Motion Blur using Velocity Buffers

Graphics Assignment CSE 409

Team AmarGraphics
2005067 - Masnoon Muztahid
2005074 - Dipanta Kumar Roy Nobo
2005090 - Tawkir Aziz Rahman

August 10, 2025

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)

Introduction to Motion Blur

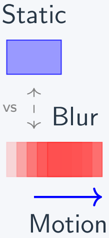
What is Motion Blur?

- Visual artifact from object movement during exposure
- Creates streaking effects along motion direction
- Essential for realistic rendering
- Conveys speed and movement
- Reduces temporal aliasing

Why Important?

- Human vision expects blur
- Fast objects appear choppy without it
- Enhances immersion

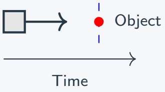
Figure 1: Motion Blur



Real-World vs. CG Motion Blur

Real-World Motion Blur

- ✓ Natural phenomenon
- ✓ Finite shutter speed
- ✓ Continuous exposure
- × No post-control
- × May be unwanted



CG Motion Blur

- ✓ Precise control
- ✓ Adjustable amount
- ✓ Selective application
- × Computationally heavy
- × Needs special techniques

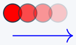
Key Insight: Real-world = continuous integration, CG = discrete sampling

Types of Motion Blur

Motion Blur Types

ObjectMotion

Moving objects



CameraMotion

Moving camera

DeformationBlur

Shape changes

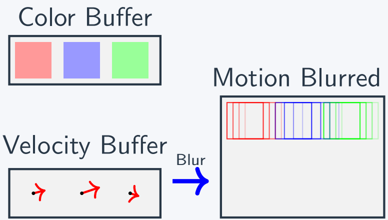
Implementation Techniques:

- Multi-sampling (accumulation)
- Velocity buffers **(our focus)**
- Post-process filters
- Per-pixel motion vectors

What is a Velocity Buffer?

Definition:

- Render target with per-pixel velocity vectors
- Encodes 2D screen-space motion
- RG channels store x,y velocity
- Enables post-process motion blur



Advantages:

- ✓ Single geometry pass
- ✓ Efficient post-processing
- ✓ Quality control
- ✓ Deferred rendering compatible

Storage (RGBA):

- R: Horizontal velocity
- G: Vertical velocity
- B: Depth/unused
- A: Blur mask

How Velocity Buffers Are Computed

Step 1: Vertex Shader

```
layout(location = 0) in vec3 position;
uniform mat4 currentMVP, previousMVP;
out vec4 currentPos, previousPos;

void main() {
    vec4 worldPos = vec4(position,
        1.0);
    currentPos = currentMVP * worldPos
        ;
    previousPos = previousMVP *
        worldPos;
    gl_Position = currentPos;
}
```

Step 2: Fragment Shader

```
in vec4 currentPos, previousPos
    ;
uniform vec2 screenSize;
out vec2 velocity;

void main() {
    vec2 currNDC = currentPos.
        xy / currentPos.w;
    vec2 prevNDC = previousPos.
        xy / previousPos.w;
    vec2 currScreen = (currNDC
        * 0.5 + 0.5) *
        screenSize;
    vec2 prevScreen = (prevNDC
        * 0.5 + 0.5) *
        screenSize;
    velocity = currScreen -
        prevScreen;
}
```

Mathematical Example:

$$
\begin{aligned}
\text{currNDC} &= (0.2, 0.1) \\
\text{prevNDC} &= (0.0, 0.0) \\
\text{screenSize} &= (1920, 1080) \\
\text{currScreen} &= (0.2 \times 0.5 + 0.5) \times 1920 = 1152 \\
&\quad (0.1 \times 0.5 + 0.5) \times 1080 = 594 \\
\text{prevScreen} &= (0.0 \times 0.5 + 0.5) \times 1920 = 960 \\
&\quad (0.0 \times 0.5 + 0.5) \times 1080 = 540 \\
\text{velocity} &= (1152, 594) - (960, 540) \\
&= (192, 54) \text{ pixels}
\end{aligned}
$$

Core Idea: Use current & previous transforms to compute screen-space motion vectors

Velocity Buffer Computation Pipeline

Current & Previous MVP | Compute Velocity | RG16F Buffer

VertexShader → FragmentShader → Store inBuffer

Mathematical Formulation

$P_{curr} = MVP_{curr} \times P_{world}$

$P_{prev} = MVP_{prev} \times P_{world}$

$NDC = P.xy/P.w$

$Screen = (NDC \times 0.5 + 0.5) \times Size$

$V = Screen_{curr} - Screen_{prev}$

Special Considerations

- First frame: velocity $= 0$
- Camera motion: uniform pixel velocity
- Clamp extreme velocities

Buffer Details

Format: RG16F

Channels: R=horizontal, G=vertical

Range: $\pm$1024 pixels

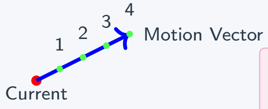
Pipeline Summary: Transform vertices with both current and previous MVP matrices, compute screen-space motion vectors in fragment shader, store as velocity buffer

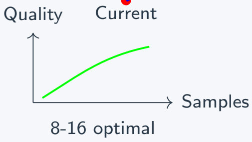
Applying Blur Using the Buffer

Motion Blur Shader:

```
// Post-process motion blur
uniform sampler2D colorTexture, velocityTexture;
uniform float blurScale; uniform int maxSamples;
in vec2 texCoord; out vec4 fragColor;

void main() {
    vec2 velocity = texture(velocityTexture,
        texCoord).xy * blurScale;
    float speed = length(velocity);

    if (speed < 0.5) {
        fragColor = texture(colorTexture, texCoord);
        return;
    }

    velocity = normalize(velocity) * min(speed,
         20.0);
    vec3 result = vec3(0.0);

    for (int i = 0; i < maxSamples; ++i) {
        float t = float(i) / float(maxSamples - 1);
        vec2 coord = texCoord - velocity * t /
             textureSize(colorTexture, 0);
        result += texture(colorTexture, coord).rgb;
    }
    fragColor = vec4(result / float(maxSamples),
         1.0);
}
```

Sampling Strategy:





Key Parameters:

- **blurScale**: Global intensity (0-2)
- **maxSamples**: Quality (4-32)
- **Velocity clamp**: Max blur (10-50px)
- **Threshold**: Skip static areas

Color Averaging Example:

maxSamples = 4
velocity = (0.02, 0.01)

Sample 0 (t=0.0):
coord0 = (0.5, 0.5)
color0 = (0.8, 0.2, 0.1)

Sample 1 (t=0.33):
coord1 = (0.493, 0.497)
color1 = (0.6, 0.4, 0.3)

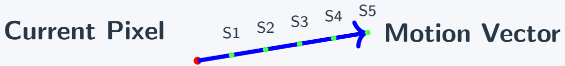
Sample 2 (t=0.67):
coord2 = (0.487, 0.493)
color2 = (0.4, 0.6, 0.5)

Sample 3 (t=1.0):
coord3 = (0.48, 0.49)
color3 = (0.2, 0.8, 0.7)

result = (color0 + color1
+ color2 + color3) / 4

result = (0.5, 0.5, 0.4)

Sampling Along Motion Vector

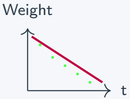


Sampling Methods:

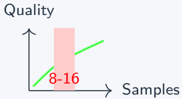
- Fixed step
- Adaptive
- Jittered
- Weighted

Optimal: 8-16 samples

Sample Weights:

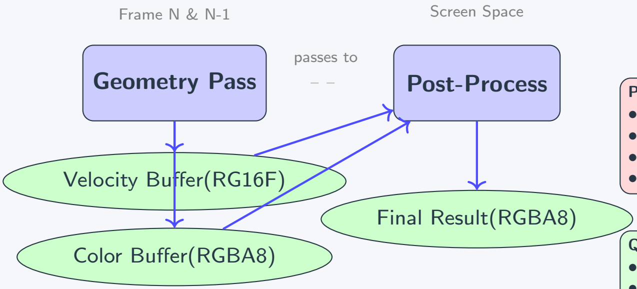


Quality vs Performance:



Key: Sample along motion vector with weights decreasing by distance. 8-16 samples

Complete Shader Pipeline Overview



Performance:
- Half-precision floats
- Adaptive samples
- Early exit threshold
- Tile optimization

Quality:
- Samples: 4-32
- Clamp: 0-50px
- Strength: 0-2
- Noise filter

Velocity Gen:
```
vel = curr - prev;
```

Blur Sample:
```
col += tex(uv-vel*t);
```

Pipeline Summary: Generate motion vectors in geometry pass, then apply directional blur in post-process for efficient real-time motion blur

Pros and Cons of Velocity Buffer Motion Blur

ADVANTAGES

Performance:

- Single geometry pass
- GPU-friendly processing
- No temporal accumulation

Quality:

- Per-pixel motion vectors
- Controllable blur amount
- Handles complex motion

Integration:

- Deferred rendering compatible
- Engine-agnostic approach

DISADVANTAGES

Limitations:

- No sub-pixel accuracy
- Occlusion handling issues
- Memory bandwidth cost

Artifacts:

- Ghosting artifacts
- Edge bleeding
- Velocity discontinuities

Implementation:

- Needs previous frame data
- Complex animated meshes

Motion Blur in Popular Game Engines

Unreal Engine

- Per-object motion blur
- Temporal upsampling
- TAA integration

Unity HDRP

- Camera + object blur
- Quality presets
- VR optimized

CryEngine

- Advanced sampling
- Radial blur support
- Dynamic quality

Common Implementation Features

- Velocity buffer generation
- Post-process blur filter
- Quality/performance settings
- Motion threshold controls
- Temporal stability improvements
- VR/mobile optimizations
- Artist-friendly parameters
- Debug visualization tools

Final Thoughts

Key Takeaways

- **Velocity buffers** provide an efficient solution for real-time motion blur
- **Single-pass rendering** makes them suitable for modern deferred pipelines
- **Post-process flexibility** allows for quality/performance tuning
- **Wide adoption** in commercial game engines proves their effectiveness

Best Practices:

- Use 16-bit float precision
- Implement velocity clamping
- Add temporal stability filters
- Provide artist controls
- Test with various content types

Future Improvements:

- AI-enhanced sampling
- Hardware RT integration
- Advanced temporal filtering
- Mobile/VR optimizations
- Real-time quality adaptation