique was made to be provided as a proprietary effect by the Godot game engine for its future releases.

1 Introduction and Related Work

A motion blur effect simulates a real phenomena inherit to cameras and shutter speed, with photographed elements appearing smeared and blurred in their direction of movement. The blur amount correlates directly to the velocity of the object relatively to the camera's view times the shutter speed, to put simply. To simulate that effect in digitally rendered scenes, there exists multitude of methods with varying capabilities for both real time applications and offline rendering.

1.1 previous methods

In McGuire et al.'s [2012] paper, which is a staple work on the subject, was described a reconstruction filter technique that used tile based dominant velocity buffers to dilate a high resolution velocity buffer, and use them when generating a believable motion blur post process effect.

His approach had a complexity of O(kn/m) with k being the radius of maximum velocity range captured by the reconstruction filter in pixel space, and m being the amount of parallel processing units available, an example being the many cores in modern GPU's.

1.2 Jump Flood Algorithm

The jump flood algorithm is a data processing pattern common in creations of Signed Distance fields and Voronoi diagrams. It is popular for its ability to perform complex comparisons over large data sets at the complexity of O(log(n)). For that reason its the best and only plausible algorithm with which to generate high resolution high quality distnace related maps in realtime applications, making it the best candidate for established post process effects such as soft shadows and signed distance functions (used for high quality sillouhette rendering) in those applications.

Jump Flood Algorithm Overview the algorithm works by iterating on a data set of size w * h $\log_2(\max(w, h)) + 1$ times. For each data point, sample in 8 directions described by this kernel: (0, 1), (1, 0), (0, -1), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1), multiplied by pow(2, iteration_index - 1). This simply ensures that every pixel in the resulting data set is aware and have selected the most fit data point from the original data set, and is pointing to it.

Limitations Of the Jump Flood Algorithm for its simplicity and efficiency the Jump Flood Algorithm introduces limitations revolving mainly around the conditions driving it. In order to achieve consistent and smooth results, the sample preferences needs to stay simple and continuous throughout, aswell as stay the same for each of the passes, as it relies on sensitive symmetry for a smooth coverage and to produce correct results. Both applications described above, being the signed distance functions and voronoi diagrams can be both described simply as a single distance comparison for each pixel until all pixels are compared.

1.3 Personal Preference

In McGuire et al.'s [2012] paper, they dilate the velocity buffer in both directions by $||V|| / 2$. This is a common practice as is both yields a better base for blurring, and it allows us to require half the dilation radius for the same velocity. The issue I present with this is that at rapid velocity changes, the dilation would often extend beyond the object's final position, producing blurring outside its apparent sillouhete and against viewer intuition. I propose that this exact problem is also felt to a degree at less extreme scenarios and could be a leading cause to distain and reported motion sickness by the viewer from this effect in popular media such as video games and other realtime applications. For that reason I chose to follow a retrospective velocity blur principle, as I believe it is worth the extra required dilation radius and elaborate blending work to eliminate the added ghosting as a result of the sillouhette's leading edge being left bare.

2 Algorithm

2.1 oveview

The method in question takes in a velocity buffer V, and for p passes transfers data from two high resolution buffers A and B in the direction A -> B if (p % 2 > 0) else B -> A, with the first pass transferring the data to A from V instead of B. This is following the Jump Flood Algorithm described in the section above, with a few caveats being as followed: 1) I do not iterate as many times as necessary to reach full coverage of the image buffer, and instead only enough to reach coverage of K, being the max velocity length to dilate 2) I use complex sets of conditiions that take depth and velocity directionality into account, some are non-continuous 3) I add the option to multiply all step sizes by m, sacrificing some accuracy while decreasing range complexity by a logarithmic amount 4) At the last pass first discard values that do not have sufficient fitness values, and on the ones left i perform backtracking of the dilation to mitigate the perpendicular bleed of velocities.

Motivating Analysis The key idea behind this velocity dilation method is that it converts an O(kn) or an O(kn/m) [McGuire et al. 2012] algorithms into an O($\log_2$(kn / m)) algorithm. Additionally, while McGuire et al.'s 2012 methods uses n / k tiles resolution for the final velocity dilation buffer, I get to keep my dilation buffers at full resolution, allowing for maximum amount of detail and fidelity. In addition, the dilation of velocities stay within the the ranges necessary for those velocities, with no prior clamping of velocity values being required, meaning k is just a maximum, and the final dilation radius ends up as small as the dominant relevant velocity's length. This in turn minimizes velocity bleed in the direction of the velocity. In addition, I can make the dilation algorithm be aware of the depth of elements, to prevent background elements affecting foreground elemens. Lastly, the implementation allows for as many passes as necessary, with each pass being identical, which means on modern graphics pipelines, these passes can be made in quick succession one after another, with no rebinding of buffers, and only the push constants need to be updated to let the shader know which pass index it is on.

2.2 Buffers

velocity_sampler (V) is a sharp input velocity buffer in ndc space. depth_sampler (Z) is a sharp input depth buffer in ndc space. Note that both V and Z have been perpective divided prior to this. All buffers are retrieved after the transparent pass and before post processing passes like bloom and FXAA.

I also generate two R32G32B32A32_SFLOAT full resolution buffers A and B.

2.3 Dilation Passes

The algorithm follows the description above, and this is the glsl code for it:

```
const int kernel_size = 8;

const vec2 check_step_kernel[kernel_size] = {
	vec2(0, 0),
	vec2(1, 1),
	vec2(0, 1),
	vec2(-1, 1),
	vec2(1, 0),
	vec2(1, -1),
	vec2(-1, 0),
	vec2(-1, -1),
	vec2(0, -1),
};

// near plane distance
float npd = 0.05;

vec4 get_value(bool a, ivec2 uvi, ivec2 render_size)
{
	if ((uvi.x >= render_size.x) || (uvi.x < 0)  || (uvi.y >= render_size.y) || (uvi.y < 0))
	{
		return vec4(-1, -1, 0, 1);
	}

	if(a)
	{
		return imageLoad(buffer_a, uvi);
	}
	return imageLoad(buffer_b, uvi);
}

void set_value(bool a, ivec2 uvi, vec4 value, ivec2 render_size)
{
	if ((uvi.x >= render_size.x) || (uvi.x < 0)  || (uvi.y >= render_size.y) || (uvi.y < 0))
	{
		return;
	}
	if(a)
	{
		imageStore(buffer_a, uvi, value);
		return;
	}

	imageStore(buffer_b, uvi, value);
}

// Motion similarity
// ----------------------------------------------------------
float get_motion_difference(vec2 V, vec2 V2, float parallel_sensitivity, float perpendicular_sensitivity)
{
	vec2 VO = V - V2;
	double parallel = abs(dot(VO, V) / max(DBL_MIN, dot(V, V)));
	vec2 perpen_V = vec2(V.y, -V.x);
	double perpendicular = abs(dot(VO, perpen_V) / max(DBL_MIN, dot(V, V)));
	float difference = float(parallel) * parallel_sensitivity + float(perpendicular) * perpendicular_sensitivity;
	return clamp(difference, 0, 1);
}
// ----------------------------------------------------------

vec4 sample_fitness(vec2 uv_offset, vec4 uv_sample)
{
	vec2 sample_velocity = -uv_sample.xy;

	if (dot(sample_velocity, sample_velocity) <= FLT_MIN)
	{
		return vec4(FLT_MAX, FLT_MAX, FLT_MAX, 0);
	}

	double velocity_space_distance = dot(sample_velocity, uv_offset) / max(FLT_MIN, dot(sample_velocity, sample_velocity));

	double mid_point = params.motion_blur_intensity / 2;

	double absolute_velocity_space_distance = abs(velocity_space_distance - mid_point);

	double within_velocity_range = step(absolute_velocity_space_distance, mid_point);

	vec2 perpen_offset = vec2(uv_offset.y, -uv_offset.x);

	double side_offset = abs(dot(perpen_offset, sample_velocity)) / max(FLT_MIN, dot(sample_velocity, sample_velocity));

	double within_perpen_error_range = step(side_offset, params.perpen_error_thresh * params.motion_blur_intensity);

	return vec4(absolute_velocity_space_distance, velocity_space_distance, uv_sample.z, within_velocity_range * 
within_perpen_error_range);
}

bool is_sample_better(vec4 a, vec4 b)
{
	if((a.w == b.w) && (a.w == 1))
	{
		return a.z < b.z;
	}

	float nearer = a.z > b.z ? 1 : 0;
	return a.x * b.w * nearer < b.x * a.w;
}

vec4 get_backtracked_sample(vec2 uvn, vec2 chosen_uv, vec2 chosen_velocity, vec4 best_sample_fitness, vec2 render_size)
{
	//return vec4(chosen_uv, best_sample_fitness.x, 0);// comment this to enable backtracking

	int step_count = 16;

	float smallest_step = 1 / max(render_size.x, render_size.y);

	float max_dilation_radius = pow(2, params.last_iteration_index) * params.sample_step_multiplier * smallest_step / 
(length(chosen_velocity) * params.motion_blur_intensity);

	float general_velocity_multiplier = min(best_sample_fitness.y, max_dilation_radius);

	for(int i = -step_count; i < step_count + 1; i++)
	{
		float velocity_multiplier = general_velocity_multiplier * (1 + float(i) /  float(step_count));

		if(velocity_multiplier > params.motion_blur_intensity + 0.2 || velocity_multiplier < FLT_MIN)
		{
			continue;
		}

		vec2 new_sample = uvn - chosen_velocity * velocity_multiplier;

		if((new_sample.x < 0.) || (new_sample.x > 1.) || (new_sample.y < 0.) || (new_sample.y > 1.))
		{
			continue;
		}

		vec2 velocity_test = textureLod(velocity_sampler, new_sample, 0.0).xy;

		if(get_motion_difference(chosen_velocity, velocity_test, params.parallel_sensitivity, params.perpendicular_sensitivity) <= 
params.velocity_match_threshold)
		{
			chosen_uv = new_sample;
			best_sample_fitness.x = velocity_multiplier;
			return vec4(chosen_uv, best_sample_fitness.x, 0);
		}
	}

	return vec4(uvn, best_sample_fitness.x, 1);
}

void main()
{
	ivec2 render_size = ivec2(textureSize(velocity_sampler, 0));
	ivec2 uvi = ivec2(gl_GlobalInvocationID.xy);
	if ((uvi.x >= render_size.x) || (uvi.y >= render_size.y)) 
	{
		return;
	}
	vec2 uvn = (vec2(uvi) + vec2(0.5)) / render_size;

	int iteration_index = params.iteration_index;

	float step_size = round(pow(2, params.last_iteration_index - iteration_index));

	vec2 uv_step = vec2(step_size) * params.sample_step_multiplier / render_size;

	vec4 best_sample_fitness = vec4(FLT_MAX, FLT_MAX, FLT_MAX, 0);

	vec2 chosen_uv = uvn;
```

```
    vec2 chosen_velocity = vec2(0);

    bool set_a = !bool(step(0.5, float(iteration_index % 2)));

    for(int i = 0; i < kernel_size; i++)
    {
        vec2 step_offset = check_step_kernel[i] * uv_step;
        vec2 check_uv = uvn + step_offset;

        if((check_uv.x < 0.) || (check_uv.x > 1.) || (check_uv.y < 0.) || (check_uv.y > 1.))
        {
            continue;
        }

        if(iteration_index > 0)
        {
            ivec2 check_uv2 = ivec2(check_uv * render_size);

            vec4 buffer_load = get_value(!set_a, check_uv2, render_size);

            check_uv = buffer_load.xy;

            step_offset = check_uv - uvn;
        }

        vec4 uv_sample = vec4(textureLod(velocity_sampler, check_uv, 0.0).xy, npd / textureLod(depth_sampler, check_uv,
0.0).x, 0);

        vec4 current_sample_fitness = sample_fitness(step_offset, uv_sample);

        if (is_sample_better(current_sample_fitness, best_sample_fitness))
        {
            best_sample_fitness = current_sample_fitness;
            chosen_uv = check_uv;
            chosen_velocity = uv_sample.xy;
        }
    }

    if(iteration_index < params.last_iteration_index)
    {
        set_value(set_a, uvi, vec4(chosen_uv, best_sample_fitness.x, best_sample_fitness.w), render_size);
        return;
    }

    float depth = npd / textureLod(depth_sampler, uvn, 0.0).x;

    if(best_sample_fitness.w == 0 || depth < best_sample_fitness.z)
    {
        set_value(set_a, uvi, vec4(uvn, best_sample_fitness.x, 0), render_size);
        return;
    }

    vec4 backtracked_sample = get_backtracked_sample(uvn, chosen_uv, chosen_velocity, best_sample_fitness, render_size);

    set_value(set_a, uvi, backtracked_sample, render_size);

    return;
}
```

The main structure of it follows a standard jump flood algorithm pattern, with the caveats mentioned above. The reason behing each one being: 1) it is obvious I only want to be aware of pixels within the range of the largest velocity I want to consider, so last_iteration_index is not bound to the screen's resolution. 2) I am multiplying uv_step by sample_step_multiplier, allowing me to decrease the complexity of the code while achieving a comparable quality result. 3) I compare the sample fitness using as basic conditions as possible. 4) At the last pass I discard all velocities that were assigned in the process of discovery based on their fitness coefficient, and velocities that are left are run through a backtracking algorithm to further filter perpendicular velocity bleed.

A demo project containing a working interactible implementation can be found at https://github.com/sphynx-owner/JFA_driven_motion_blur_demo

3 Results

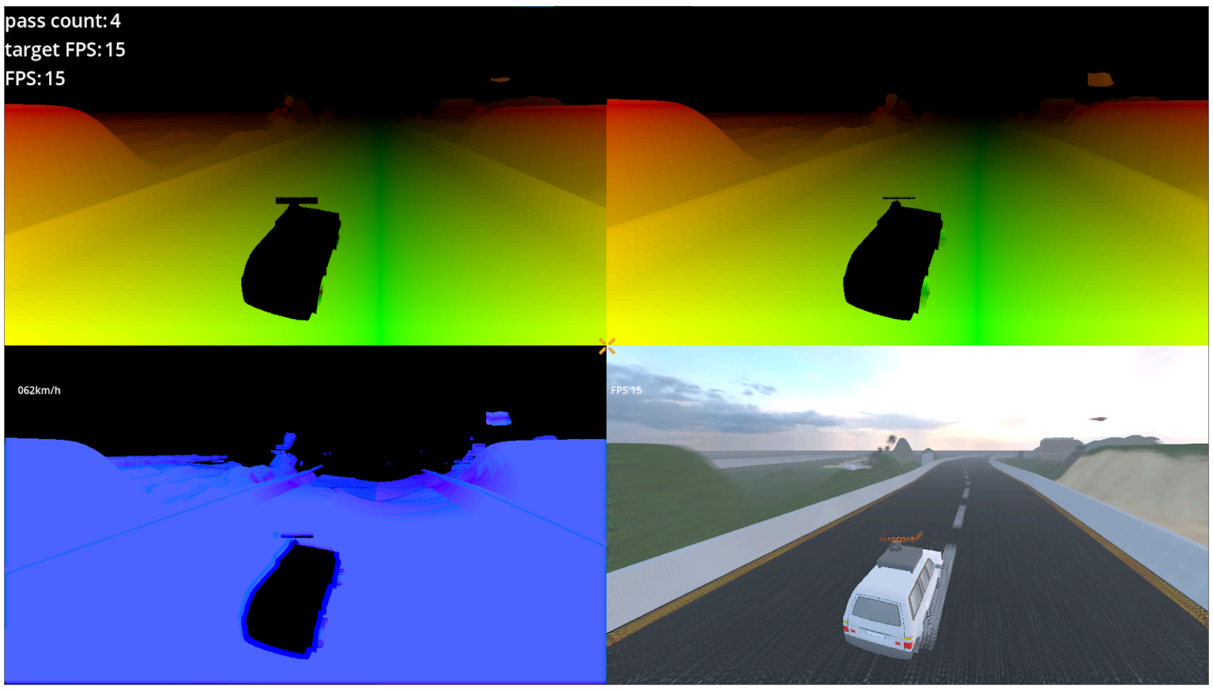

Figure 1: screenshot of in game debug footage using the jump flood velocity dilation. Blur is set at intensity of 1, framerate is limited to 15, and sample step multiplier is set to 1, The image is produced with 4 passes. in the image you can see a vehicle speeding down a highway, to the sides a hilly landscape, and to the top right of the view a plane going to the right. top left image is enhanced absolute input velocity buffer, bottom left image shows enhanced absolute offsets of the velocity buffer generated by the dilation process, top right shows final enhanced absolute velocity map being used.

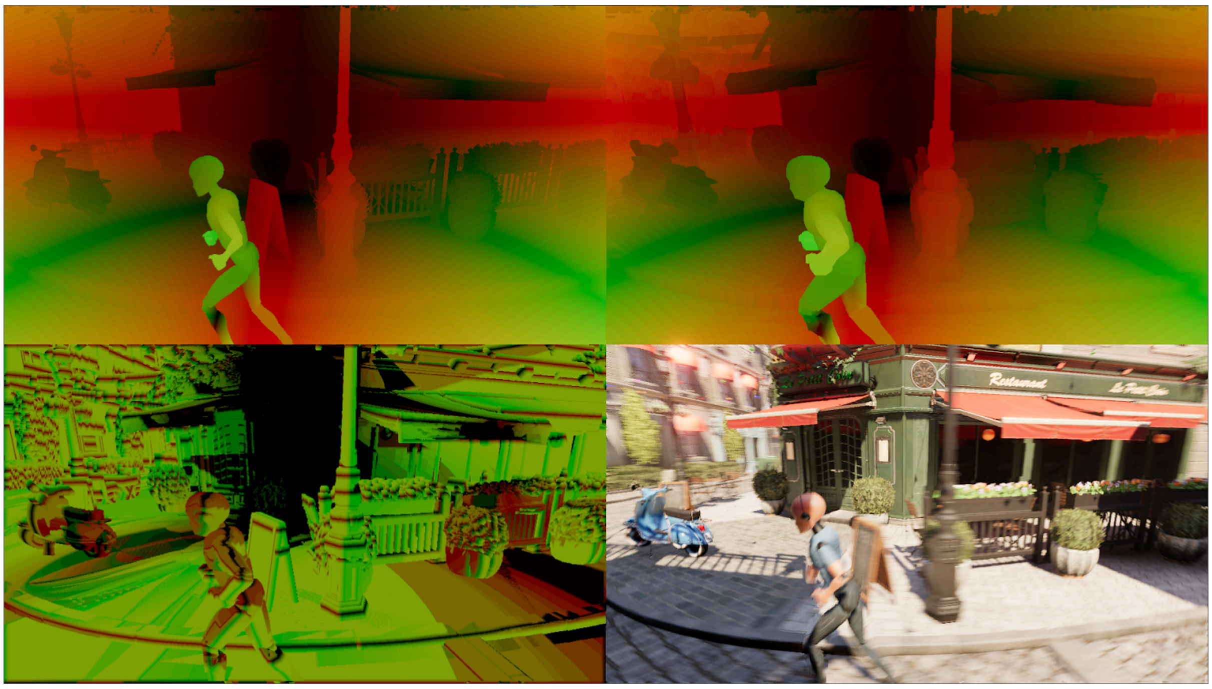

Figure 2: game footage, 1920 x 1080 with 4 passes at ~30 fps, blur intensity at 1, sample step multiplier at 1, a mannequin is running down the street in front of a cafe with a sign, light pole, bushes, and a scooter.

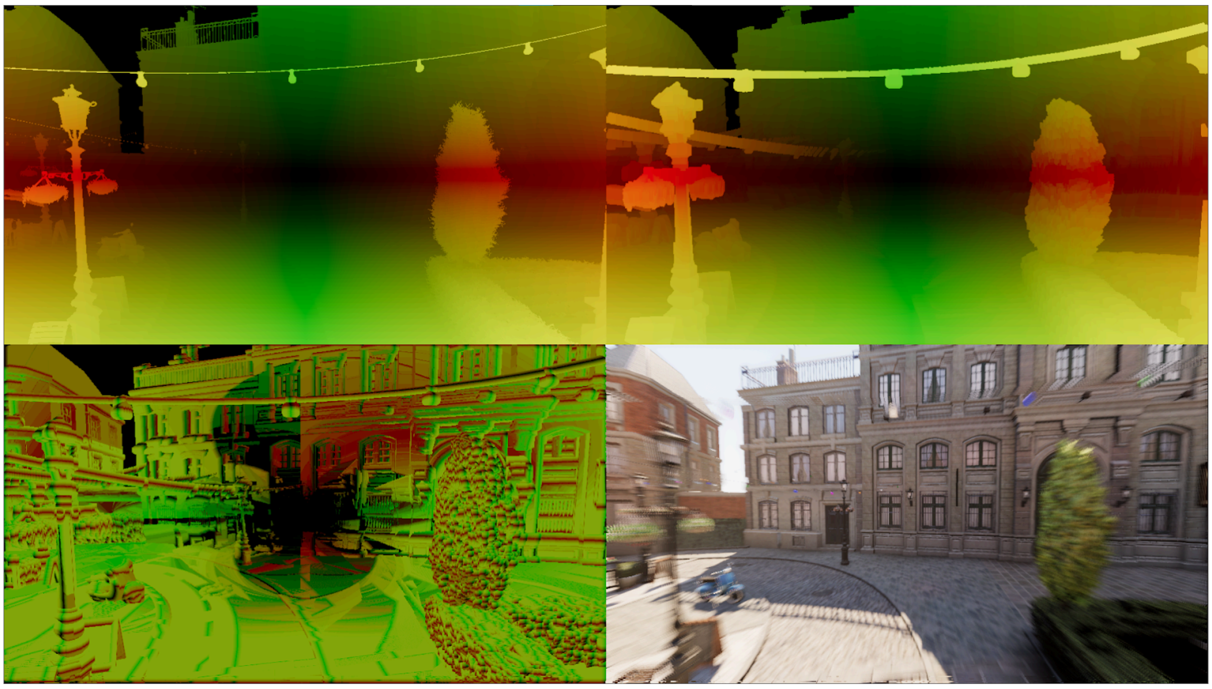

Figure 3: game footage, 1920 x 1080 with 4 passes at ~30 fps, blur intensity at 1, sample step multiplier at 1, flying through a street with a lamp to the left and a bush to the right. note the attention given to each leaf by the algorithm.

Note the extending of the hill to the left and the plane in figure 1. In figure 2 and figure 3, The velocity dilation offset image at the bottom left reveals the level of details in the velocity dilation that can be achieved with this method, especially arond high detail elements like the bushes.

4 Discussion

4.1 Artistic Control

This method for velocity dilation offers control over radius of dilation, as well as control over thresholds including perpendicular error threshold, which controls how much perpendicular offset can a sample have from the velocity it is pointing to, and the parallel threshold, which directly correlates to the blur intensity an artist may desire, and it controls how far down the velocity vector can you include a sample as covered by it. In addition you can control the sample step scale, to allow for greater reach with little to no quality reduction.

You can also expose othe attributes like the motion similarity threshold when performing dilation backtracking, but I did not get to do this in my implementation yet.

4.2 Limitations

The nature of the algorithm makes it really tough to avoid peripheral bleed of velocities, as the strict geometry of the samples in the jump flood algorithm makes it hard to narrow down anything related to the directions of vectors, meaning the connection between a sample's offset and the velocity it stores can be purely heuristic. This is why the perpendicular threshold is present, and it needs to be carefully chosen as to not cause any holes in the dilation, which is shown in figure 4.

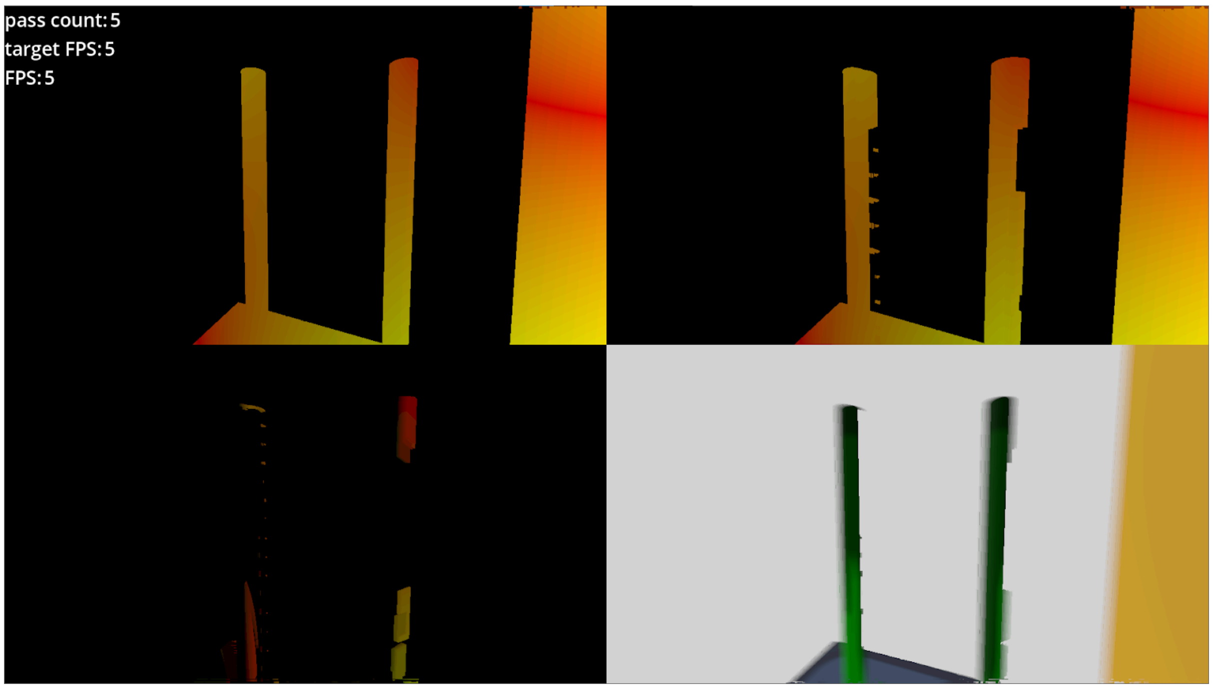

Figure 4: simple mesh examples, camera is looking to the left, and the perpendicular velocity threshold is set to 0.01, which is way lower than recommended.

The lack of strict perpendicular dilation limit causes significant sideways bleed that is proportional to the velocities of the elements in the image, and it can be mostly observed in figure 3 by the widening of the hanging ornaments' silouhettes, as they are moving upwadrs, but also get wider to the sides.

Another obvious example is with the railing in figure 5, which bleeds onto the car at high enough velocities.

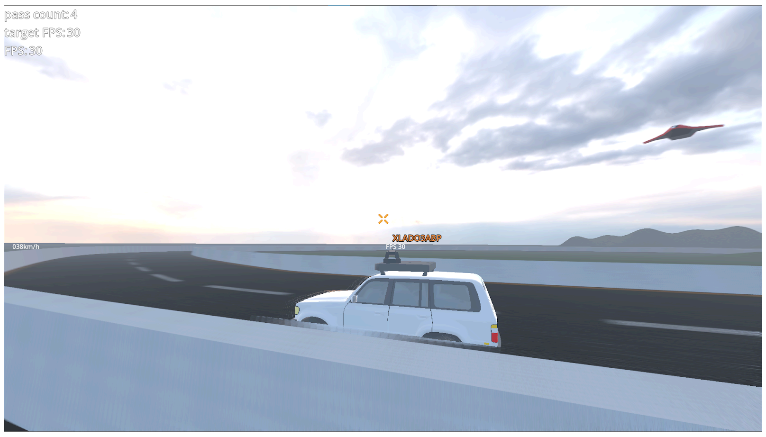

Figure 5: game footage of car speeding down highway, showcasing the peripheral bleed caused by the velocity dilation at even optimal threshold values.

In order to mitigate this, I have introduced the dilation backtracking on the last pass, which samples from each pixel back in the direction of the velocity dilated to it, to see if a similar velocity can be observed within a sensible range, at which point if no similar velocity was found, return the original velocity at that pixel, and if a sufficient velocity was found, return that position instaed.

This leads to significant improvements on the quality of the result, and is able to neutralize most if not all artifacts regarding peripheral dilation as can be seen in figure 6.

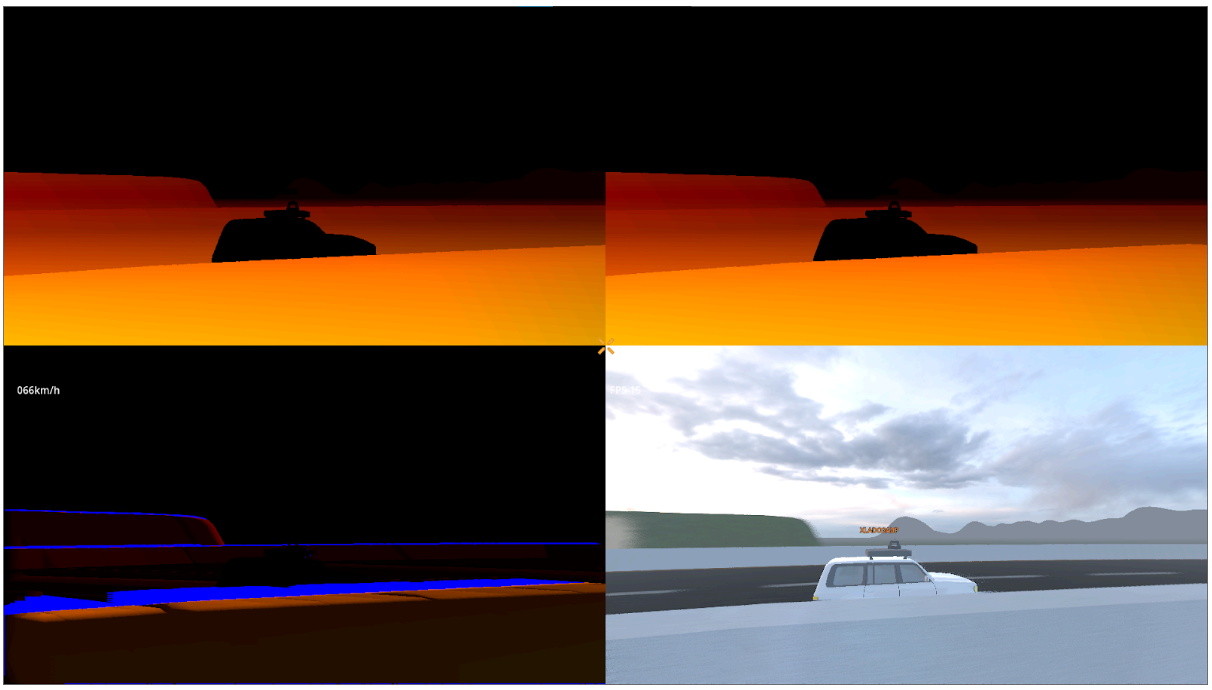

Figure 6: game footage of car speeding down highway, at the bottom left, the blue sections indicates samples that have been backtracked to the original velocity value at those pixels, resulting in a crisp edge under the same conditions as figure 5

Though some attention still needs to be given to the sensitivity of the backtracking check, and the amount of iterations to check for a fit velocity sample, and while not visible anymore, the peripheral bleeding of velocities can still affect the final result in case of severy overflow causing large areas of the image to revert entirely to the original velocity values, degrading the quality of the blur significantly.

Another limitation is the lack of control over the sampling ranges, as the only choises are powers of 2. This can be worked around by using the sample step multiplier to reach higher precision when selecting max dilation radius, at the small sacrifice of some accuracy in the final result. Moreover, due to the dynamic range of dilation in the final result, the actual radius serves as a max value and does not affect the resolution of the result, unless extreme velocities that would utilize the entire range are observed.

4.3 Future Work

This discovery was made as part of my attempts to create a motion blur effect for the later releases of the Godot game engine to arrive with built in. In order to make that happen I would have to improve peripheral aspects such as the motion blur itself, borrowing as much as I can from provable methods and encorporating them into my own.

In general, I believe this approach is still in its infancy, and much more is left to be discovered and perfected. The areas I would shift my focus to would be improving the performance and efficiency of the implementation and reducing artifacts caused by peripheral bleeding.

Acknowledgements I thank the Godot vfx community for getting me interested in the subject, and McGuire for his great paper on the matter.

McGuire, M., Hennessy, P., Bukowsky, M. AND Osman, B., 2012.A Reconstruction Filter for Plausible Motion Blur, 1–8.
https://casual-effects.com/research/McGuire2012Blur/McGuire12Blur.pdf

Douglas, B., 2021, The Jump Flood Algorithm | Visualized and Explained
https://www.youtube.com/watch?v=A0pxY9QsgJE