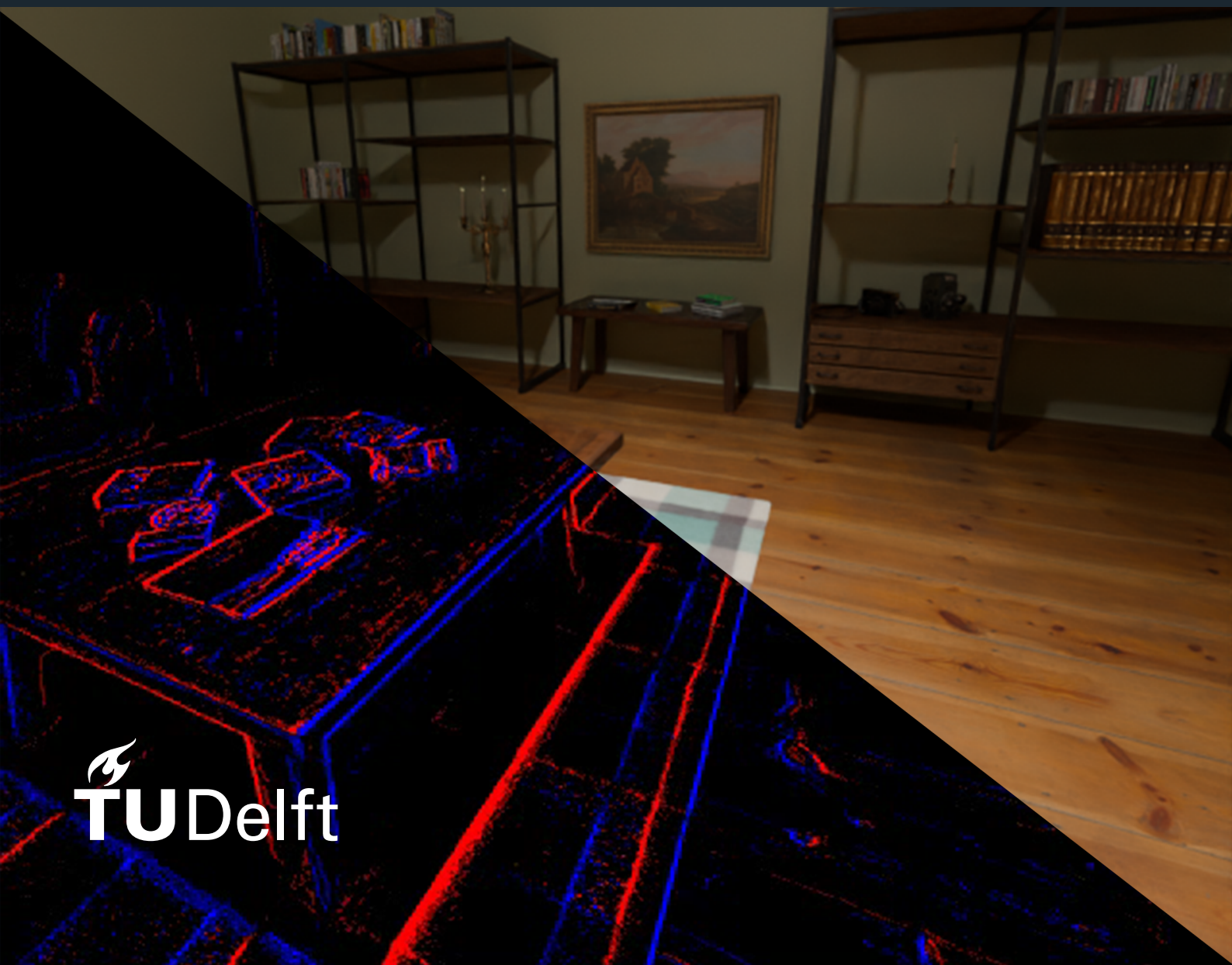
# Continuous-Time Modelling of Pixel Attributes

## The Development of a Continuous Frame Interpolation Method and a Fast, Real-Time Event Camera Simulator

R.J.N. Zwikker



TU Delft

# Continuous-Time Modelling of Pixel Attributes

## The Development of a Continuous Frame Interpolation Method and a Fast, Real-Time Event Camera Simulator

MSc Thesis report

by

R.J.N. Zwikker

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on March 31, 2026 at 11:00

*Thesis committee*:

| | |
|---|---|
| Chair: | Dr.ir. C. De Wagter |
| Supervisor: | Prof.Dr. G.C.H.E. de Croon |
| External examiner: | Dr. N. Tömen |
| Additional Supervisor: | Y. Wu MSc |
| Place: | Faculty of Aerospace Engineering, Delft |
| Project Duration: | Februari, 2025 - Januari, 2026 |
| Student number: | 4994388 |

An electronic version of this thesis is available at `repository.tudelft.nl`.

Faculty of Aerospace Engineering · Delft University of Technology

"Science is made up of so many things that appear obvious after they are explained."

*— Dune, Frank Herbert*




Copyright © R.J.N. Zwikker, 2026
All rights reserved.

# Preface

After this thesis, I look back at an amazing time at the TU Delft. I have learned so much. I have loved my time as a student here and will undoubtedly cherish the memories created here for my entire life.

While not a traditional subject for the Control & Simulation section, I have thoroughly enjoyed working on it. The combination of rendering, optimisation, and computer vision is an amazing intersection of my interests. Through this thesis, I have been able too put many of the skills that I have learned to their full use.

For this work, I want to thank everyone that has helped me complete it. I want to thank Yilun, for his drive to perfect the project and his helpful insights. I will remember our discussions and tangents, discussing increasingly less relevant and more ambitious things to implement! I also want to thank Guido, for his optimism and understanding, when things did not progress as fast as I would have liked and our planning had to be adjusted once again. Additionally, I want to thank Matthijs for his encouragements and unending support. Finally, I want to thank my family and friends, for lending a listening ear whenever Unreal Engine did not want to cooperate or whenever I was rambling endlessly about this thesis!

Thank you!

*Reinier Zwikker,*
*Delft, 29th of January 2026*

# Abstract

There is an increasing interest in Event-based cameras for computer vision purposes. These cameras provide benefits such as high temporal resolution and high dynamic range by capturing the change in brightness instead of the absolute brightness signal. In order to train learning-based behaviours with event-based cameras, it is essential to be able to simulate the output of these cameras in a virtual world. Unlike their traditional frame-based counterparts, there are no real-time event-based camera rendering engines yet. All existing methods of generating events from virtual scenes impose a significant computational burden, simulating events by generating frames at a very high temporal resolution and converting these frames to events. This thesis proposes a novel method of event generation, by continuously interpolating between frames at a low temporal resolution, using auxiliary data from the rendering engine, such as depth and optical flow. In this method, the backwards pixel sampling problem is solved using a triangle grid created between all pixels, that is used to model the continuous response of each pixel between rendered frames. Other than previous methods that used VFI to temporally upsample LFR frames for event generation, this method does not create intermediate frames, instead simulating events directly from the continuous model. This method is implemented on the CPU, where it is shown to correctly interpolate the signal between frames. It is also implemented in Unreal Engine, where it is used to generate events in near real-time.

# Contents

List of Figures vii

List of Tables ix

1 Introduction 1

**I Literature Review and Research Definition 3**

2 Literature Review 4
- 2.1 Optical Flow and Brightness Constancy . . . 4
- 2.2 Video Frame Interpolation Techniques . . . 4
- 2.3 Working Principle and Capabilities of Rendering Engines . . . 5
  - 2.3.1 Rasterisation Rendering Engines . . . 6
  - 2.3.2 Path-Tracing Rendering Engines . . . 7
  - 2.3.3 Modern Ray-Tracing and Unreal Engine . . . 7
- 2.4 GPGPUs: General Purpose computing on Graphical Processing Units . . . 8
- 2.5 Event-Based Cameras and their Uses . . . 8
  - 2.5.1 Working Principle of Event-Based Cameras . . . 8
  - 2.5.2 Challenges of Event-Based Cameras . . . 10
  - 2.5.3 Different Representations of Event Data . . . 11
  - 2.5.4 Event-Based Computer Vision and Neuromorphic Algorithms . . . 11
  - 2.5.5 Event Datasets . . . 11
- 2.6 Working Principle of Event-Based Camera Simulators . . . 12
  - 2.6.1 Current Event-Based Camera Simulators . . . 12
  - 2.6.2 Gap for better simulators . . . 14

3 Problem Statement and Research Definition 15
- 3.1 Research Objective and Research Questions . . . 15
- 3.2 Problem statement . . . 16
- 3.3 Structure and Scope of this Work . . . 17

**II Main Work 18**

4 Methodology 19
- 4.1 Continuous-Time Pixel Attribute Interpolation . . . 20
- 4.2 Mesh-Based Frame Interpolation . . . 22
  - 4.2.1 Edge Collisions . . . 23
  - 4.2.2 Vertex Collisions . . . 25
  - 4.2.3 Chaining Collisions into Control Points . . . 26
  - 4.2.4 Bi-Directional Interpolation . . . 27
- 4.3 Frame Interpolation from Intensity-Time Curves . . . 28
- 4.4 Event Generation from Intensity-Time Curves . . . 28

5 Implementation 29
- 5.1 Implementation of Grid-Based Frame Interpolation . . . 29
  - 5.1.1 Triangle Grid Generation . . . 30
  - 5.1.2 Windowed Collision Search . . . 31
  - 5.1.3 Chaining of Collisions . . . 33
  - 5.1.4 Merging of Forward and Backward Passes . . . 33

5.2 Implementation of Event Generation in Unreal Engine 5 . . . 35
5.2.1 ESIM in Unreal Engine . . . 36
5.2.2 FastESIM using Grid-Based Frame Interpolation . . . 36

6 Results 37
6.1 Qualitative Analysis of Grid-Based Frame Interpolation . . . 37
6.2 Quantitative Analysis of Grid-Based Frame Interpolation . . . 41
6.2.1 Panoramic Case . . . 41
6.2.2 Realistic Room Case . . . 44
6.3 Performance of Unreal Engine Plug-in . . . 49

**III Closure 51**

7 Conclusions 52

8 Recommendations 54

References 58

**Appendix 59**

A Additional Results 60

B Source Code of Implemented Methods 67
B.1 Implementation of Stand-Alone Version . . . 67
B.2 Implementation of Unreal Engine Plug-In . . . 68

C Interfaces 69
C.1 Command-Line Interface . . . 69
C.2 Python Package Interface . . . 69
C.3 Unreal Engine Plug-In Interface . . . 70

# Nomenclature

**List of Abbreviations**

| | |
|---|---|
| AER | Address-Event Representation |
| APC | Active Pixel Sensor |
| API | Application Programming Interface |
| ATFI | Arbitrary-Time Frame Interpolation |
| CPU | Central Processing Unit |
| CS | Compute Shader |
| CTFI | Center-Time Frame Interpolation |
| CUDA | Compute Unified Device Architecture |
| DAViS | Dynamic and Active Pixel Vision Sensor |
| DR | Dynamic Range |
| DVS | Dynamic Vision Sensor |
| FPS | Frames per Second |
| GPGPU | General-purpose computing on Graphics Processing Units |
| GPU | Graphics Processing Unit |
| HDR | High Dynamic Range |
| HFR | High Frame-Rate |
| ISO | Camera Gain (International Organization for Standardisation) |
| LDR | Low Dynamic Range |
| LFR | Low Frame-Rate |
| MAE | Mean Absolute Error |
| ML | Machine Learning |
| NeRF | Neural Radiance Field |
| NN | Neural Network |
| PS | Pixel Shader |
| RDG | Render Dependency Graph |
| RMSE | Root-Mean-Square Error |
| SIMD | Single Instruction, Multiple Data |
| SIMT | Single Instruction, Multiple Threads |
| SISD | Single Instruction, Single Data |
| SNN | Spiking Neural Network |
| SVE | Scene View Extension |
| VFI | Video Frame Interpolation |
| VS | Vertex Shader |

**List of Symbols**

| | |
|---|---|
| $\Delta$ | Triangle |
| $\epsilon$ | Edge |
| $\lambda$ | Distance (along/from edge) |
| $\vec{F}'$ | Backward Optical Flow |
| $\vec{F}$ | (Forward) Optical Flow |
| $C$ | Contrast Threshold |
| $C_n$ | Negative Contrast Threshold |
| $C_p$ | Positive Contrast Threshold |
| $I_t$ | Image Frame at time $t$ |
| $P$ | Pixel |
| $p$ | Polarity |
| $t$ | Time |
| $V$ | Vertex |

# List of Figures


1.1 Comparison between frame-based and event-based cameras. Frames are snapshots of a scene, while events provide sparse data throughout time. . . . . 1

2.1 A 2D illustration of different rendering approaches: Rasterisation is very quick, path-tracing is very accurate, and ray-tracing is a hybrid between the two. . . . . 6
2.2 Comparison between frame-based and event-based cameras. Frames are snapshots of a scene, while events provide sparse data throughout time. In this example, the square moves from side to side. The events only fire at the moving leading and trailing edge of the square. The dashed square is drawn with the events as a indication of its location at the times at which a frame is captured. Positive events are shown in red and negative events are shown in blue. . . . . 9
2.3 Sampling strategies of different event-based camera simulators. Fixed sampling uses a predetermined high frame rate, often above the Nyquist Rate of the fastest movement in the scene. Adaptive samplings adjusts the sampling rate to the current maximum optical flow magnitude in the frame. The upsampling strategy uses a low frame rate input that is temporally upsampled using VFI methods. . . . . 12

4.1 Example of an $I/t$ curve modelling the the change in intensity over time. . . . . 19
4.2 The subtle difference between the forward and backward optical flow fields, $\vec{F}$ and $\vec{F}'$ . . . 20
4.3 A mapping is required between the original frame and the interpolated frame. The correct backwards optical flow needs to be found using the available forward optical flow and depth maps, defined on the pixels in the original frame. The blue mapping is correct, since the forward optical flow points from $P$ points back at $P'$, but the green pair is incorrect, since the optical flow from $!P$ does not point back to $P'$ . . . . . 21
4.4 The continuous frame interpolation method uses a grid of triangles, where each pixel of the image becomes a vertex of the grid. . . . . 22
4.5 A geometric representation of the edge collisions. . . . . 23
4.6 Vertex collisions require special handling, as the edge algorithm cannot find the leaving or entering triangle. The vertex collisions are found by their parallel distance, . . . . . 25
4.7 Collisions are chained based on the connected triangles. The chains are converted to control points using the depth value. In this case, there are three chains, that are combined into control points, denoted by the cyan line. . . . . 26
4.8 Illustration of control points with linear connections modelling the $I/t$ curve. Due to the nature of the method, when the $I/t$ curve changes often, there will also be more control points to model it. . . . . 27

5.1 The pipeline implemented in C++. . . . . 29
5.2 A vertex, edge, and triangle of the triangle grid, with the geometric attributes stored with each part. . . . . 30
5.3 A region of occlusion, where an object that is closer to the camera obscures another. The inverted triangles are removed and the occluded regions of the bottom triangles are hidden by the upper triangles. . . . . 31
5.4 Illustration of the chaining algorithm. . . . . 34
5.5 The pipeline implemented in Unreal Engine. Currently, it is slightly simplified compared to the full method. . . . . 35

6.1 Interpolation of a test scene. Interpolation generated by the C++ version. . . . . 37
6.2 $I/t$ curve of a pixel from the test scene. The movement of the checkerboard pattern can be seen in the intensity of the pixel. . . . . 38

6.3 Color interpolation of a frame in the Alley 2 scene from the Sintel Dataset. Interpolation generated by the C++ version. . . . . 39
6.4 Color interpolation and event generation of a frame in the Cave 2 scene from the Sintel Dataset. Interpolation and events generated by the C++ version. . . . . 39
6.5 Color interpolation and event generation of a helicopter with moving rotor, compared to rendered high frame-rate ground truth. The middle two rows show a cropped view of the rotor blade, where artifacts appear due to the linear optical flow model. Interpolation and events generated by the C++ version. . . . . 40
6.6 Interpolated frames used for the panoramic analysis. In this example, the sampled frames are $9$ degrees apart. Interpolation generated by the C++ version. . . . . 42
6.7 Example of RMSE over the rotation in the panoramic sequence for the sequence with frames spaced $9$ degrees apart. Each frame, the interpolation methods are equal to the ground truth, in-between both diverge. When there is more detail in the current frame, the RMSE of the interpolation is higher. The RMSE is shown here in the range $0..255$. . . . . 42
6.8 The difference between the true and linear optical flow in the panoramic case. When projected to the screen, the linear flow has a constant speed in screen coordinates. The projection of the true optical flow will move slower at the edges and faster in the middle, due to motion perpendicular to the screen. . . . . 43
6.9 The comparison of mean RMSE between ESIM and FastESIM for the Panoramic and Realistic Room sequence. The RMSE is normalised to the range $0..1$. . . . . 43
6.10 A view of the Realistic Room scene, rendered in Blender. . . . . 44
6.11 Interpolated frames used for the Realistic Room analysis. In this example, a key-frame spacing of $125$ frames is shown. Interpolation generated by the C++ version. . . . . 45
6.12 Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of $500$ frames. The RMSE is shown here in the range $0..255$. . . . . 45
6.12 Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of $500$ frames. The RMSE is shown here in the range $0..255$. . . . . 46
6.13 The event frames of the Realistic Room scene, generated by ESIM and FastESIM with a contrast threshold of $0.05$ and accumulation time of 10 ms. In the two left columns, the event frames can be seen, while the error of both methods is shown in the right columns. . . . . 47
6.14 Event Frame metrics for the Realistic Room scene for two contrast thresholds: $C = 0.1$ and $C = 0.05$. The top row shows the mean RMSE of the whole event frames compared to the ground truth, while the bottom row uses a sparse-aware metric to quantify the quality of the prediction. The F1 metric is the geometric mean of the precision and recall of the prediction. The precision quantifies how many of the ground truth events are predicted and the recall quantifies how many of the predicted events were correct. A higher F1 value is better. . . 48
6.15 Event Frames generated using the FastESIM Unreal Engine plug-in. . . . . 50

A.1 Alley 2 scene with generated event frames, from the Sintel Dataset. Interpolation and Events generated by C++ version. . . . . 62
A.2 A scene with a camera. Interpolation generated by C++ version. . . . . 63
A.3 Complex house scene rendered in Blender. Interpolation and Events generated by C++ version. . . . . 63
A.4 Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of $500$ frames. The RMSE is shown here in the range $0..255$. . . . . 64
A.5 Test Scene rendered in Unreal Engine. Events generated by Unreal plug-in. . . . . 65
A.6 Photorealistic room scene rendered in Unreal Engine. Events generated by Unreal plug-in. 66

# List of Tables

2.1 Comparison of different methods to represent $N$ amount of events of size $E$. $H$ and $W$ represent the frame height and width, respectively. $F$, $B$, or $K$ denotes the amount of frames, bins, or timestamps used. Table adapted from [18]. . . . . 11
2.2 Comparison of existing Event-Based Camera Simulators. Most simulators use High-Frame-Rate (HFR) frames to sample events. Adapted from [1]. . . . . 13

6.1 Render FPS and conversion factor comparison between the different simulators. The conversion factor is defined as the time unit of events that is generated per time unit required to render those events. . . . . 49

A.1 Panoramic comparison between ESIM and FastESIM for varying optical flow. The RMSE is normalised to the range $0..1$. . . . . 60
A.2 Realistic Room scene comparison between ESIM and FastESIM for varying optical flow. The RMSE is normalised to the range $0..1$. . . . . 60
A.3 Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.1$ on the range $0..1$. All metrics show the same order. . . . . 61
A.4 Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.05$ on the range $0..1$. All metrics show the same order. . . . . 61
A.5 Sparse-Aware Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.1$ on the range $0..1$. All metrics show the same order. . . . . 61
A.6 Sparse-Aware Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.05$ on the range $0..1$. All metrics show the same order. . . . . 62

B.1 The source files used in the C++ version of the FastEventGenerator. . . . . 67
B.2 The source files used in the Unreal Engine version of the FastEventGenerator. . . . . 68

# 1 Introduction

In robotics, researchers look more and more at nature to inspire their next innovation. One of these areas is small, agile, flying robots, inspired by flying animals such as birds and insects. These animals need robust and energy efficient ways to see and interpret their surroundings. The main mechanism in nature for this is vision. Many new techniques in the field of visual processing are still being developed, to allow our robots to interpret the world around them. One of these topics uses a camera sensor inspired by the natural eye to view brightness changes instead of absolute brightness. These intensity changes are called 'Events' and these sensors are called 'Event-Based' sensors.

Traditional cameras operate in distinct frames, capturing an entire scene at the same point in time. Frames from these cameras can, alongside many other purposes, be used in robotics to gain valuable information about the operating space of a robot. Other than these frame-based cameras; event-based cameras instead capture the change in brightness, not the absolute brightness. Each pixel individually fires an event immediately when its brightness changes. The difference is shown in Fig. 1.1. Compared to frame-based cameras, they have a higher dynamic range, a better time-resolution and lower power consumption [1]. This allows these cameras to see obstacles faster and in more lighting conditions, while consuming less power. Because of these benefits, event-based cameras are increasingly used instead of, or next to, frame-based cameras in robotics, especially in small, agile flying robots.

The behaviour of these kind of robots is often trained using machine learning. In order to teach and test these behaviours, virtual simulated worlds are often used, to reduce the time, effort, and other resources required to achieve a satisfactory learned behaviour policy. Frame-based cameras can be simulated using rendering engines, which can convert a virtual world into frames that such a camera would produce. These simulations can be used to design robots and their control strategies in many environments without costly testing in the real world. There is a very large body of work developing these traditional rendering engines in both academia and industry, and they are widely used [2].

There is a large paradigm shift between frame-based workflows and their event-based counterparts. This divide does not only exist for the algorithms trained on events, it also extends to the simulators that can generate data for these algorithms to be developed and trained on. Various methods to convert a virtual world into events that would be generated by event-based cameras have been developed. These existing event simulators generate events with the required time resolution by rendering many frames from

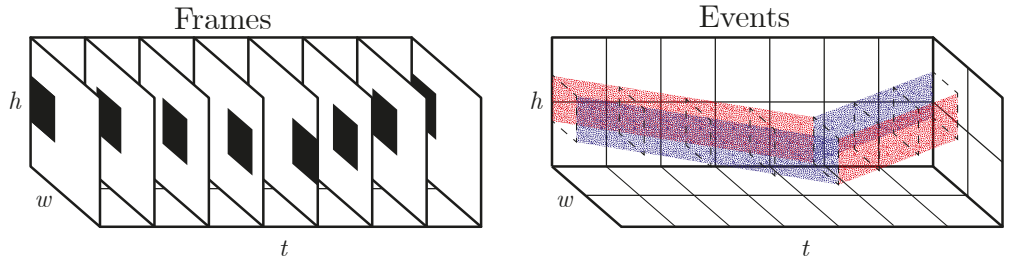



**Figure 1.1:** *Comparison between frame-based and event-based cameras. Frames are snapshots of a scene, while events provide sparse data throughout time.*

traditional rendering engines, such that the movement between each frame is very small. This results in thousands of frames for each second of events [3], much more than the typical frame-rate of a frame-based simulator. Because they need to render so many frames, they are very slow and therefore impractical to use for real-time applications, such as machine learning.

In this thesis, a new event-camera simulator has been developed, that can simulate the output of an event-camera moving through a virtual world in near real-time. By leveraging more of the auxiliary information that is generated from rendering engines, a novel frame interpolation technique is used to reduce the amount of frames needed with minimal overhead. A novel continuous interpolation method has been developed, so that events can be placed very accurately in time. Using this novel interpolation method, this new event-camera simulator will allow other research to use techniques such as machine learning to be applied much more efficiently to platforms with event-based cameras, by allowing fast training and simulation. Developed interpolation method will also be suitable for use as a video frame interpolation method.

This thesis is divided in three parts. Part I contains a literature review that has been conducted on event-based cameras and event-based simulation techniques. After the literature review, the research objective and questions are further specified. Part II consists of the methodology, where the working principle of the new simulator architecture will be discussed, and the results, where the implementation of the developed simulator will be shown and evaluated against the state-of-the-art tools. Finally, Part III contains the conclusions and recommendations for further research and development.

# Part I

## Literature Review
## and
## Research Definition

# 2

# Literature Review

In this literature review, an overview will be given of various relevant topics for the simulation of event-based cameras. The discussion of these topics will give the necessary background and will allow for the development of a novel method of rendering and simulating event-based cameras. In the first section, Section 2.1, the principle of optical flow and the brightness constancy assumption will be shown. In Section 2.2, multiple general video frame interpolation techniques will be examined, as these will be used in the event simulation process. In Section 2.3, the basic techniques behind rendering will be elaborated. Additionally, the three main implementations of rendering engines will be highlighted. Section 2.4 will continue with the principle of General Purpose GPUs, using graphics hardware to do general purpose computing. Then, in Section 2.5, the basics of event-based cameras and their working principle, benefits, and drawbacks will be discussed. Next to this, various uses of these cameras and algorithms developed for them will be shown. Finally, in Section 2.6, the existing research into event-based camera simulators will be shown and the working principle of each simulator will be discussed.

## 2.1. Optical Flow and Brightness Constancy

The optical flow of an image describes the movement of each pixel in the image [4]. It shows the direction and magnitude of this motion, usually defined as a vector field, with a vector at each pixel. In a sequence of images, these vectors point from the pixel in one frame to the location where this pixel will move to in the next frame. The optical flow field is traditionally dependent on two constraints: The flow smoothness constraint and the brightness constancy constraint.

The first constraint says that in order for the optical flow to make sense, the pixels in the image must move in a (semi-)smooth manner; it is not possible to define optical flow vectors for random noise, where no structured motion can be found. Of course, this smoothness constraint is not strict, as there will be local discontinuities at occlusion and accretion regions. The second constraint limits the brightness change of individual pixels between frames. Using this assumption, the brightness of a pixel does not change, only its location in the frame. For the construction of methods using optical flow, this is a very useful framework, as it makes calculations much easier. Yet, in the real world, there is no brightness constancy: moving objects will also change colour due to changing lighting condition, specular materials, and many more sources. This is why a linear colour changing factor, where the colour is interpolated from a start to an end value, is often included [5], at the cost of more complexity.

Optical flow is usually defined in the screen space reference system. This is a two dimensional space, with the two dimensions along the pixel grid axis. A third dimension from the screen into the scene can be added, to include changes in depth. In a rendering engine, optical flow is used for many visual effects, such as motion blur and temporal anti-aliasing. In this case, the true optical flow can be determined from the known camera and object motion in the scene. For image sequences where this information is not available, other methods are required. While originally optical flow was calculated using algorithms based on the partial derivatives of the motion in the scene and iterative methods, modern algorithms use mostly learning-based methods to determine optical flow from image sequences.

## 2.2. Video Frame Interpolation Techniques

Video Frame Interpolation (VFI) methods increase the temporal resolution of frame sequences by interpolating new frames between given frames [6]. The most naive approach to this problem, would be

to interpolate between the intensity values of each frame, based on the time distance to either frame. However, this introduces significant artifacts, as objects would disappear in one location, to appear out of thin air in some other spot. Instead, the method should consider the motion in the scene, to accurately predict where objects have to be between the two frames. There are three main methodologies that VFI algorithms follow: motion compensation-based, deep learning-based, and diffusion-based.

Early frame interpolation techniques, before the advent of deep neural networks, used block matching, similarity of blocks of pixels between the two frames, to estimate motion in the frame and pixel-level warping to synthesise new frames [6]. Newer methods use deep learning-based approaches for the flow estimation with pixel warping, or end-to-end learned pipelines for interpolation. End-to-end learned methods use convolutional neural networks to generate in-between frames. In learning-based frame interpolation methods, a distinction can be made between Center-Time Frame Interpolation (CTFI) and Arbitrary-Time Frame Interpolation (ATFI). While CTFI always generates a new frame in the middle of the given frames, ATFI takes a parameter $t$ between $0$ and $1$, to generate a frame at that fraction between the first and second frame. Recently, diffusion models have been used to generatively solve the frame interpolation problem.

Two basic approaches to pixel warping exist: Forward and Backward warping [6]. These techniques are named after the approach they take towards the interpolated frame: Forward warping warps pixels from the original frame to the interpolated frame, while backward warping samples the interpolated frame from the original frame. During forward warping, pixels in the original frame are warped to their new locations using the forward optical flow of the image, $\vec{F}$ and subsequently merged into the pixels of the interpolated frames. The forward optical flow is defined from the pixels in the original frame to the pixels in the interpolated frame and can be view in normal or reversed time: $\vec{F}_{0\to t}$ or $\vec{F}_{1\to t}$, respectively. Artifacts can appear when there is no flow going to a certain region, causing holes in the output, or when multiple regions flow into each other, causing overlaps. For holes, which occur when objects appear from an occluded region, there is no information available, while for overlapping regions, caused by occluding objects, too much information is present and the correct source needs to be determined. Optical flow maps are usually represented as linear vectors between frames. This limits their performance, since the motion between frames is often non-linear.

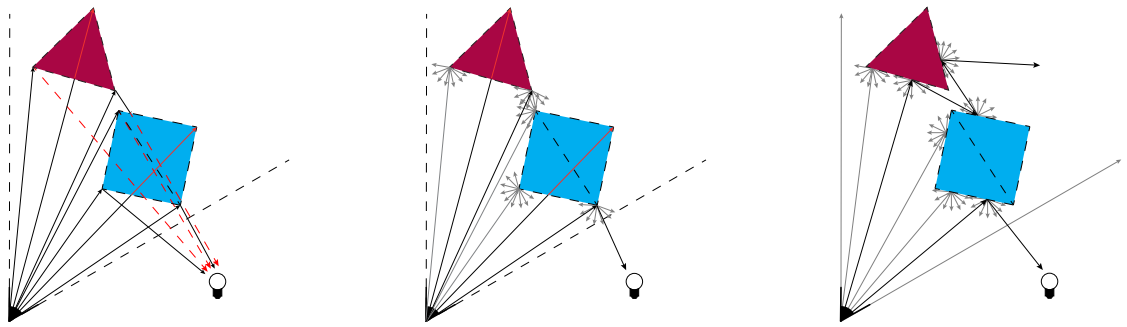
In backward warping, the value at the interpolated pixels is instead sampled from the original image based on backward optical flow, $\vec{F'}$. This backward optical flow is defined as the origin location of each interpolated pixel in the original frame and can also be defined for both time directions: $\vec{F'}_{t\to 0}$ or $\vec{F'}_{t\to 1}$. With an ideal backward flow map, this would result in a correct image, for all regions that exist in either of the frames that is interpolated between. However, obtaining this backward warping flow is not trivial. Many techniques use learned methods to generate these flow maps.

One such method that uses a combination between pixel warping and learning based methods is SoftSplat [7]. It uses an off-the-shelf optical flow method to estimate forward optical flow between two frames. Then the pixels are forward warped using the scaled optical flow to time $t$. Overlapping pixels are merged using a softmax blending, weighted with a learned importance map.

Another method, that uses the backward warping instead of forward warping, is SuperSloMo [8]. This method uses a CNN to synthesise bi-directional optical flow between the given frames. Afterwards, for each $t$ that is requested, the backward optical flow is estimated from the forward optical flow and refined using a learned model. This is performed for both directions, from the first to the second frame and in reverse. This backward optical flow is used to sample from the original images, with a learned blending factor between the colours estimated from the first and from the second frame. This allows the method to perform ATFI on any two frames, however, the performance is dependent on the training data. The SuperSloMo framework is computationally intensive, because it needs to infer the backward flow refinement and frame blending networks for each $t$ that is generated.

## 2.3. Working Principle and Capabilities of Rendering Engines

In essence, a rendering engine is a computer program that can convert a stored representation of a scene, consisting of various objects, lights, and a viewpoint, to an image (or frame) [2]. By repeatedly running this process, while slightly moving the objects each time according to a predetermined animation, multiple frames can be rendered and combined into a video.



**(a)** *Rasterisation Rendering* **(b)** *Ray-Tracing Rendering* **(c)** *Path-tracing Rendering*

**Figure 2.1:** *A 2D illustration of different rendering approaches: Rasterisation is very quick, path-tracing is very accurate, and ray-tracing is a hybrid between the two.*

The most common way of representing the objects in a scene is by using a mesh that consists of points, called vertices, that are connected by edges and combine into triangles, squares, also called 'quads', or polygons of higher order, sometimes called 'ngons'. Usually, these higher order polygons are subdivided and simplified into triangles. The scene is viewed from a certain viewpoint. This virtual camera is modelled after a real camera, with a position and direction, a view frustum, and view plane. The view frustum defines the edges of what the camera can see in the world and the view plane can be thought of as the screen placed in the virtual world. The final result, the rendered image, is a frame that consists of many pixels, each has an intensity and a colour, determined by the projection of the objects in the scene on the view plane, that need to be calculated by the engine.

In these scenes, each object has various parameters that govern how it should respond to light. The coloured light intensity, or radiance, of each object is modelled by considering the colour that the object emits itself and the amount of radiance that is reflected from other sources in the scene. For a sample point on the surface of such an object, this model is captured in the rendering equation:

$$L_o(\vec{p},\vec{v}) = L_e(\vec{p},\vec{v}) + \int_{\vec{l}\in\Omega} f\left(\vec{l},\vec{v}\right) L_o\left(r\left(\vec{p},\vec{l}\right),-\vec{l}\right)\left[\vec{n}\cdot\vec{l}\right]^+ d\vec{l} \tag{2.1}$$

where $L_o(\vec{p},\vec{v})$ is the outgoing radiance from a point $\vec{p}$ in the direction $\vec{v}$, $L_e(\vec{p},\vec{v})$ is the radiance self-emitted by point $\vec{p}$ in direction $\vec{v}$, $\Omega$ denotes the hemisphere of directions $\vec{l}$ above the point $\vec{p}$, $f(\vec{l},\vec{v})$ is a function that defines the reflective properties of light incoming from $\vec{l}$ when viewed from $\vec{v}$. The output of the equation also appears again as input, as the amount of light that is coming from a certain direction must itself again be calculated. This time, the term appears with $r(\vec{p},\vec{l})$, which is the ray-casting function that returns the location of a surface point the first object hit in the direction $\vec{l}$. Finally, $[\vec{n}\cdot\vec{l}]^+$ is the dot product between the normal vector of the surface of point $\vec{p}$ and $\vec{l}$, keeping only the positive values.

In order to generate a frame by obtaining a value for each pixel, using the rendering equation, there are two main rendering engine strategies: *rasterisation* engines and *path-tracing* engines. Recently, *ray-tracing* was added to these strategies as a hybrid approach between the two. Fig. 2.1 shows a diagram of each approach. The following sections will highlight each method.

### 2.3.1. Rasterisation Rendering Engines

The *rasterisation* rendering pipeline consists of three main steps: the Vertex Shader (VS), the rasteriser, and the Pixel Shader (PS) [2]. Because this pipeline has been refined and standardised over many years, GPUs have dedicated hardware circuits to efficiently calculate these steps. First, the properties of each vertex are evaluated in the vertex shader. These properties include the vertex position on the screen, its colour, and the normal vector to the object that it is part of. The vertex shader is a small program that runs for every vertex in the scene, in parallel on the GPU. In the vertex shader, the shade of the vertex is

determined from its intrinsic properties and the scene information, such as the position and intensity of lights, using the rendering equation (Eq. (2.1)). By summing the radiance that is reflected at the vertex from each individual light in the scene, the final shaded colour can be determined.

Next, for each triangle, it is determined which pixels it occupies. This process is called rasterisation and lends it name to the entire method. Finally, in the pixel shader, the output value of each pixel is determined from the triangle that occupies this pixel on the screen. As a triangle is built up from three vertices of which we know the properties, the output value at the pixel location can be obtained by interpolating the properties from its vertices to the pixel location. When multiple triangles overlap at the same pixel, only the one closest to the camera is used. This is evaluated using the depth of each triangle at the pixel: each time a new triangle is drawn to the screen, the current depth is compared to the new depth and only when the new depth is closer will the colour and depth of the pixel be updated.

This process results in a final image that shows the objects in the scene, illuminated by the lights in the scene. There are many more visual effects that have to be considered to render a realistic image, such as shadows and transparency. The rasterisation approach uses various tricks to display advanced lighting effects. For example, shadows are calculated by checking for blocking objects between a vertex and a light when shading that vertex. Transparency is usually rendered by separating the process into two passes, first all the opaque objects are rendered, then all transparent objects placed are on top.

### 2.3.2. Path-Tracing Rendering Engines

Compared to rasterising engine, a *path-tracing* rendering engine uses a more physics-based approach [9]. Instead of solving the rendering equation, Eq. (2.1), only at the vertex locations, it casts rays from the camera in the direction of a pixel on the screen and evaluates the equation at the point that is hit by this ray. Because the rendering equation has a recursive component, the incoming radiance at this point is determined by casting new rays from this location. This creates many paths with many sections, from the camera into the scene, that finally provide the value to place at a pixel.

Solving the integral in the rendering equation is often done through Monte Carlo methods. By randomly sampling new rays in the domain, the true value of the radiance integral can be estimated. Using more samples reduces the noise introduced into the image by the random sampling. While this method allows the engine to generate images with fewer paths, still many are needed, and rendering occurs at rates that are far removed from real-time. Often, noise reduction techniques are used to reduce the amount of samples needed.

One implementation of this path-tracing method is the rendering engine Cycles in Blender[1]. It can render physically accurate scenes, but each frame requires a significant computation effort to generate. Cycles samples the scene using random rays, that are distributed to divide them as equally as possible over the scene. Many physical camera effects are simulated in Cycles, such as motion blur, depth of field, caustics, subsurface scattering, and volumetric scattering. Cycles also generates many render passes in the rendering process, such as the primary result: the color pass, but also a depth pass, an optical flow pass, a crypto-matte pass[2], and a normal pass. Additionally, the different contributions to the color pass, such as diffuse, glossy, volumetric, or emissive lighting, can each be retrieved independently.

### 2.3.3. Modern Ray-Tracing and Unreal Engine

Modern real-time rendering uses a hybrid method, that combines the fast rasterisation method with the pretty physics-based approach from the path-tracing methods [2]. Initial scenes are still generated using the rasterisation pipeline, where vertices and triangles are converted to pixels, but afterwards, the values of these pixels are found using ray-tracing methods. An example that uses such an implementation is Unreal Engine[3], which provides a very fast and realistic rendering pipeline. The rendering pipeline in Unreal Engine is designed for real-time applications and is designed such that it can easily be extended with new custom steps. Unreal Engine uses various shaders of multiple types in its rendering pipeline: Vertex Shaders, executed for each vertex; Pixel Shaders, for each pixel; or Compute Shaders, that can be used to compute more general tasks. All shaders get compiled into a Render Dependency Graph (RDG),

[1] Open Source 3D Software: Blender. *blender.org*

[2] A crypto-matte pass shows the different objects that contribute to the value of each pixel, similar to a segmentation pass

[3] Open Source Game Engine: Unreal Engine. *unrealengine.com*

which defines the order and input-output relations between all shaders, so each frame can get rendered in the most efficient order.

## 2.4. GPGPUs: General Purpose computing on Graphical Processing Units

Graphical Processing Units (GPUs) were initially added to computers to aid the Central Processing Unit (CPU) in the massively parallel tasks that come from rendering images for running games and other graphics workloads [10]. CPUs are designed with a few very strong cores, where each core can perform complicated tasks very efficiently, but only one task at a time. A GPU on the other hand, designed for parallel computing, has many small cores, that each are slower than one big core, but combined are much faster, if the task at hand is separable in many parallel tasks. The first GPUs were dedicated to graphics tasks only, but because of their efficacy for all other massively parallel tasks, these GPUs were quickly being used for all other sorts of workloads, such as scientific computing and AI. From the need to program these cards in a flexible and general purpose way, CUDA was created. With this new GPGPU (General Purpose computing on GPUs) framework, these cards could be integrated into bigger programs, where only the most parallel tasks of the program gets delegated to the GPU, and the CPU is used for all sequential parts.

A single CPU core is designed to work on its own. It can execute a single task on a single piece of data: Single Instruction, Single Data (SISD), or execute a single task on a vector of a small amount of values: Single Instruction, Multiple Data (SIMD). On the contrary, cores in a GPU are designed to work together. GPUs with the CUDA architecture work on the Single Instruction, Multiple Thread (SIMT) paradigm: these cores are grouped in warps, where each core in the group performs the same operation, but on different data. This way, large parallel tasks, such as matrix operations, can be performed by all cores in tandem, each performing the same calculation on another part of the matrix. A weakness of this architecture appears when these cores need to diverge. Each branching path needs to be performed by all cores in the warp, where the cores that have taken another branch are masked. Another bottleneck on the GPU is memory. In a typical system, there is a separation between the host memory, that of the CPU, and the device memory, local to the GPU. Any tasks that the CPU wants to offload to the GPU, first needs to be transferred. During the task, the intermediate steps need to be stored as efficiently as possible for all the threads in each warp, otherwise the memory overhead can cause more bottlenecks. These effects can greatly reduce performance and needs to be taken into account when designing algorithms for CUDA.

## 2.5. Event-Based Cameras and their Uses

In this thesis, an event-camera simulator will be developed. This pipeline will simulate the output of a real event-camera. Compared to frame-based cameras, event-cameras provide a fundamentally different way to look at the world. Inspired by biological eyes, instead of providing snapshots timed to a clock firing at an arbitrary rate, these cameras output continuous-time information about the world. These event-cameras provide many benefits over frame-cameras, such as a much higher temporal resolution, lower latency, higher dynamic range, and lower power consumption. The output of these neuromorphic cameras, a continuous stream of events, is very well suited to be combined with neuromorphic image processing techniques.

In this section, the working principle and particularities of the event-based camera will be explored. First, the difference between frame-based and event-based cameras will be further discussed. Next, the strengths and weaknesses of Event-Based vision will be highlighted. Finally, multiple promising learning and neuromorphic algorithms for event-based cameras will be shown. The development of these algorithms shows the usefulness of and need for a fast event-based camera simulator.

### 2.5.1. Working Principle of Event-Based Cameras

In a frame-based camera, a clock decides when it is time to record a new frame. This signal is sent to each pixel on the sensor, which start accumulating light intensity information as a voltage. When the preset exposure time has elapsed, each pixel reports the total accumulated intensity, which defines the brightness at that pixel in the captured frame. Afterwards, the sensor can start recording the next frame. In a global shutter sensor, all pixels start and stop exposing at the same time, while in a rolling shutter sensor, they are sampled in a fixed sequence, usually top to bottom. All the sampled pixels are used to construct the same frame.

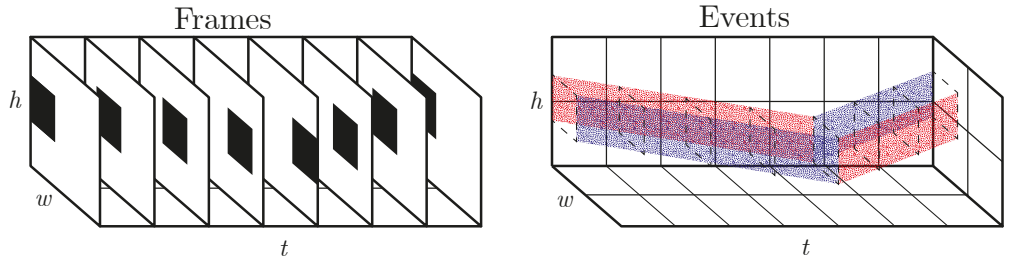



**Figure 2.2:** *Comparison between frame-based and event-based cameras. Frames are snapshots of a scene, while events provide sparse data throughout time. In this example, the square moves from side to side. The events only fire at the moving leading and trailing edge of the square. The dashed square is drawn with the events as a indication of its location at the times at which a frame is captured. Positive events are shown in red and negative events are shown in blue.*

In contrast to the synchronous nature of the frame-camera, the individual pixels of the event-camera work asynchronously. Each pixel stores a certain reference intensity. The pixel continuously converts the incoming photons to a voltage and monitors the deviation of this voltage from the reference level. When it detects a deviation larger than some preset threshold $C$, the pixel fires an event $\langle [x, y], t, p \rangle$ with its location in the frame $[x, y]$, the current time $t$, and the polarity $p \in \{-1, 1\}$ of the change [1]. The threshold on the intensity deviation is generally symmetrical, but can also be different for an positive $C_p$ or negative $C_n$ change in intensity. The reference intensity is updated to the new intensity and the process begins again. The duration that the pixel takes from triggering an event to being ready for the next is called the refractory period. This time limits the rate of events per pixel. This design results in a pixel that will respond almost instantly to a change in brightness, but stays silent when the intensity does not change [11].

For a frame-based sensor, if too many photons are captured during the exposure, the sensor clips and the information about the real total brightness is lost. To recover these highlights, the exposure time needs to be reduced. However, this can lead to problems at the other end of the brightness spectrum. In a region of very low light, when the exposure time is reduced, a larger sensitivity is needed to accumulate enough voltage to rise above the noise floor of the pixel. The exposure time and sensitivity of the sensor (ISO) can be changed to adapt the camera to the current environment. The combination of these two parameters determine the *Dynamic Range* of the camera, which is defined as the range from the darkest to brightest intensity that the sensor can record at the same time. For a frame-based sensor, these settings are the same for all pixels, which means that the sensor has to choose an operating window and cannot recover information from a very bright part and a very dark part of an image at the same time. Additionally, when the intensity of the light changes drastically, the settings of the sensor need to change. In contrast, each pixel in the event-based sensor only records the change in log intensity, regardless of whether this intensity is very bright or very dark. The event-based sensor can read the same amount of information from bright and dark parts of an image at the same time. It also does not need to change settings when changing from a dark to bright environment. In event cameras, there is a limitation on the rate of change of the intensity, which is determined by the refractory period of the pixel after it has fired an event. Typical frame-based cameras can obtain a dynamic range of 60 to 85 dB, while the very best cinema cameras can achieve around 102 dB[4]. In contrast, an event-based camera can achieve a dynamic range of over 120 dB[5], while being much smaller, consuming significantly less power, and being a fraction of the cost[6].

The temporal resolution of the frame-based sensor is limited by the exposure time and the time required to process the recorded frame, because the only time information from such a camera is stored is at the time a frame is captured. When the values from all pixels on the sensor need to be written away to a file, this time is not insignificant. This causes information from most frame-based sensors to have a low temporal resolution. Using high-end specialised hardware[7], the time required to process a frame can be greatly reduced, resulting in high-speed cameras with a very high temporal resolution. These cameras can

[4] ARRI ALEXA 35 Xtreme has 17 stops (102 dB) of dynamic range. *arri.com/en/camera-systems/cameras/alexa-35-xtreme*

[5] iniVation DAVIS346 has 120 dB of dynamic range. *docs.inivation.com/hardware/current-products/davis346.html*

[6] An Arri Alexa 35 Xtreme costs upwards of €50.000, a DAVIS346 camera is around €4.000

[7] Phantom TMX 7510 High-Speed Camera. *phantomhighspeed.com/products/cameras/tmx/7510*

achieve frame rates of up to $1.75 \times 10^6$ fps on small frame windows, or 375.000 fps with resolutions similar to current event-based cameras, a time resolution of about 2.5 µs per frame. However, they have other downsides. Because the exposure time is also very short, a lot of light is needed to overcome the noise in the sensor. Additionally, because the frame rate is so high, the data generated by these cameras is also very large, yet, most of the captured frames will have large identical portions containing redundant data. Event-based cameras solve these problems, by only storing information as it changes, directly when it changes. This results in a very high temporal resolution for the information in the form of events. Events are captured with a timestamp that has microsecond resolution and can be transmitted with sub-millisecond latency to be processed [11]. When capturing fast moving subjects, or with fast ego-motion of the camera, the event-based sensor records only the actual information from the motion with very minimal motion blur, very high accuracy, and very little redundant information.

The electrical architecture of the event-based sensor allows it to operate very power-efficiently [11]. The sensor only uses power to transmit changes of pixels and does not have to repeatedly sent the same information. Additionally, in applications where the information about the motion of the scene is desired, a setup with a traditional frame-based sensor would require power hungry processing to convert the generated frames into the information that is required. With an event-based sensor, this information is already the output of the sensor, and this expensive step can be removed. This allows the vision system to be designed with very minimal power requirements, which is very beneficial for applications with a strict energy budget, such as computer vision on robotics and drones.

The features of event-based cameras, their high dynamic range, high temporal resolution, and low power usage, rival that of individual, high-end, specialised cameras; that each are designed to achieve one of these three features. A frame-based camera that can achieve a similar dynamic range cannot achieve the same temporal resolution. A frame-based camera that can achieve the same temporal resolution does not have the same dynamic range and a frame-based camera that has the same low-power requirements does not have the same qualities as the previous two cameras. The event-based sensor provides a way to achieve all of these at a relatively low cost and very low weight. Obviously, these specialised cameras will not be replaced by event-based cameras soon, as they are not designed for the same applications and have other features that an event-based sensor does not have, such as high pixel resolution or very colour accurate recording. Still, for computer vision, event-based vision provides very promising characteristics.

### 2.5.2. Challenges of Event-Based Cameras

Since event-based sensors capture the world in such a different way to frame-based sensors, there are various challenges to overcome to use make effective use of these new sensors [11]. Frame-based vision pipelines are largely not applicable to event-based data.

Traditional frame-based sensors output dense frames at relatively fixed time intervals. Each new frame provides a vision processing algorithm with a new work package to process. Because the new information arrives is discrete packages, these algorithms are often pipelined, where the total process is subdivided into smaller steps that can run simultaneously on consecutive frame packages. These existing frame-based vision algorithms cannot easily be used on event data, since this data is asynchronous and sparse. Therefore, new algorithms and pipelines need to be developed to process event data efficiently.

At the root of the paradigm shift from frame-based sensors to event-based sensors lays the change from storing the absolute intensity of a pixel, which requires a large bit-depth to store with sufficient resolution, to the binary direction of change of the intensity, which only requires one bit to store[8]. This principle allows the event camera to efficiently report events over the entire intensity spectrum, without needing to use a lot of bits to store the absolute intensity, which is why it has such a large dynamic range. This does however mean that the absolute intensity is not available and the vision processing algorithms need to be changed to use relative brightness instead. This relative brightness does not only depend on the changes in the scene, but also on the ego motion of the camera, which further complicates the vision processing.

All camera sensors have to deal with noise, non-linearities and other non-ideal effects. Just like for frame-based sensors, these effects need to taken into account or mitigated when dealing with data from event-based sensors. However, unlike frame-based sensors, which have existed for a long time, where all

[8] The time at which an event has occurred is still stored, increasing the data usage of events. This is irrelevant in this case, because the resolution and dynamic range are decoupled when using relative intensity.

**Table 2.1:** *Comparison of different methods to represent $N$ amount of events of size $E$. $H$ and $W$ represent the frame height and width, respectively. $F$, $B$, or $K$ denotes the amount of frames, bins, or timestamps used. Table adapted from [18].*

| Representation | Dimensions | Description | Discards |
|---|---|---|---|
| Raw Events | $N \times E$ | Store all $N$ events | - |
| Event Frame [12] | $F \times H \times W$ | Images of accumulative summed events | Temporal and polarity information |
| Event Frame [13] | $F \times 2 \times H \times W$ | Images of accumulative events | Temporal information |
| Surface of Active Events [17] | $F \times E \times H \times W$ | Image of most recent time stamp | Earlier time stamps |
| Voxel Grid [14] | $B \times H \times W$ | Voxel grid summing event polarities | Event polarity |
| Average Time Surfaces [15] | $F \times 2 \times H \times W$ | Image of average timestamp in window | Temporal information |
| Inceptive Time Surfaces [16] | $F \times 3 \times H \times W$ | Image of filtered timestamp & event count | Temporal information |
| Event Spike Tensor [19] | $B \times 2 \times H \times W$ | 4D grid of convolutions | Temporal information into $B$ bins |
| TORE Volumes [18] | $K \times 2 \times H \times W$ | 4D grid of last $K$ timestamps | All information before last $K$ events |

these effects have been extensively studied and characterised, and many methods to counteract them have been developed; for event-based sensors this analysis has not been performed as extensively and these mitigation methods do not exist yet.

### 2.5.3. Different Representations of Event Data

Like frame data, many ways of representing and storing event data exist. Various methods can be seen in Table 2.1. There are many research investigations into the most efficient method of storing and compressing events. Raw events from an event-based camera consist of 4 data fields: $\langle [x, y], t, p \rangle$. Of these, $t$ is usually chosen to be most accurate, while $[x, y]$ can be encoded more efficiently and $p$ only requires a single bit. Combined, such raw events can take up significant amounts of data, similar to high-dynamic-range frames. For many applications, the very high temporal resolution is not required, and the events can be stored in a more efficient manner by removing some of the unnecessary data. Additionally, many vision processing methods are available for frame-like data, but not for sparse event data.

This is why it can be beneficial to generate pseudo-frame representations from event data. For a fixed time-window, all events can be summed, to achieve a single net value for each pixel [12]. This resulting Event Frame captures the magnitude of change over the time-window, but discards all temporal information inside the time-window, as well as any positive events that are cancelled by negative events, and vice-versa. Instead of summing all events, the positive and negative events can be summed into separate counts, resulting in an Event Frame with two channels for each pixel, instead of one [13]. In this case, the temporal information inside the time-window is still discarded, but the true magnitude of events fired for each pixel is retained. Other information about events inside the time-window can be stored instead, such as an transformer encoded value based on near events, used in voxel grids [14], or a value encoded by the average time of near events, as in [15] or with filtered events in [16]. Instead of accumulating events in a fixed time-window, other methods limit the computational burden by only looking at the last event for each pixel [17], which is useful is very sparse situations, or by looking at a fixed amount of past events for each pixel [18].

### 2.5.4. Event-Based Computer Vision and Neuromorphic Algorithms

Many different uses for event-based computer vision have been found, spread over many different fields. One example of such a use-case is in the automotive field, where there is much interest in event-based cameras for self-driving capabilities [20]. Here, the high temporal resolution and high dynamic range of event-based vision allows it to react quickly to many circumstances that might arise on the road. Next to out-of-cabin vision, event-based vision can also be used for in-cabin monitoring, to detect driver state and attention. Another field where event-based vision is an quickly emerging technology is the robotics community [21]. Their low power usage and low weight, combined with a high temporal resolution, allows these cameras to be used in small robots. The output of these neuromorphic cameras is very suitable to be paired with further neuromorphic processing, such as Spiking Neural Networks (SNN) [22].

### 2.5.5. Event Datasets

In order to develop and evaluate event-based vision processing methods, datasets of event data are very important [1]. In order for different implementations to be compared to each other, it is crucial that they are

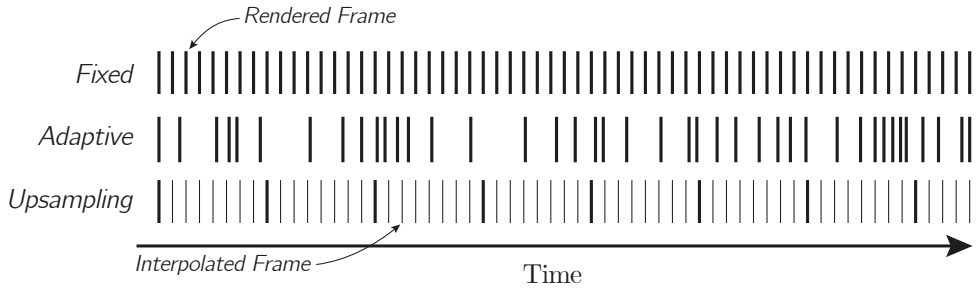



**Figure 2.3:** *Sampling strategies of different event-based camera simulators. Fixed sampling uses a predetermined high frame rate, often above the Nyquist Rate of the fastest movement in the scene. Adaptive samplings adjusts the sampling rate to the current maximum optical flow magnitude in the frame. The upsampling strategy uses a low frame rate input that is temporally upsampled using VFI methods.*

evaluated on similar input data. While some datasets exist, the cost of creating very accurate datasets remains high. Often, these real-world datasets do not supply ground truth data or labels, which can limit their applicability to certain tasks. Another downside of using existing datasets for training is that the camera path is fixed, so there is no opportunity for controller-in-the-loop training.

## 2.6. Working Principle of Event-Based Camera Simulators

To lower the barrier of entry and to efficiently use machine learning techniques on event-based cameras, it is very useful to have a simulator that can render the events from a virtual world as an event-based camera would from the real world. As the event-based sensor is a revolutionary new way of capturing the outside world, traditional render engines are not able to produce a mirror of the data that is recorded. These render engines are designed in the same frame-based paradigm as the frame-based camera, and therefore also see the virtual world in slices of time, unable to produce continuous data of that which occurs in-between frames. In 2017, the first event-based camera simulator, the DAVIS Simulator, was released. Since then, many new iterations and different simulators have been published. In Table 2.2, these simulators are listed. This section compares the different techniques used by these simulators to approximate or recover this missing data and provides insight on the development gap that still exists. Three main strategies exist: fixed rate rendering, adaptive rate rendering, and frame-rate upsampling. These strategies are illustrated in Fig. 2.3.

### 2.6.1. Current Event-Based Camera Simulators

The first simulators, DAVIS Simulator [23] and ESIM [3], used a 3D scene in Blender to render intensity frames at a very high frame-rate. In order to sample a signal without loss of information, it has to be sampled with at least twice the bandwidth of the signal, the Nyquist rate. For the DAVIS Simulator, this frame-rate was fixed. The difference between these intensity frames was used to generate events. Extending this idea, ESIM used a variable frame-rate, based on the optical flow of the image. The optical flow distance was set by selecting the distance to travel along a known camera trajectory for each frame. While this resulted in a more efficient rendering process, making sure that this motion is small enough still caused the necessary frame-rate to become very high[9], which in turn still caused a large computational burden when rendering events. Rendering one minute of event data may take multiple hours with this process. While computationally intensive, these methods do guarantee a correct idealised event simulation.

Instead of using a rendering engine to render many frames, the Vid2E simulator takes already existing low frame-rate frames and uses existing frame-rate upsampling techniques[10] to generate high frame-rate image sequences as input to generate events [24]. Vid2E uses Super SloMo, a learning based frame interpolator [8]. Using frame interpolation means that a lower frame-rate can be used for rendering and it allows the method to convert existing lower frame-rate datasets to events. The event generation was

[9]High Frame-Rate (HFR) is defined in this thesis as rates above 100 FPS, but for event generation can typically be 1000+ FPS, while Low Frame-Rate (LFR) is defined as around 1 to 100 FPS.

[10]Video Frame interpolation (VFI) techniques have been discussed in Section 2.2.

**Table 2.2:** *Comparison of existing Event-Based Camera Simulators. Most simulators use High-Frame-Rate (HFR) frames to sample events. Adapted from [1].*

| Simulator | Publication Date | Operating Principle |
|---|---|---|
| DAVIS Simulator [23] | 2017 | Sampling from HFR frames from 3D scene in Blender |
| ESIM [3] | 2018 | Sampling from HFR frames from 3D scene in Blender/Unreal |
| Vid2E [24] | 2019 | Upsampling from LFR image sequences |
| V2E [25] | 2020 | ESIM with added noise patterns |
| ICNS Simulator [26] | 2021 | Improved noise values and validation for V2E |
| VISTA [27] | 2021 | LFR frame interpolation within VISTA |
| DVS-Voltmeter [28] | 2022 | Simulate DVS-voltage between HFR frames |
| V2CE Toolbox [29] | 2023 | Directly generate Voxel Grids from HFR frames |
| Carla: DVS Camera [30] | 2023 | LFR frame interpolation within CARLA |
| Prophesee: VtoE [31] | 2023 | Sampling from given HFR frames |
| EvDNeRF [32] | 2023 | Generate events from NeRF |
| Text-to-Events [33] | 2024 | Generate events from text-prompt using Diffusion |

performed by linearly interpolating between the brightness of each upsampled frame and triggering events based on a contrast threshold $C_p$ or $C_n$. This contrast threshold is sampled from a uniform distribution between $C_{\text{min}} = 0.05$ and $C_{\text{max}} = 0.5$ for each pixel separately. Next to the raw event output, another event representation was used for a learning demonstration, where the events are distributed into spatio-temporal grids of a certain temporal width.

In machine learning, when transferring a strategy that was taught in a simulator to the real world, all differences between these environments can cause performance degradation, because the behaviour that is learned using the rules of the simulator might not be effective in the real world. This phenomenon is called the sim-to-real gap. In order to reduce this sim-to-real gap in event-based camera simulations, various works have added effects to convert the ideal event simulation from previous simulators into a more realistic event simulation, that is also valid under difficult, low light, conditions. The V2E simulator adds realistic noise patterns, which are also found in real event cameras [25]. This is achieved with multiple steps. First, the intensity is converted from linear to log, with an exception for low intensities to reduce quantization noise. Next, an intensity-dependent low-pass filter is applied, to simulate the limited photoreceptor bandwidth under low light conditions. Finally, the events are sampled from the difference in logarithmic intensity. Various noise effects are added to the pipeline, such as *threshold variation*, sampled from a Gaussian distribution; *hot pixels*, which have an abnormally low threshold and therefore continuously fire events; and *leak events*, which occur due to the leaking of the reference value and can cause spontaneous positive events.

The ICNS simulator tweaks these noise models and validates them against data from a real event-based sensor [26]. DVS-Voltmeter uses another technique for modeling the noise, by introducing noise into the otherwise linear interpolation between frames, to randomise the distribution of the events [28]. It is shown that events generated using this model resemble realistic events better, compared to previous interpolation techniques, when using a lower frame-rate, but are less accurate. The V2CE simulator aims to generate events first in voxel representation, instead of raw events, using learning methods, and later convert these voxels to event streams, in order to improve the temporal accuracy of converted datasets [29].

A recent development in the space of event-based camera simulators is the use of novel learning based models for event generation. Using Neural Radiance Fields (NeRFs), EvDNerF [32] generates events from learned volumetric representations of scenes. Using this simulator, novel views into the scene can be queried from the radiance field. The simulator is trained on events generated using Vid2E: LFR rendered frames from many angles are upsampled and converted to events. This method allows for flexible generation of realistic event stream of paths through a trained scene, but needs to be retrained for every new scene that needs to be converted. ‘Text-to-Events’ [33] trained a latent diffusion model to generate event streams from text prompts. This would allows for easy generation of simple datasets, but does not grant the ability to explicitly set the motion and content of a scene. So, while this method allows for quicker generation, it does not allow for specific motion commands to be integrated with the simulator. While the event streams generated by this method capture the large motion structures, they are still very unrealistic.

Next to these core improvements, multiple works have integrated the event generation principles into

existing simulators. VISTA is a simulator designed for autonomous car development [27]. This simulator uses existing image datasets to generate novel viewpoints for learning. It generates events using frame interpolation with an adaptive sampling rate, like ESIM, and a Gaussian threshold. CARLA is another autonomous car development simulator [30]. It is build on Unreal Engine to render 3D scenes of generated road layouts. It can produce event streams by uniform sampling between two high frame-rate rendered frames from Unreal, similar to the DAVIS Simulator. Prophesee produces event cameras and published an extensive toolbox for event data processing [31]. In this toolbox, they provide a conversion tool for converting high frame-rate video into events.

### 2.6.2. Gap for better simulators

At their core, frame-based rendering engines are not well suited to generate events. They are hyper-optimised to render frames at fixed points in time and do not care for the time in-between these frames, while event streams show exactly that time. The way that previous event simulators have overcome this problem is to either make the time in-between frames very short by rendering at a very high frame-rate, or by generating in-between frames by interpolating from frames that are further apart. The big downside to these approaches is the computational burden imposed, either from rendering many frames, or interpolating them. Because the entire frame needs to be rendered at the rate of the fastest moving pixel, many regions are rendered at a very high rate, even though the events in these regions would also be captured at a lower rate.

There are two main directions to improve event simulation. The first is a better integration into existing rendering engines, as there are many features of rendering engines that can be used to speed up the event generation process. The optical flow and depth passes[11] are under-used in current methods. All simulators based on interpolation use existing interpolation methods that interpolate a whole frame at a certain requested timestamp. For event generation, a model of continuous intensity of each pixel is much more relevant than whole frames at predetermined timestamps. A new simulator could be much faster with an integrated frame interpolation solution that can quickly model this intensity continuously.

The second direction to improve is to start from the ground up with a first-principles event renderer. Because frame renderers are inherently not suitable for event rendering, starting from scratch and writing an engine that does not need frames as intermediate representation would provide the best performance. Simultaneously, this also provides the biggest challenge, as many decades of graphics research becomes irrelevant and all optimisations that have been discovered for rendering frames becomes unusable. Such a first-principles event renderer would be able to convert a scene representation, stored in a similar fashion as frame renderers, with a certain motion, to events. Because this renderer does not evaluate the position of each object at a given moment, but considers the continuous motion of the scene, only the parts of the scene that actually generate events are considered and there are no unnecessary frames or regions of frames rendered.

[11] Rendering engines render the final image in separate stages, called ‘passes’. Internally the optical flow and depth are, among other passes, generated and used during the rendering process. See Section 2.3.

# 3

# Problem Statement and Research Definition

In this chapter, the research objective and research questions of this thesis will be defined. Additionally, the scope and planning will provided.

## 3.1. Research Objective and Research Questions

This thesis has a development goal, shown in the Research Objective. The tool that will be developed as a result of this thesis should provide other researchers with a method to generate simulated events of high quality in a fast manner. Other than previous methods, that were too slow due to the high computational burden imposed by them, this new simulator should be able to be used to train a machine learning algorithm in real-time, with the simulator and the controller in the loop.

**Research Objective**

In this thesis, a faster, real-time, method of simulating Event Cameras will be developed.

The method proposed in this thesis generates events by modelling pixel attributes between given frames in a continuous way. Previous event simulators sample the scene many times to generate this data. By sampling at twice the rate of the fastest motion in the scene, it is guaranteed that this pixel data is accurate enough to find all events that would be generated. The intensity is linearly interpolated between two samples. The amount of frames that need to be generated to achieve this rate is very high. Many new simulators have been proposed since the first was released in 2017, but these simulators all focus on improving the realism of the generated events by including noise modelling and improved hardware simulation techniques. These simulators still use the technique of high frequency sampling to generate their events. This method is inherently slow, as generating so many frames produces a high computational load. Some simulators use frame interpolation, but still use this to generate in-between frames at a very high frequency. Recent developments in real-time rendering have improved the performance of these methods, but they are still severely limited by the rate at which new frames can be generated.

Instead of interpolating using standard video frame interpolation techniques, the information between frames that is required to reconstruct events should be modelled in a more efficient manner than frames at fixed rates. Current video frame interpolation methods work in a discrete approach: given a time fraction between the first and second frame, the method generates a frame there. However, for this case, the continuous variant is needed: Based on the given data, what happens in between the frames continuously? To represent this information, a model needs to be found that can describe it more efficiently than discrete high-frequency frames.

**Research Question 1**

How can modelled pixel attributes be represented between frames?

The first research question will provide a direction to search for a representation that can describe the desired attributes between frames with enough accuracy to generate events. While this representation is needed to synthesise events, it can also be used to interpolate frames at any point between the two frames. Both usages of this representation will be evaluated in this thesis.

Existing simulators use rendering engines to generate frames at the necessary time intervals. Next to intensity frames, these rendering engines use other data from the scene internally for the rendering process, such as the depth and optical flow of the scene. This information can be used to model between frames, as the movement of pixel attributes through the frame is defined by it. Research Question 2 will be used to develop a new method of modelling pixel attributes, for example intensity from the intensity frames, using the depth and optical flow information and the representation from Research Question 1.

**Research Question 2**

How can continuous-time pixel attributes be modelled from low frequency frames and data generated by a rendering engine?

After this new method has been developed, to determine if it can be used for event generation and frame interpolation, it needs to be efficiently implemented. Using various computational techniques, the representation of the inter-frame data needs to be calculated as fast as possible. The developed method will be implemented on the GPU for the best performance. Research Question 3 will be used to develop and evaluate this fast implementation.

**Research Question 3**

How can continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be computed in (near) real-time?

After the method has been implemented, its accuracy needs to be evaluated. Research Questions 4 and 5 will be used to provide a measure of the performance of the new method for video frame interpolation and event synthesis, respectively. The accuracy of interpolated frames is also included here, because this is a good indicator of the event quality: If the interpolated frames are accurate, the information is correctly modelled and can be used to generate events.

**Research Question 4**

Can the continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be used to upsample the time resolution of video accurately?

Even with accurate frames, it is important to verify the accuracy of the generated events themselves. Research Question 5 will quantify the accuracy of the generated events.

**Research Question 5**

Can the continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be used to synthesise events accurately?

These five Research Questions will quantify the performance of a new way to perform continuous video frame interpolation and the new, faster, event camera simulator outlined in the Research Objective.

## 3.2. Problem statement

From the literature study, it can be concluded that the current method of generating events, using difference frames of high frame-rate video, either directly rendered or temporally upsampled, is too slow to be used for real-time applications and imposes a significant computational burden when generating event datasets. A new method should be found that is fast, while still being accurate. This will allow for real-time training using simulated event-based cameras and will allow for quick generation of event datasets for other purposes.

## 3.3. Structure and Scope of this Work

The new interpolation method will be developed in several stages. After the method is designed mathematically, an initial prototype will be developed in Python. This prototype will be used to verify the correctness of the method. Because the method requires many calculations in parallel, it is well suited to be computed on a GPU. As a stepping stone to be implemented on the GPU, the method will first be implemented in C++. Finally, this implementation will be converted to a GPU version, such that it can efficiently used.

Based on the idealised ESIM simulator, many works have improved the accuracy and realism of the generated events by introducing various noise modelling strategies, non-linearities, and other effects. This work will go back to ESIM, and develop a faster simulator method, that can generate idealised events fast. The developed method will work both on existing frames and directly in a rendering pipeline. Later, in future research, these effects can be re-applied to the output of the fast simulator.

# Part II

## Main Work

# 4

# Methodology

Looking through the perspective of a single pixel in a camera, all it can 'see' is a single intensity value. As this intensity value changes over time, an Intensity over Time ($I/t$) curve appears. Events are generated by each pixel from the difference in light intensity over time. In order to simulate the generation of events, it is therefore vital to model a representation of this $I/t$ curve for each pixel, illustrated in Fig. 4.1. From this curve, the event generation is trivial. The intensity is one of the various pixel attributes that can be modelled between frames. Others include colour, depth, or the flow information itself. In this chapter, intensity will be used as an example, but the process is the same for all other pixel attributes.

Various methods can be imagined to obtain these $I/t$ curves. As seen in Section 2.6, previous methods have obtained the curves with fast sampling, such that a new sample is obtained fast enough for a good reproduction of the underlying curve. This is usually done at a rate twice that of the fastest motion in the scene [3]. However, for each sample, the entire frame needs to be rendered. As many samples are required, these methods are very computationally expensive. By interpolating between samples, a lower sample rate should still be able to reconstruct the intensity curve. Some methods have taken this approach, using existing VFI methods. These methods can generate new frames in between the low frequency frames. Still, entire frames are generated above the rate of the fastest moving object, However, this object will usually not occupy the whole frame, resulting in many regions that are oversampled. By tightly coupling the interpolation scheme to the pixel level, this unnecessary work can be eliminated.

This chapter will describe the method that has been developed to achieve this. In Section 4.1, the concept of continuous-time frame interpolation will be introduced. Afterwards, in Section 4.2, all necessary steps in the new grid-based frame interpolation method will be mathematically defined. Finally, in Sections 4.3 and 4.4, the last step from $I/t$ curve to interpolated frames or events will be shown

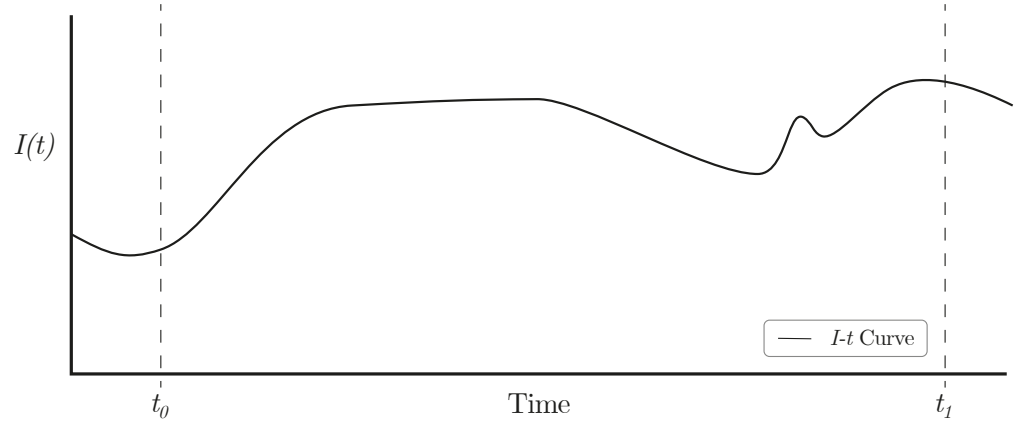
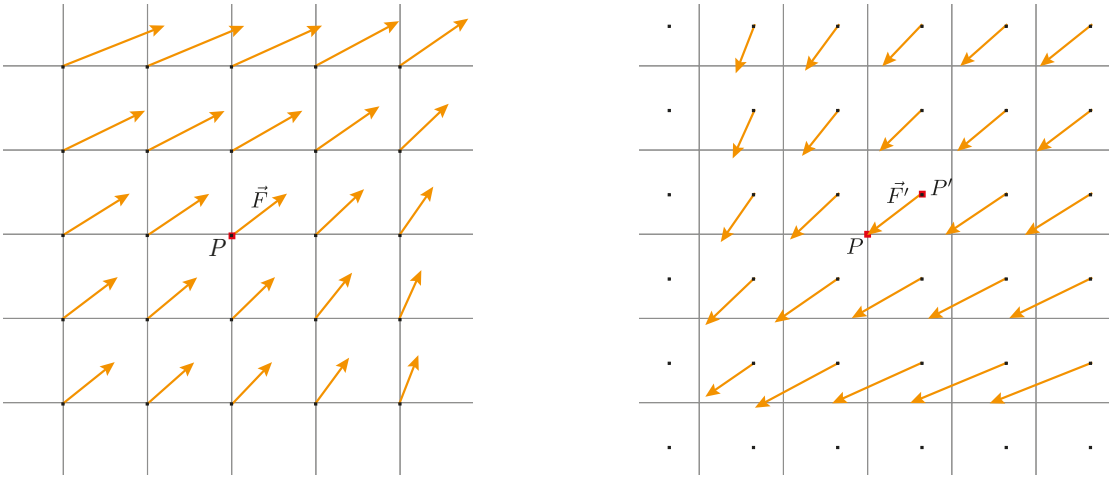



**Figure 4.1:** *Example of an $I/t$ curve modelling the the change in intensity over time.*




**(a)** *The forward optical flow field $\vec{F}$, originating from the original pixels $P$ towards the movement of the pixels in the frame.*

**(b)** *The backward optical flow field $\vec{F'}$, originating from the interpolated pixels $P'$ and pointing back at the source of these pixels in the original frame.*

**Figure 4.2:** *The subtle difference between the forward and backward optical flow fields, $\vec{F}$ and $\vec{F'}$*

## 4.1. Continuous-Time Pixel Attribute Interpolation

The optical flow of an image can be used to interpolate between two frames, $I_0$ and $I_1$, taken at $t_0$ and $t_1$. Under the brightness constancy assumption[1], the forward optical flow $\vec{F}$ (Fig. 4.2a) is defined as vector from a pixel in the original frame to its new location and the backward optical flow $\vec{F'}$ (Fig. 4.2b) is defined as the vector from a pixel in the interpolated frame to its origin in the original frame. The subtle difference between these two flow fields comes from the place at which they are defined, either flowing from the original pixels or from the interpolated pixels. At any time $t$ in-between the two frames, an interpolated frame can be obtained for each pixel by sampling from the frame at $t_0$ on the location defined by the backward optical flow vector $\vec{F'}_{t\to 0}$ or sampling from the frame at $t_1$ using $\vec{F'}_{t\to 1}$. The value of a pixel in the interpolated frame at $t$ is defined as the value sampled from the location that this pixel has moved from in the original frame. Due to occlusions and other visual effects, these optical flow vector fields are unique for each $t$ and can be discontinuous.

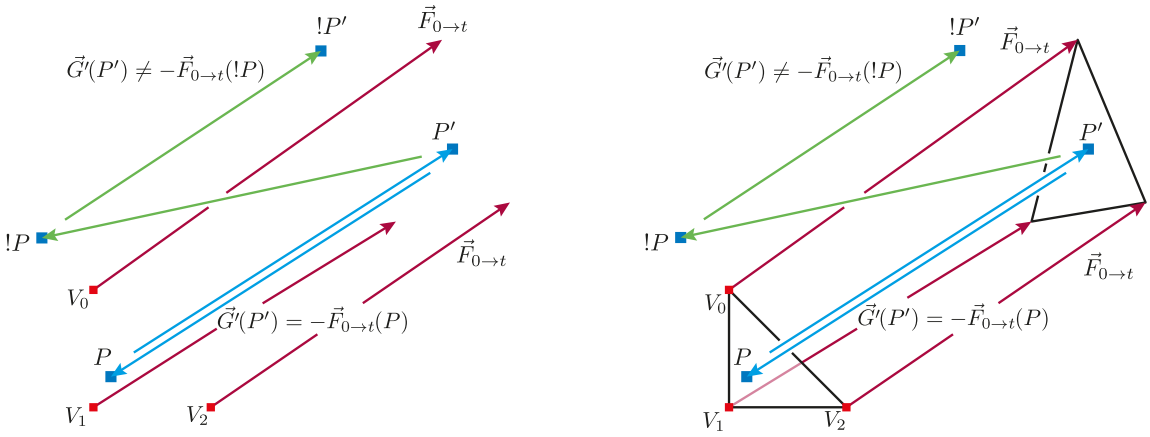
The backward optical flow fields $\vec{F'}_{t\to 0}$ and $\vec{F'}_{t\to 1}$ need to be found from the available optical flow fields, $\vec{F}_{0\to 1}$ and $\vec{F}_{1\to 0}$. One naive method of obtaining $\vec{F'}_{t\to 0}$ from $\vec{F}_{0\to 1}$ is using forward splatting. In this method, optical flow between the frames is scaled to time $t$ to obtain a forward optical flow map:

$$\vec{F}_{0\to t} = \vec{F}_{0\to 1}\frac{t-t_0}{t_1-t_0} \tag{4.1}$$

By warping each pixel with the forward optical flow, it is splatted onto the interpolated frame. To create a new frame from these splatted pixels, and obtain $\vec{F'}_{t\to 0}$, these pixels are distributed to the nearest four pixels using bilinear weights. A merging strategy is needed to determine how the different splatted pixels onto the new pixels should be combined. When varying $t$ between $t_0$ and $t_1$, this results in pixels that move through the squares defined between the pixel grid. The path of each pixel is defined by the entry and exit point of these pixels.

Problems arise with this method when the optical flow diverges or converges, as there will either be too little or too many pixels to construct new pixels. When there are too little pixels, a hole will appear in the reconstructed frame. When there are too many pixels, it is required to determine which pixels should be shown and which should be obscured. Solving this is not straightforward, since it depends on the depth, distance from the pixel centre, and connectivity with pixels around it. If two warped pixels are connected, the area between them should be filled in and occlude pixels that have a greater depth. However, if they were not connected, the pixels behind them should remain visible. Without the knowledge of connectivity

[1] See Section 2.1.




**(a)** *Without the connected triangle, it is hard to find the correct mapping between the original and interpolated frame.*

**(b)** *The connected triangle provides a clear mapping at any point in the interpolated frame.*

**Figure 4.3:** *A mapping is required between the original frame and the interpolated frame. The correct backwards optical flow needs to be found using the available forward optical flow and depth maps, defined on the pixels in the original frame. The blue mapping is correct, since the forward optical flow points from $P$ points back at $P'$, but the green pair is incorrect, since the optical flow from $!P$ does not point back to $P'$*

between the pixels, the blending strategy cannot work without producing visible artifacts.

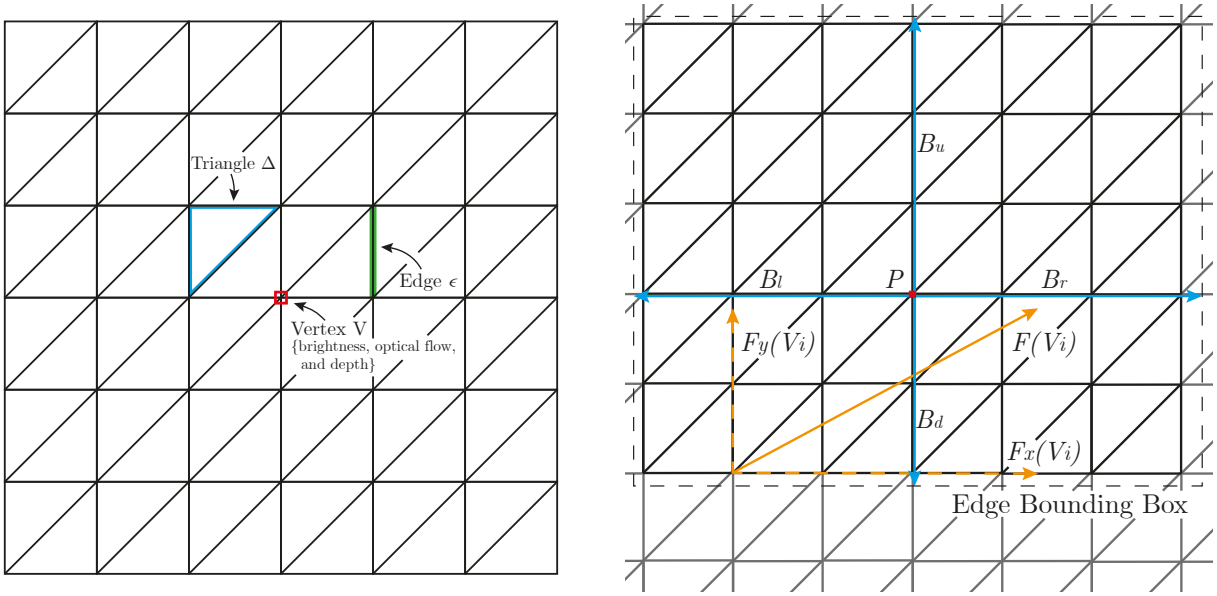
Other practical problems remain as well, such as the determination of the area of influence of each pixel. Because of the bilinear splatting, any value that is splatted within the square of pixels around this pixel contributes to the value of the new pixel. This technique assumes that, in a patch of continuous movement, when one pixel leaves the area of influence of a pixel, another will enter from the other side, so the summed value will stay constant. However, with errors in the optical flow and the imprecision of floating point numbers, these pixels will not be neatly aligned, which will cause discontinuities and errors in the $I/t$ curve. These problems occur because the computed backwards flow is ambiguous and does not allow for a true sampling of the value in the original frame.

In general, the problem that needs to be solved is to find a transformation $\vec{G}'$ from the point $P'$ in the interpolated frame to the point $P$ in the original frame, such that the known forward flow $\vec{F}$ from this location points back to $P'$:

$$\vec{G}'(P') = -\vec{F}\Big(\vec{G}'(P')\Big) \tag{4.2}$$

$\vec{F}$ is generally not defined at the point $P$ that $\vec{G}'(P')$ ends up at, and therefore needs to be interpolated from the nearest pixels in the original frame where forward optical flow field $\vec{F}$ is defined. In Fig. 4.3a, this principle is shown: When $\vec{G}'$ is correct, the forward and backward flows point at each other, otherwise, the forward flow will point in another direction. If the found transformation $\vec{G}'(P')$ satisfies this equation, it is equal to the backward optical flow field $\vec{F}'$.

To overcome these problems and remove the splatting ambiguity, a novel triangle mesh-based warping method is proposed. In this method, each pixel in the original frame is seen as a vertex in a grid of triangles. These vertices are given the properties of the pixels. This grid is deformed using the forward optical flow $\vec{F}_{0\to 1}$, scaled by the evaluation time. At any time $t$, this grid is fully defined. Each pixel location in the interpolated image is inside of one or multiple triangles. Like a rasterising rendering engine, at any arbitrary time $t$, each pixel can determine where it should sample from in the original image by determining its




**(a)** *A section of the triangle grid, with vertices $V$, edges $\epsilon$, and triangles $\Delta$.*

**(b)** *The bounding box that is considered for a single pixel. An example flow vector $\vec{F}(V_i)$ has been drawn on a random vertex $V_i$, with its horizontal and vertical components, $\vec{F}_x(V_i)$ and $\vec{F}_y(V_i)$*

**Figure 4.4:** *The continuous frame interpolation method uses a grid of triangles, where each pixel of the image becomes a vertex of the grid.*

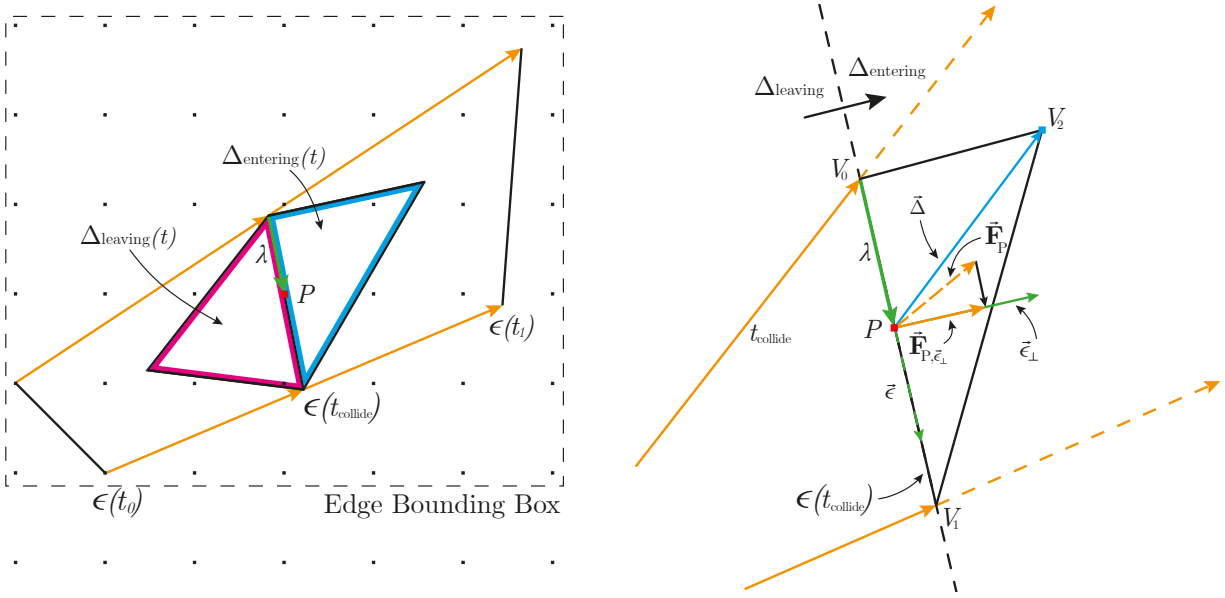
location in the triangle that it is inside that is the closest to the camera. When the triangle is added in Fig. 4.3b, the correct transformation becomes clear. Because each triangle has a direct mapping to the original frame, all locations in the interpolated frame can be mapped to a location in the original frame: this is the $\vec{G}'$ that is desired and equal to the backward optical flow $\vec{F}'_{t\to 0}$.

## 4.2. Mesh-Based Frame Interpolation

In order to interpolate from a frame, the various render passes generated by the rendering engine are converted to a mesh-like grid representation. The render passes contain the brightness, depth, and forward optical flow information for all rendered pixels. This information is represented in the triangle grid. This grid, shown in Fig. 4.4a, consists of vertices $V$, each containing a brightness $I$, depth $d$, and an optical flow vector $\vec{F}$. On each grid-square, the vertices are connected into two triangles $\Delta$, one upper-left triangle $\Delta_{\text{upper}}$ and one lower-right triangle $\Delta_{\text{lower}}$. Each triangle consist of three edges $\epsilon$, that are shared with the neighbouring triangles.

The forward optical flow defines the movement of each vertex. Each moves from their starting point of the grid at $t_0$ in the direction of the vector, until it reaches the end of the vector at $t_1$. This results in a moving grid, of which the triangles change shape and size based on the movement of the vertices. At each $t$, the grid will have a defined shape. On the original pixel locations $\vec{P}$, new pixel values need to be determined to construct a new image at each $t$. For each pixel, this value can be determined if the triangle that currently overlaps this location is known. Each time an edge crosses a pixel, this pixel moves from being in one triangle to the next. So, an efficient method to determine the occupied triangles at each $t$ between $t_0$ and $t_1$ is to determine all triangle switches in that time. In turn, these triangle switches occur when a pixel hits an edge of the grid.

For each pixel in the image, all edges in a bounding box around the pixel, shown in Fig. 4.4b, are tested for collisions. The size of this bounding box is determined by the local maximum optical flow. Any edge that starts further away from the pixel than the local maximum cannot reach the pixel, and therefore does not have to be tested. The maximum optical flow is measured in the four axis directions, $\{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$, and the bounding box is sized to these four distances around the pixel: $\{B_u, B_d, B_l, B_r\}$. In order to reduce the amount edges to be checked, the whole image is split up into fixed size windows. For each window,




**(a)** *The collision between a moving edge $\epsilon$ and the point $P$ is evaluated. The orange vectors show the optical flow vectors of the edge. When the edge is at $\epsilon(t_{collide})$, it collides with the point. The collision occurs at fraction $\lambda$ between the two vertices of the edge.*

**(b)** *When a collision has been found, for each connected triangle, it is determined on which side of the moving edge it is. This is calculated with the dot product between the vector to the centre of the triangle and the vector that is perpendicular to the edge in the direction of travel.*

**Figure 4.5:** *A geometric representation of the edge collisions.*

the local maximum of optical flow is used to determine the bounding box sizes around each pixel.

### 4.2.1. Edge Collisions

Testing the collision between a moving edge and a stationary point involves solving two parameters: the time $t$ at which the edge coincides with the point and the fraction $\lambda$ of the edge at which the collision occurs. The position of an edge is defined by the movement of its two vertices, each in turn defined by their optical flow. Each point $\vec{H}$ in the swept area of the moving edge can thus be described by the following vector equation:

$$\vec{H} = (1-\lambda)\left[\vec{V}_0 + t\vec{F}_{\vec{V}_0}\right] + \lambda\left[\vec{V}_1 + t\vec{F}_{\vec{V}_1}\right] \tag{4.3}$$

where $\vec{V}_0$ and $\vec{V}_1$ are the original locations of the vertices and $\vec{F}_{\vec{V}_i}$ is the optical flow that is defined for each vertex. Using this definition, all $t$ and $\lambda$ combinations that are between $0$ and $1$ correspond to points that fall into the swept area of the edge and thus, if a pixel is located at such a point, would provide a valid collision between the edge and the pixel. In Fig. 4.5a, the swept area of edge $\epsilon$ between $t_0$ and $t_1$ is drawn, with the optical flow vectors of the vertices in orange. The pixel that is currently being evaluated is shown as $P$. At $t_{\text{collide}}$, the edge collides with this pixel, at a certain $\lambda$. To find these $t$ and $\lambda$, the solutions need to be found for when $\vec{H}$ is equal to the known pixel location $\vec{P}$:

$$\vec{P} = \vec{H} \quad \rightarrow \quad \begin{bmatrix} P_x \\ P_y \end{bmatrix} = (1-\lambda)\left(\begin{bmatrix} V_{0,x} \\ V_{0,y} \end{bmatrix} + t\begin{bmatrix} F_{V_0,x}V_{0,x} \\ F_{V_0,y}V_{0,y} \end{bmatrix}\right) + \lambda\left(\begin{bmatrix} V_{1,x} \\ V_{1,y} \end{bmatrix} + t\begin{bmatrix} F_{V_1,x}V_{1,x} \\ F_{V_1,y}V_{1,y} \end{bmatrix}\right) \tag{4.4}$$

This vector equation can be solved for $t$ and $\lambda$:

$$At^2 + Bt + C = 0 \quad \rightarrow \quad t = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \tag{4.5}$$

$$A = F_{V_1,y}\left(F_{V_0,x} - F_{V_1,x}\right) - F_{V_1,x}\left(F_{V_0,y} - F_{V_1,y}\right) \tag{4.6}$$

$$\begin{aligned} B = {} & F_{V_1,y}\left(V_{0,x} - V_{1,x}\right) - F_{V_1,x}\left(V_{0,y} - V_{1,y}\right) \\ & + \left(P_x - V_{1,x}\right)\left(F_{V_0,y} - F_{V_1,y}\right) - \left(P_y - V_{1,y}\right)\left(F_{V_0,x} - F_{V_1,x}\right) \end{aligned} \tag{4.7}$$

$$C = \left(P_x - V_{1,x}\right)\left(V_{0,y} - V_{1,y}\right) - \left(P_y - V_{1,y}\right)\left(V_{0,x} - V_{1,x}\right) \tag{4.8}$$

$$\lambda = \frac{\left(P_x - V_{0,x}\right) - F_{V_1,x}t}{\left(V_{0,x} - V_{1,x}\right) + \left(F_{V_0,x} - F_{V_1,x}\right)t} \tag{4.9}$$

Here, $t$ is the solution to the quadratic formula, using the coefficients $A$, $B$, and $C$; each defined by the known properties of the edge. This means that there are sometimes two collisions: geometrically, this can happen if the optical flow vectors are opposed or crossed, as some part of the area is swept twice in these cases. $\lambda$ follows from each $t$. If $t$ and $\lambda$ are both within the range of $0..1$, this edge collides with this pixel. $t$ determines the distance that the edge has travelled when it hits this vertex and $\lambda$ determines where along the edge this collision occurs.

The edge is part of two triangles, one of which was overlapping the pixel before the collision happened, and the other will be overlapping the pixel after the collision has happened. To create a full travel history of each pixel, it is important to determine which of the triangles is which. For both triangles, the dot product between the vector to the third, opposing, vertex of the triangle and the vector perpendicular to the edge in the direction of travel is used to determine the side of the edge that this triangle is on. In Fig. 4.5b, a geometric representation of this determination is shown, where the triangle is determined to be an entering triangle, because it is one the side of the edge that it will travel towards, determined by the orange motion vectors. Since $t$ and $\lambda$ are known, the direction of the edge at this point, $\vec{\epsilon}$ is also known. The flow vector is interpolated between the flow vectors of the two vertices of the edge to the collision point, resulting in the local flow vector $\vec{F}_P$. This vector is projected to the vector perpendicular to the edge direction, $\vec{\epsilon}_\perp$, resulting in the vector that is perpendicular to the edge in the direction of travel, $\vec{F}_{P,\vec{\epsilon}_\perp}$. Using the dot product with the triangle vector $\vec{\Delta}$, it is determined on which side each triangle is. Mathematically, this is determined by the following test:

$$\begin{cases} \vec{F}_{P,\vec{\epsilon}_\perp} \cdot \vec{\Delta} < 0 & \rightarrow \quad \Delta = \Delta_{\text{entering}} \\ \vec{F}_{P,\vec{\epsilon}_\perp} \cdot \vec{\Delta} > 0 & \rightarrow \quad \Delta = \Delta_{\text{leaving}} \end{cases} \tag{4.10}$$

where:

$$\vec{F}_{P,\vec{\epsilon}_\perp} = \vec{F}_P - \texttt{proj}_{\vec{\epsilon}}\,\vec{F}_P \;=\; \vec{F}_P - \hat{\epsilon}\left(\vec{F}_P \cdot \hat{\epsilon}\right) \tag{4.11}$$

$$\vec{F}_P = (1-\lambda)\;\vec{F}_{\vec{V}_0} + \lambda\,\vec{F}_{\vec{V}_1} \tag{4.12}$$

$$\vec{\epsilon} = \left[\vec{V}_1 + t\vec{F}_{\vec{V}_1}\right] - \left[\vec{V}_1 + t\vec{F}_{\vec{V}_0}\right] \tag{4.13}$$

$$\hat{\epsilon} = \frac{\vec{\epsilon}}{\|\,\vec{\epsilon}\,\|} \tag{4.14}$$

In the case that $\vec{F}_{P,\vec{\epsilon}_\perp} \cdot \vec{\Delta} = 0$, the entering or leaving triangle cannot be determined at this point. When this occurs, the collision is shifted back in time until a clear entering and leaving triangle can be discerned. If no result can be obtained, the collision is discarded. This value can be zero when triangles have exactly no area at the time of the collision. In regions with contracting or discontinuous flow, triangles can become inverted. After the area of a triangle has been inverted, it is ignored from the point of inversion.

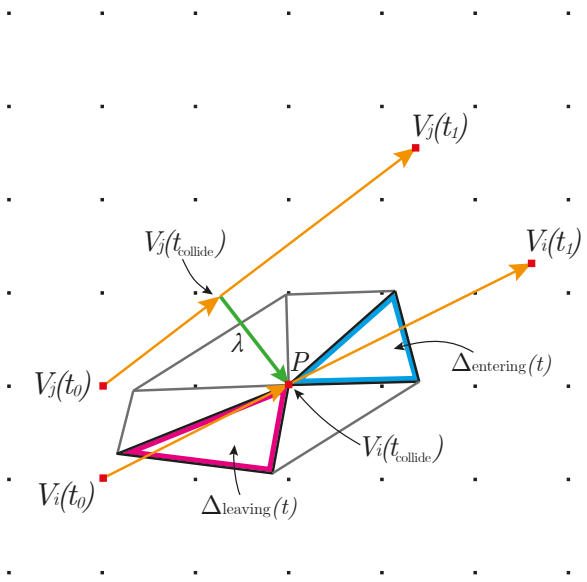



**Figure 4.6:** *Vertex collisions require special handling, as the edge algorithm cannot find the leaving or entering triangle. The vertex collisions are found by their parallel distance,*

### 4.2.2. Vertex Collisions

A special case occurs when a vertex exactly hits a pixel. In this case, when $\lambda$ is either $0$ or $1$, the edge collision method cannot determine which triangle is the leaving and which is the entering triangle, because it could be any of the six triangles that are adjacent to this vertex. In Fig. 4.6, this case is shown for two example vertices, $V_a$ and $V_b$. In real images, this exact collision can happen by coincidence, but it is more likely to occur when one of the components of the optical flow of a pixel is zero, because the flow is either perfectly horizontal or vertical. A initial pass is conducted for each pixel, in which it looks for vertex collisions with all vertices that can reach this pixel. In the example, during their travel from $t_0$ to $t_1$, $V_a$ does hit a pixel and $V_b$ does not. This is calculated by looking at the minimum parallel distance $\lambda$ between the pixel $P$ and the path of the vertex. This minimum distance occurs when the vertex is perpendicular to the pixel, called $V_i(t_{\text{collide}})$. The following equation describes this position:

$$t\vec{F}_{V_i} + \lambda \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \frac{\vec{F}_{V_i}}{\| \vec{F}_{V_i} \|} = \vec{p} \tag{4.15}$$

where $\vec{F}_{V_i}$ is the flow vector from the vertex $V_i$ and $\vec{p}$ is the known vector from the vertex position at $t_0$ to the pixel position:

$$\vec{p} = \vec{P} - \vec{V}_i(t_0) \tag{4.16}$$

Solving Eq. (4.15), the distance is calculated as follows:

$$\lambda = \frac{p_y F_{V_i,x} - p_x F_{V_i,y}}{\|\vec{F}_{V_i}\|} \tag{4.17}$$

If the distance is zero ($\lambda = 0$), the time of the collision can be calculated:

$$t = \frac{\vec{p} \cdot \vec{F}_{V_i}}{\|\vec{F}_{V_i}\|^2} \tag{4.18}$$

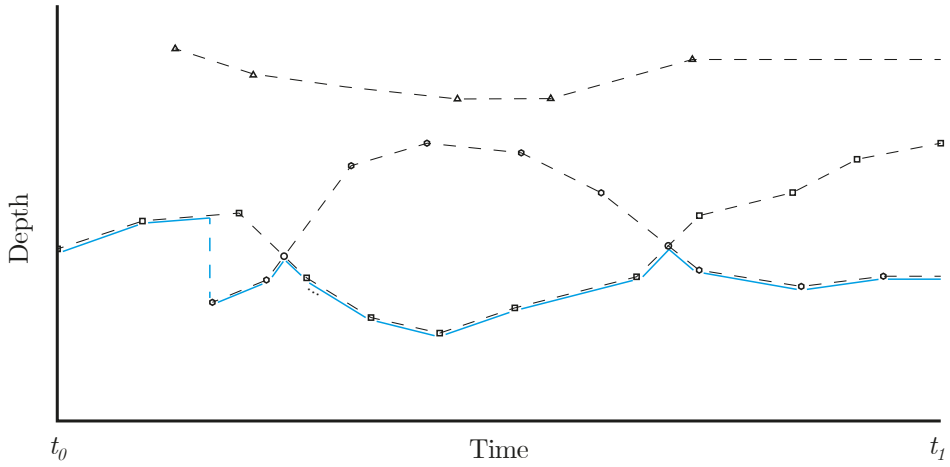



**Figure 4.7:** *Collisions are chained based on the connected triangles. The chains are converted to control points using the depth value. In this case, there are three chains, that are combined into control points, denoted by the cyan line.*

Since these exact vertex collisions are rare with respect to edge collisions and a geometric method, like the edge direction algorithm, would be more computationally expensive, a simpler method is used to determine the leaving and entering triangle: It is tested whether the pixel is inside each triangle slightly before and after the collision time using the pixel position in barycentric coordinates of each triangle.

### 4.2.3. Chaining Collisions into Control Points

These collisions generated by vertices and edges subsequently combined into chains of collisions, by matching the leaving triangle from one collision with the entering triangle of the next collision. For each pixel, one or more chains are generated. These chains are defined as sequences of collisions, where each chain represents the movement of a part of the grid over a pixel. Since the location of each collision is known, the intensity and depth at these points can be found. Therefore, each link in a chain of length $N$ has a time $t$, an intensity $I$, and a depth $z$:

$$[\{t_0, I_0, z_0\}, \{t_1, I_1, z_1\}, \{t_2, I_2, z_2\}, ..., \{t_n, I_n, z_n\}, ..., \{t_N, I_N, z_N\}] \tag{4.19}$$

There can be multiple chains of varying time regions and chain length for each pixel. At every point in time, the chain that is closest to the camera should be visible. By merging all chains of a pixel, selecting only the closest chain at each collision using the known depth of the collisions, a single sequence of control points can be generated for each pixel. These control points contain a colour or intensity value and a time. Together, the control points of a pixel approximate the $I/t$ curve of that pixel as a piecewise linear function. In Fig. 4.7, this process is shown. This pixel has three chains, that are combined into one sequence of control points. These control points consist of a time $t$ and intensity $I$, completely describing the $I/t$ curve:

$$[\{t_0, I_0\}, \{t_1, I_1\}, \{t_2, I_2\}, \{t_3, I_3\}, ..., \{t_n, I_n\}, ..., \{t_N, I_N\}] \tag{4.20}$$

Changes in the $I/t$ curve are governed by the pixels, which are modelled with triangles. In areas with more active pixels, the pixels will switch triangles more often, resulting in more control points, while areas that are less active will generate less control points. In Fig. 4.8, it can be seen how these control points will model the $I/t$ curve.

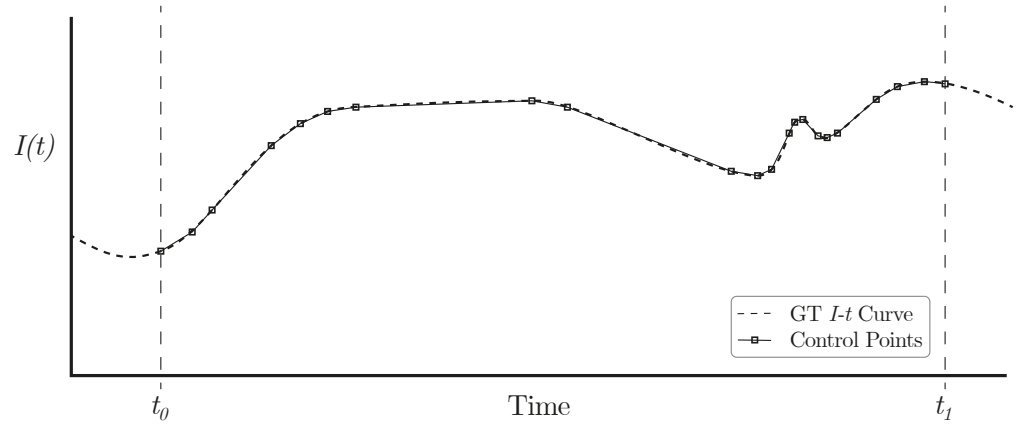



**Figure 4.8:** *Illustration of control points with linear connections modelling the $I/t$ curve. Due to the nature of the method, when the $I/t$ curve changes often, there will also be more control points to model it.*

### 4.2.4. Bi-Directional Interpolation

When moving the grid forwards in time along the flow direction, artifacts can appear. Regions of occlusion are handled correctly, but artifacts appear in regions of accretion, regions that were previously out of frame, or that were otherwise not visible in the first frame, but are in the second. Because there is no information in these regions, they cannot accurately be interpolated by just this pass. Because of these occlusion and accretion regions, in general, these flow fields are not the inverse of each other:

$$\vec{F}_{0\to1} \neq -\vec{F}_{1\to0} \tag{4.21}$$

These problematic regions of accretion become regions of occlusion when time is reversed. So, in this reverse time, this information is present. By not only using the forwards flow field $\vec{F}_{0\to1}$ from $I_0$ to $I_1$ in a forward pass, where the intensity is sampled from $I_0$, but also the reverse forward flow $\vec{F}_{1\to0}$ from $I_1$ to $I_0$ to compute a backwards pass, where the intensity is sampled from $I_1$ instead, the regions that include artifacts due to accretion in the forward pass can be filled in with correct information from the complementary backward pass.

Merging the forward and backward passes provides a full interpolation without accretion artifacts. In the triangle grid, when there is divergence of the flow, the spacing between the vertices increases. This means that triangles are bigger when there is less information in this pass. Therefore, when merging, for each evaluated pixel, the smallest triangle has the most weight in the combined value. On the boundary between two diverging regions, one row of very large triangles appears. These triangles are excluded and replaced by smaller triangles with information from the other pass. This is combined with the amount of time that this pass is from its origin frame and used to calculate a weight for each pixel in both passes. The passes are then merged using these weights:

$$w_{0\to1} = \frac{\sqrt{1-t}}{\exp\left(k\left|A_{\Delta_{0\to1}}(t)\right|\right)} \tag{4.22}$$

$$w_{1\to0} = \frac{\sqrt{t}}{\exp\left(k\left|A_{\Delta_{1\to0}}(t)\right|\right)} \tag{4.23}$$

$$I_{\text{merged}}(t) = \frac{w_{0\to1}}{w_{0\to1}+w_{1\to0}} \cdot I_{0\to1}(t) + \frac{w_{1\to0}}{w_{0\to1}+w_{1\to0}} \cdot I_{1\to0}(t) \tag{4.24}$$

where $A_\Delta(t)$ denotes the area of a triangle at time $t$ and $k$ is a constant parameter to determine the influence of the area on the weight.

## 4.3. Frame Interpolation from Intensity-Time Curves

Sampling the merged $I/t$ curve at any $t$ between $0$ and $1$ for each pixel, the frame at that time $t$ can be constructed. Once the curve for each pixel has been calculated, the evaluation is very simple, which allows for many frames to be interpolated quickly between the two original frames. In this case, a linear interpolation between the nearest control points is used, but more advanced interpolation schemes, that use more neighbouring control points, could yield improved results.

$$I(t) = I_n + \frac{t - t_n}{t_{n+1} - t_n} (I_{n+1} - I_n) \tag{4.25}$$

where $n$ is the index of the first control point just before $t$ and $n + 1$ is the index of the first control point just after $t$. $t_n/t_{n+1}$ and $I_n/I_{n+1}$ denote the time and intensity at $n/n + 1$, respectively.

## 4.4. Event Generation from Intensity-Time Curves

Knowing the continuous $I/t$ curve, the exact event locations in $t$ can be quickly sampled, by using the slope between the control points. At any time $t$, there is a control point before and after $t$: $t_n \leq t \leq t_{n+1}$. From the current stored $I$, obtained at the last triggered event, the moment that the threshold is reached at $I + C_P$ or $I - C_N$ can be calculated:

$$t_{\text{event}} = t_n + (I_{n+1} - I + C_P) \left[\frac{dI}{dt}\right]_{n \to n+1} \quad \textbf{or} \quad t_{\text{event}} = t_n + (I_{n+1} - I - C_N) \left[\frac{dI}{dt}\right]_{n \to n+1} \tag{4.26}$$

where $n$ is the index of the current control point and $n + 1$ is the next control point, $t_n$ is the time at the current control point, $I_{n+1}$ is the intensity of the next control point, and $\left[\frac{dI}{dt}\right]_{n \to n+1}$ is the slope of between control point $n$ and control point $n+1$. If the slope is positive, the positive threshold is used, and the polarity $p$ of the upcoming event is positive, otherwise, the negative threshold is used and the polarity is negative. If the $t_{\text{event}} \leq t_{n+1}$, a new event is fired for this pixel at $t_{\text{event}}$. This process is repeated from $t_0$ to $t_1$.

# 5 Implementation

The new method is implemented in two ways: a stand-alone version and a real-time version in Unreal Engine 5. The stand-alone version can be used to interpolate existing frames and convert them into events. Additionally, it can be connected to another pipeline using the Pybind11[1] binding to Python or using command line interface in C++. For direct integration in training setups, an Unreal Engine 5 plug-in is written, that can apply the method inline with the rendering process. The stand-alone version is elaborated in Section 5.1 and the plug-in is detailed in Section 5.2.

## 5.1. Implementation of Grid-Based Frame Interpolation

The method described in Chapter 4 is implemented as stand-alone program in C++[2]. This implementation is first written as a multithreaded CPU version to test the method on larger images and to prepare for GPU acceleration in CUDA later. This version takes two colour frames, two flow frames, and two depth frames as input, one of each at $t_0$ and one at $t_1$. This information is used to interpolate the $I/t$ curve between $t_0$ and $t_1$, and to convert this curve to events. Inside the pipeline, the local $t$ is defined to be $0$ at $t_0$ and $1$ at $t_1$. The full pipeline of the CPU version is shown in Fig. 5.1.

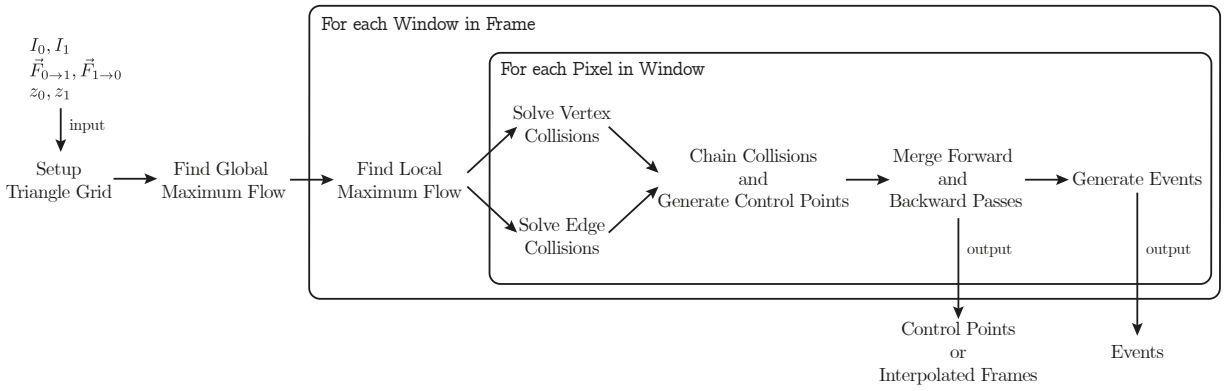



**Figure 5.1:** *The pipeline implemented in C++.*

The collision algorithm can be implemented in two ways: Evaluating a vertex or edge, and searching for the pixels that this vertex or edge would collide with, or searching for all vertices and edges around a pixel that collide with this pixel. While the amount of collision checks in the first method is smaller, the result is a list of collisions for each edge, and in the second the result is a list of collisions for each pixel. Since the second list is needed for next step in the algorithm, this method is preferred. With the other method, the collisions have to be re-ordered and redistributed, which is very costly on the GPU. The chosen method can be calculated completely independently for each vertex, without having to redistribute intermediate results.

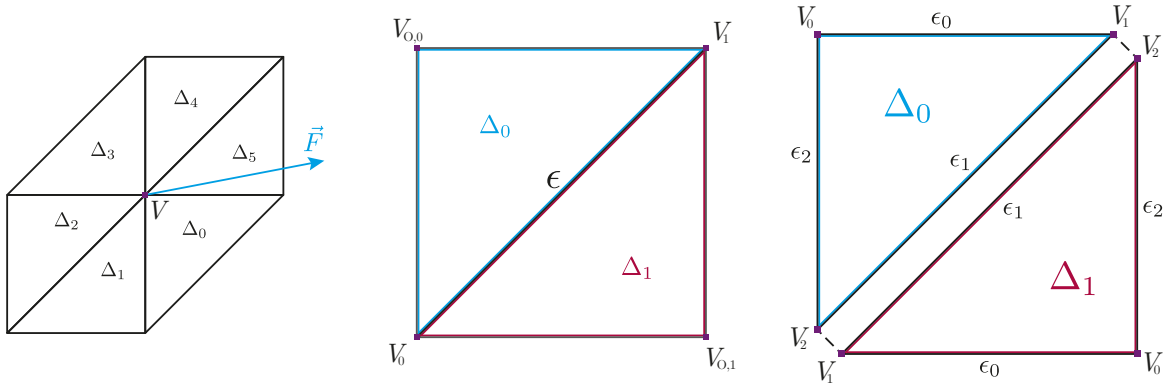
[1] Python wrapper for C++ code: pybind11.com

[2] Full code is available on github.com/ReinierZwikker/FastEventSim

### 5.1.1. Triangle Grid Generation

Before the calculation can begin, first, the grid is constructed in memory, by setting up all links between the parts of the grid. The grid consists of vertices, that each have position $[x, y]$, colour $I$, flow $\vec{F}$, and depth $z$ information. The method is implemented here to use interpolate the three values in each RGB pixel. Next to these properties, the vertex has a reference to all six triangles that use this vertex. These references are used to quickly traverse the grid later, for example when all triangles of a vertex need to be tested. On top of the vertices, the grid is extended with one more row of vertices on each outside border, to correctly handle the case when the edge of the frame becomes visible in the interpolated frame. Each edge contains references to the vertices that make up this edge and the two triangles that it is part of. Next to this, each edge stores a reference to the third vertex in each triangle, that is not part of this edge: $V_O$. This opposite vertex is used when the direction of the triangle relative to the edge needs to be calculated. The triangles contain references to their three vertices and their three edges. Each triangle calculates the time $t_{\text{max}}$ that it becomes invalid: As can be seen in Fig. 5.3, at an occlusion boundary, one row of triangles becomes inverted. These triangles should be ignored. This time is determined by testing when the signed area of a triangle becomes negative. For many triangles, this never occurs. For both the forward and backward passes, a separate, independent, grid is created. These grids operate exactly the same, only differing on received input. These grids are used to perform all further calculations.

$$\begin{aligned}
\text{Vertex} : V &= \left\{ [x, y], I, \vec{F}, z, [\Delta_0, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5] \right\} \\
\text{Edge} : \epsilon &= \{ [V_0, V_1], [\Delta_0, \Delta_1], [V_{O,0}, V_{O,1}] \} \\
\text{Triangle} : \Delta &= \{ [V_0, V_1, V_2], [\epsilon_0, \epsilon_1, \epsilon_2], t_{\text{max}} \}
\end{aligned}$$




**(a)** *A vertex of the grid, with a reference to each triangle* $[\Delta_0, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5]$ *that use this vertex. The flow vector of this vertex* $\vec{F}$ *is shown.*

**(b)** *An edge of the grid, with a reference to each vertex* $[V_0, V_1]$ *and each triangle* $[\Delta_0, \Delta_1]$ *that make up this edge. Additionally, the opposite vertices of these triangles are referenced:* $[V_{O,0}, V_{O,1}]$

**(c)** *Two triangles of the grid, with reference to each vertex* $[V_0, V_1, V_2]$*, and each edge* $[\epsilon_0, \epsilon_1, \epsilon_2]$*.* $(V_1$ *of* $\Delta_0)$ *is a reference to the same vertex as* $(V_2$ *of* $\Delta_1)$*,* $(V_2$ *of* $\Delta_0) = (V_1$ *of* $\Delta_1)$*, and* $(\epsilon_1$ *of* $\Delta_0) = (\epsilon_1$ *of* $\Delta_1)$

**Figure 5.2:** *A vertex, edge, and triangle of the triangle grid, with the geometric attributes stored with each part.*

Next, the global maximum window size is determined. This is found from the maximum optical flow in each axis direction: $\{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$. This value determines the maximum search area for each window. The size of the windows themselves is fixed and given as input. The appropriate size of these windows depends on the resolution of the image and the variety of the magnitude of the optical flow throughout the image. When the optical flow variety is low, large windows can be chosen, since each window will have similar sized bounding boxes, however, when the variety is large, smaller windows are more appropriate, because the varied bounding box size will allow for many edges to be excluded from collision checking, because these edges cannot reach the pixel.

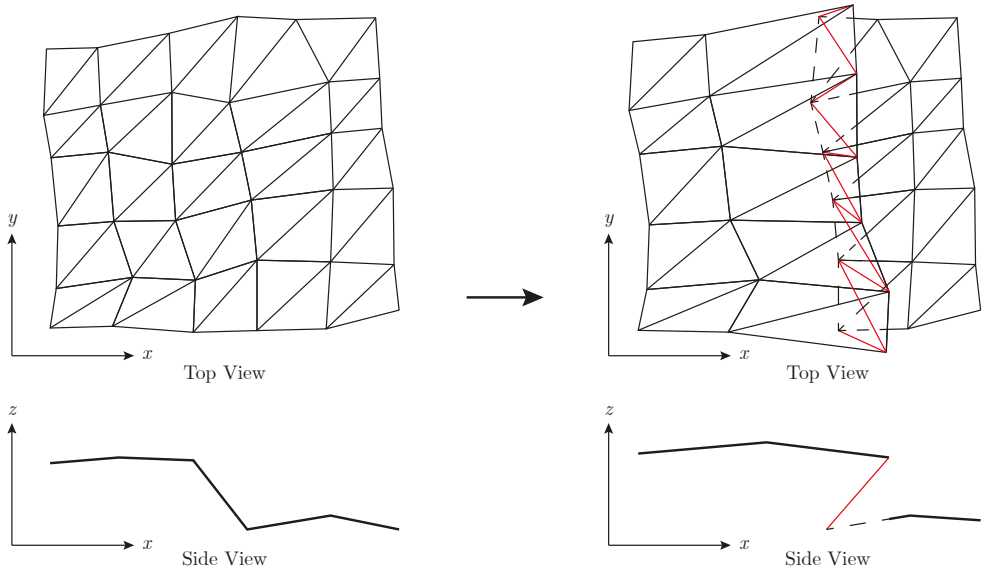



**Figure 5.3:** *A region of occlusion, where an object that is closer to the camera obscures another. The inverted triangles are removed and the occluded regions of the bottom triangles are hidden by the upper triangles.*

### 5.1.2. Windowed Collision Search

For each window in the image, a local maximum optical flow has to be found. This local maximum is found by determining the maximum optical flow from all vertices inside the window and all vertices that are within the global maximum optical flow of the edges of the window. This results in a bounding box size for each axis direction: $\{B_u, B_d, B_l, B_r\}$, where each maximum optical flow direction decides the bounding box size in the opposite direction. For each pixel inside the window, all vertices and edges within the bounding box distances around this pixel are tested for collisions.

First, each vertex in the bounding box is checked using the method described in Section 4.2.2. This is implemented as shown in Alg. 1. For each vertex, the distance to the current pixel $\vec{P}$ is compared to the optical flow magnitude of that vertex. If the optical flow magnitude is smaller than the distance, this pixel can never reach this pixel and it is ignored. A small tolerance is used here, since it is better to check pixels that are just on the boundary, since a more detailed evaluation later will determine if a collision actually occurs. If this vertex could hit the pixel, the next part checks if it actually does. In case the vertex starts at this pixel, the collision is automatically registered at $t = 0$. Otherwise, the collision distance $\lambda$ is calculated using Eq. (4.17). If this distance is smaller than a set threshold, $\lambda <$ tolerance, and the collision occurs during the interpolation time-frame, $0 \leq t \leq 1$, a new collision should be added. The $\Delta_{\text{leaving}}$ and $\Delta_{\text{entering}}$ are found by querying the position of the vertex just before and just after the collision.

Next, this process is repeated for all edges in the bounding box. If the vertices of the edge cannot reach the current pixel, the edge is ignored. Otherwise, the time $t$ and distance along the edge $\lambda$ are calculated using Eq. (4.5). If they are both within the range of $0$ to $1$, a collision occurs. For this collision, the triangle that the pixel was occupying before the collision and the triangle that the pixel is now occupying after the collision are calculated using Eq. (4.10).

**Algorithm 1:** Vertex Collisions

**for** (*Vertex* $V$ *in Bounding Box*) **{**
  **if** ($\left\|V.\vec{flow}\right\| \geq \left\|\vec{V} - \vec{P}\right\| -$ *tolerance*) **{**
    `// Find` $\lambda$ `and` $t$`:`
    **if** ($\vec{V} = \vec{P}$) **{**
      $\lambda = 0$
      $t = 0$
    **} else {**
      $\vec{X} = \vec{P} - \vec{V}$
      $\lambda = \frac{\vec{X}.yV.\vec{flow}.x - \vec{X}.xV.\vec{flow}.y}{\left\|V.\vec{flow}\right\|}$
      $t = \frac{\vec{X}.xV.\vec{flow}.x + \vec{X}.yV.\vec{flow}.y}{\left\|V.\vec{flow}\right\|^2}$
    **}**
    **if** ($\lambda <$ *tolerance and* $0 \leq t \leq 1$) **{**
      `//` $V$ `hits` $P$
      Find $\Delta_{\text{leaving}}$ and/or $\Delta_{\text{entering}}$
      $C \leftarrow \{\Delta_{\text{leaving}}, \Delta_{\text{entering}}, t, \lambda\}$
      Append $C$ to $P$.*collisions*
    **}**
  **}**
**}**

**Algorithm 2:** Edge Collisions

**for** (*Window in Frame*) **{**
  **for** (*Pixel* $P$ *in Window*) **{**
    **for** (*Edge* $\epsilon$ *in Bounding Box*) **{**
      **if** ($\left(\left\|\epsilon.V_0.\vec{flow}\right\| \geq \left\|\epsilon.\vec{V_0} - \vec{P}\right\| -$ *tolerance*$\right)$ *or* $\left(\left\|\epsilon.V_1.\vec{flow}\right\| \geq \left\|\epsilon.\vec{V_1} - \vec{P}\right\| -$ *tolerance*$\right)$) **{**
        Find $\lambda_0/\lambda_1$ and $t_0/t_1$
        **if** (*tolerance* $< \lambda < 1 -$ *tolerance and* $0 \leq t \leq 1$) **{**
          `//` $\epsilon$ `hits` $P$
          Find $\Delta_{\text{leaving}}$ and/or $\Delta_{\text{entering}}$
          $C \leftarrow \{\Delta_{\text{leaving}}, \Delta_{\text{entering}}, t, \lambda\}$
          Append $C$ to $P$.*collisions*
        **}**
      **}**
    **}**
  **}**
**}**

### 5.1.3. Chaining of Collisions

The collisions found from the vertex and edge collisions are sorted by time. The chaining algorithm, shown in Alg. 3 and visualised in Fig. 5.4, has to determine which collisions contribute to the $I/t$ curve and which collisions get occluded by others. It starts with the first collision, which is usually from the vertex that starts on the current pixel at $t = 0$. This collision is seen as the current active collision. Then, the next collision is considered, for which there are three options:

1. This new collision is a continuation of the current collision: The triangle that the current collision entered is the same as the one that the new collision leaves. This collision becomes the new active collision.
2. This new collision is not a continuation, but it is closer than the current collision, so this collision becomes the new active collision and a new chain is followed.
3. This new collision is not a continuation and it is further away, so this collision is ignored.

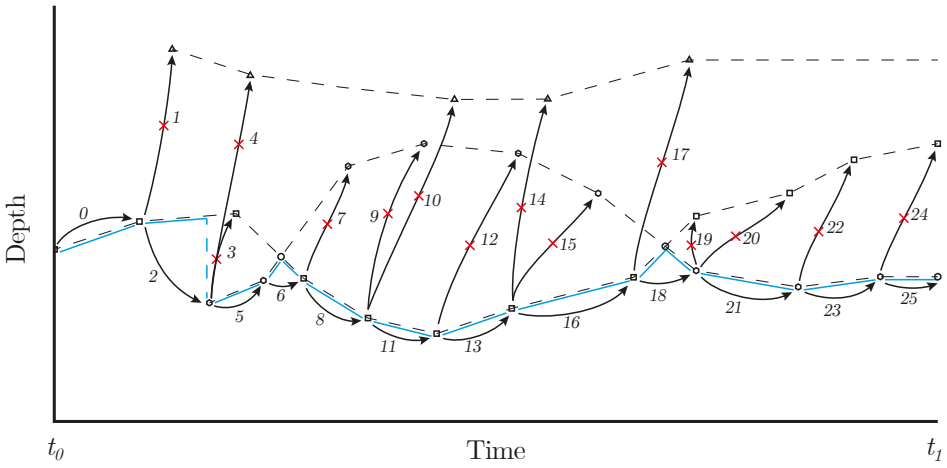
This process is repeated for every collision that was found for this pixel. This results in a time-sorted list of control points, each with a corresponding intensity value $I$, which together describe the $I/t$ curve.

**Algorithm 3:** Chaining Collisions into Control Points

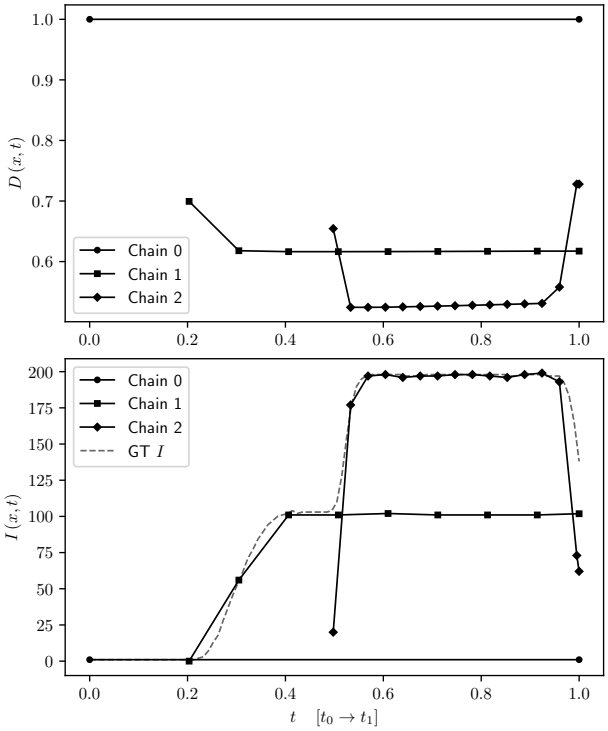
```
for (Pixel P in Frame) {
    Append P.collisions[0] to P.control points as first Control Point
    C_latest ← P.collisions[0]
    for (C in P.collisions) {
        if (C.Δ_leaving = C_latest.Δ_entering) {
            // The next collision is connected to this collision
            Append C to P.control points
            C_latest ← C
        else if (C.z(C.t) ≤ C_latest.z(C.t)) {
            // The next collision is closer than the current collision at the
               collision time
            Append C to P.control points
            C_latest ← C
        else {
            // The next collision is skipped
            // In this case, the C_latest remains the same
        }
    }
}
```

### 5.1.4. Merging of Forward and Backward Passes

This process is performed for both the forward pass; starting from $I_0$ and $z_0$ and moving with $\vec{F}_{0\to1}$, and the backward pass; starting from $I_1$ and $z_1$ and moving with $\vec{F}_{1\to0}$. In this case, the parameter $k$ is set at $3$. The backward pass is reversed in time and the two passes are merged into one final result using the triangle area metric from Section 4.2.4.




**(a)** *Collisions are chained based on the connected triangles into the final control points (cyan). The numbers indicate the evaluation order. At each collision, the algorithm is evaluated for the next collision in time and is either added (connected with arrow), or discarded (crossed out arrow).*




**(b)** *Example of generated chains in a low resolution image. The top graph shows the depth of each chain, while the bottom shows the intensity of each chain and the ground truth.*

**Figure 5.4:** *Illustration of the chaining algorithm.*

## 5.2. Implementation of Event Generation in Unreal Engine 5

For use with Unreal Engine 5, a plug-in version[3] of the algorithm is implemented. Next to our novel FastESIM method developed in this thesis, the original ESIM method [3] of generating events, through the difference of HFR frames, is also implemented. The original implementation used a slower method to retrieve frames from Unreal Engine, while the new implementation is more tightly integrated with the rendering engine. Additionally, both versions are implemented so they can be easily compared. This version works directly inside Unreal Engine and can be enabled in any existing project as a plug-in. The settings and parameters of the plug-in can be set from another program, such as a python script, with an API. The resulting event data can be streamed back through this API. The plug-in controls the movement of the camera through the scene based upon velocity values that are given using the API. More details about the interface with the plug-in can be found in Appendix C.3.

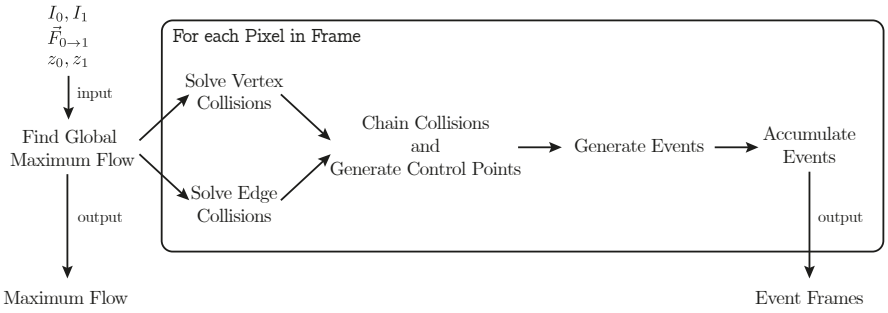



**Figure 5.5:** *The pipeline implemented in Unreal Engine. Currently, it is slightly simplified compared to the full method.*

In Unreal, the rendering process[4] occurs through shaders that execute in parallel on the GPU. These shaders are ordered in a RDG for each frame. Unreal is highly multi-threaded. The main tasks are distributed over two main threads: the Game Thread and the Rendering Thread. The Game Thread is responsible for updating, or 'ticking', all objects for each frame. Here, processes like movement of the camera or other objects, physics, or other tasks related to the simulated world are processed. The Rendering Thread on the other hand, is responsible for turning the simulated world into frames on the screen. These threads communicate extensively to make sure that the scene stays synchronised between the two threads. One of the main tasks of the Rendering Thread is running the Graph Builder, which assembles the RDG for each frame to make sure that all shaders are executed in the right order and they have all resources that they need. Additional shaders can be added to the RDG by a Scene View Extension (SVE). This is a class in the Unreal Engine, called upon by the Rendering Thread when assembling the RDG, that is made for adding post processing effects to the rendering process. A new custom SVE is used to govern the event rendering process. This SVE is responsible for asking the graph builder for the right textures and enqueueing the right shaders into the RDG.

Like the different inputs that the stand-alone version uses, different render passes are available for use in the shaders. From these passes, the plug-in uses the *Scene Color* pass, the *Scene Depth* pass, and the *Velocity* pass. The *Scene Color* pass provides the rendered colour of the current scene as four floating point values, one for each channel: Red, Green, Blue, and Alpha[5]. These values are provided in a High Dynamic Range (HDR) format, scaled to the range of $0.0$ to $1.0$, where $0.0$ is absolute darkness and $1.0$ is the edge of white clipping. However, since these values are stored as floating point values, intensities higher than one are possible: this means that the camera is clipping here and Unreal would normally limit to white in a later stage and apply a bloom effect[6] from these pixels. For the purpose of simulating event-based sensors, this HDR format is advantageous, since these sensors also have a large dynamic range. If events are generated from LDR sources, many accuracy issues can occur. The *Scene Depth* pass provides the depth from the camera in the direction of each pixel in as a floating point value. Finally,

[3] Full code is available on github.com/ReinierZwikker/FastEventGeneratorPlugin

[4] See Section 2.3 for more information about the rendering process.

[5] Alpha channel is often used for transparency.

[6] Bloom is an effect in rendering where a very bright light source causes a glow on the neighbouring region.

the *Velocity* pass contains the two dimensional optical flow vectors from the previous frame to the current frame.

For a correct simulation, it is very important to keep track of time. Unreal Engine attempts to render new frames as quickly as possible, which causes the wall clock time between frames to vary, dependent on frame complexity and external factors. ESIM [3] requires frames to be rendered with a maximum optical flow magnitude of 0.5 px between each frame. Ensuring this optical flow magnitude is not exceeded requires fine control of the instant in time when each new frame is rendered. The simulation time is therefore completely separated from the wall clock time at which each frame is rendered. Based on the current optical flow magnitude, the current delta time between frames is adjusted. The linear and rotational velocity of the camera is used to update the position of the camera with the right amount each frame. All other objects in the Unreal scene are slowed down using the Global Time Dilation feature in Unreal Engine, which allows precise control of the delta time for these objects. This results in motion in the scene that is coherently slowed down to the right speed, so that each frame is rendered from exactly the right spot. The new FastESIM method also limits the motion in the scene in the same way, but not to the same degree.

The simulator can be configured to output event frames or raw events. The event frames are sent to the API. The event frames are two dimensional frames, sent in floating point format, where each pixel in the frame contains the amount of positive and negative events fired during the accumulation time separately. The accumulation time of the event frames can be configured, which allows for fast accumulation within Unreal, without sending unnecessary data to other processes. The raw events are accumulated for each rendered frame and sent using the API.

### 5.2.1. ESIM in Unreal Engine

The new Unreal version of ESIM [3] is implemented as a Compute Shader. This shader is executed for every pixel of the screen. A reference value is stored for each pixel in a Persistent Texture. Each frame, every pixel determines the difference between the current intensity value and the reference value in logarithmic space. This difference is divided by the event threshold to determine the amount of events that should be fired, which is written to the event frame. When the optical flow is smaller than 0.5 px, there should be a maximum of one event in a single pixel, while many pixels will not fire events at all in most frames.

The maximum optical flow magnitude in the frame is determined with a sequence of shaders, where each pixel in a quarter of the frame in the sequence stores the maximum of 4 pixels in the image, dividing the size of the frame by 4 each time, until one maximum remains in the end. This value is copied from the GPU to the CPU and transferred to the game thread, in order to determine the speed of the simulation. When the accumulation time window of an event frame is larger than the time that a single frame can render, the event images are accumulated on the GPU and exported each time the event frame delta time has elapsed.

### 5.2.2. FastESIM using Grid-Based Frame Interpolation

In Unreal Engine, our novel event generation method works in a similar way to the implementation in C++. The method is also implemented as a Compute Shader, running in parallel on the GPU for each pixel. While some implementation details differ due to the differing platforms, the core is implemented using the same algorithms as the C++ version. However, due to the more restricted nature of the shader language in Unreal Engine, some simplifications are made.

In Unreal, the size of the memory resources requested from the Graph Builder needs to be determined at graph assembly time, before any shader is executed. This means that, since the velocity pass has not yet been rendered, it is not possible to know the amount of control points or events that will be generated for each pixel. This is why in the Unreal version, each pixel can output a fixed amount of control points and events. If a pixel exceeds this amount, these events will be leaked and thus discarded. At the moment, the Unreal version uses one fixed bounding box size for checking collisions, sized to the maximum optical flow of the entire frame, based on the maximum optical flow magnitude, that is calculated in a similar fashion as with the ESIM method, but now also separately determines the maximum in each direction. The delta time between each frame is determined by the maximum supported optical flow or the requested delta time between event frames.

# 6 Results

In this chapter, our new method FastESIM will be compared against the previous ESIM method and ground truth. For the developed method, both a qualitative analysis on the generated events and event frames, and a quantitative analysis on the error between the ground truth are performed. Additionally, the performance of the Unreal Engine plug-in is analysed. More results can be found in Appendix A.

## 6.1. Qualitative Analysis of Grid-Based Frame Interpolation

The implemented method is tested in various ways. Example scenes in Blender are used to evaluate the quality of the generated $I/t$ curves by comparing against high frame-rate ground truth data, also rendered from Blender. Additionally, the Sintel Dataset [34] is used to test scenes with diverse movement. This dataset contains selected scenes from a short film produced by the Blender Foundation, that has open sourced the scene files. These scenes have been rendered with depth and optical flow data for benchmarking purpose.

Fig. 6.1 shows the frame interpolation of such an example scene, in which the camera moves over a checkerboard pattern. In this scene, there is are no occlusions or accretions, other than the edge of the frame. This makes the interpolation very simple, all pixels move in the same direction and no overlapping regions have to be solved. In regions that are only visible in one frame, border artifacts appear, because the pass that does not have information for these regions is given too much weight in the merging of the forward and backward pass. The $I/t$ curve of a pixel in this frame is shown in Fig. 6.2: The checkerboard pattern moving past this pixel is clearly visible in this curve.

Next, Fig. 6.3 contains a more complicated example from the Sintel Dataset. This scene contains the a large optical flow difference between the moving character and the almost stationary background. Parts of the background that are obscured by the character in one frame, such as the region behind the arms, are successfully filled in from the other frame. In the cropped view, it can be seen that the movement of the foot is interpolated correctly, as it occludes and is occluded by the pavement correctly when it moves down.

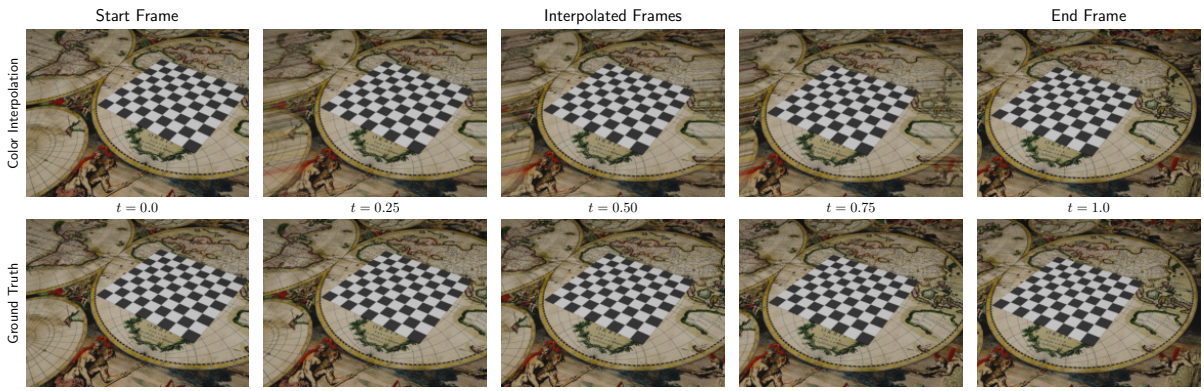



**Figure 6.1:** *Interpolation of a test scene. Interpolation generated by the C++ version.*

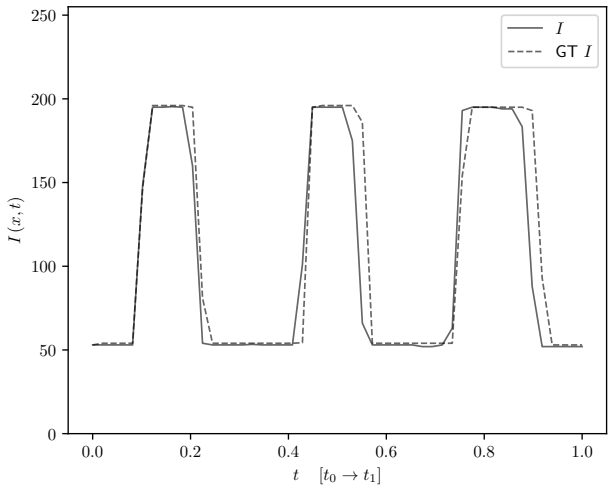



**Figure 6.2:** *$I/t$ curve of a pixel from the test scene. The movement of the checkerboard pattern can be seen in the intensity of the pixel.*

Fig. 6.4 shows an example from the Sintel Dataset that contains many overlapping regions: In the foreground, the running character is holding a spear, which is correctly rendered above the dragon that moves behind them. The camera is slowly moving to the right, which causes the background to move to the left. Using the depth information, the method successfully separates these regions and moves each individually in the right direction. Because the triangle grid is used, objects are correctly occluded. The forward and backward merging allows for gaps in either frame to be filled. The accumulated events frames in a slices of one hundredth of the entire frame motion are also visualised. In these event frames, it can be seen that the movement is correctly modelled and events are generated at the correct time and place between the frames.

In Fig. 6.5, a scene with a quickly rotating helicopter rotor is shown. This scene has also been rendered at a very high time resolution, to compare the interpolated result to the ground truth. In the cropped view, the effect of the linear optical flow model can be seen, where the interpolated rotor moves linearly between key-frames, instead of rotating correctly. This results in interpolation errors in scenes with large non-linear motion.

More examples can be found in Figs. A.1 to A.3 in Appendix A.




**Figure 6.3:** *Color interpolation of a frame in the Alley 2 scene from the Sintel Dataset. Interpolation generated by the C++ version.*

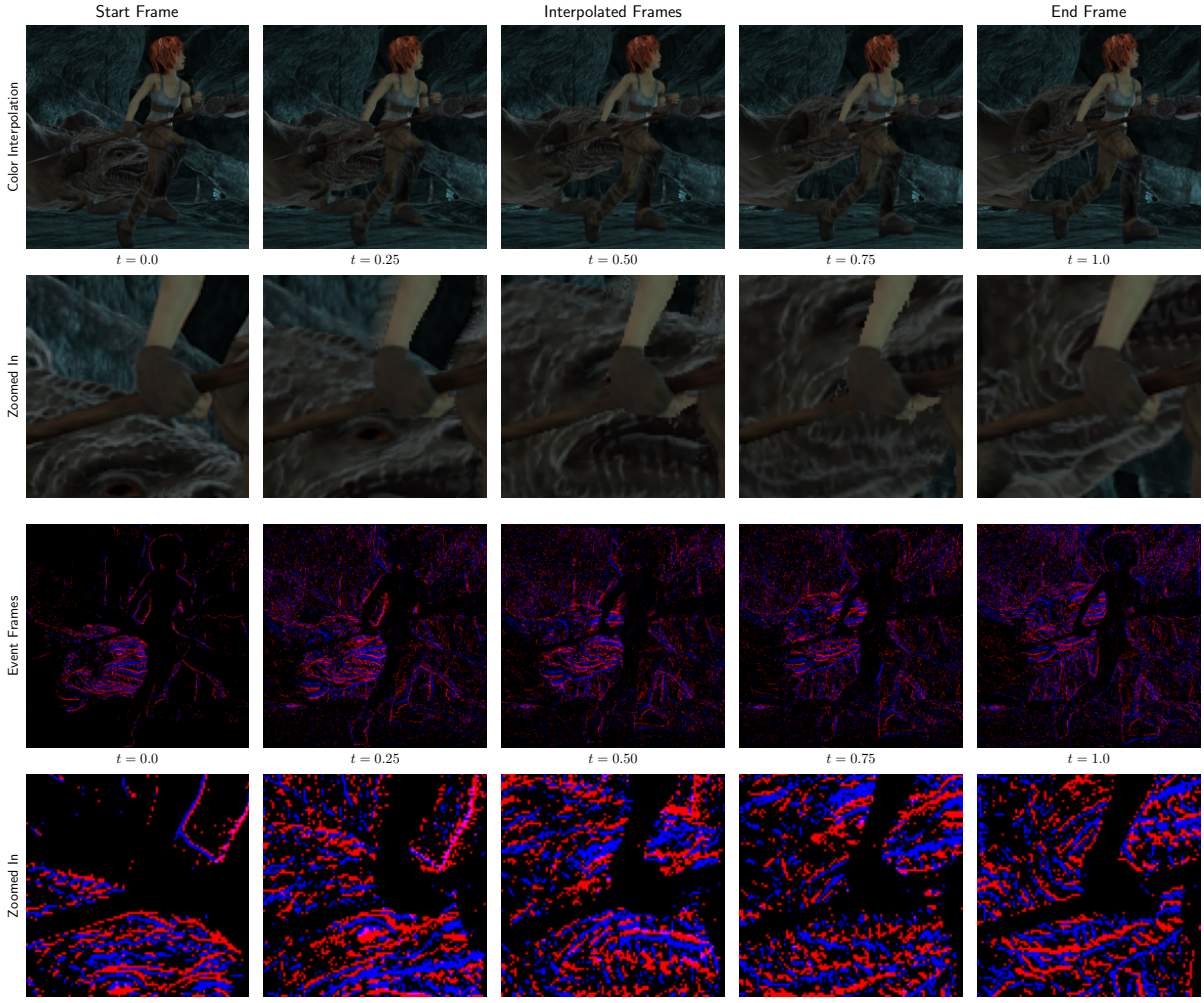



**Figure 6.4:** *Color interpolation and event generation of a frame in the Cave 2 scene from the Sintel Dataset. Interpolation and events generated by the C++ version.*

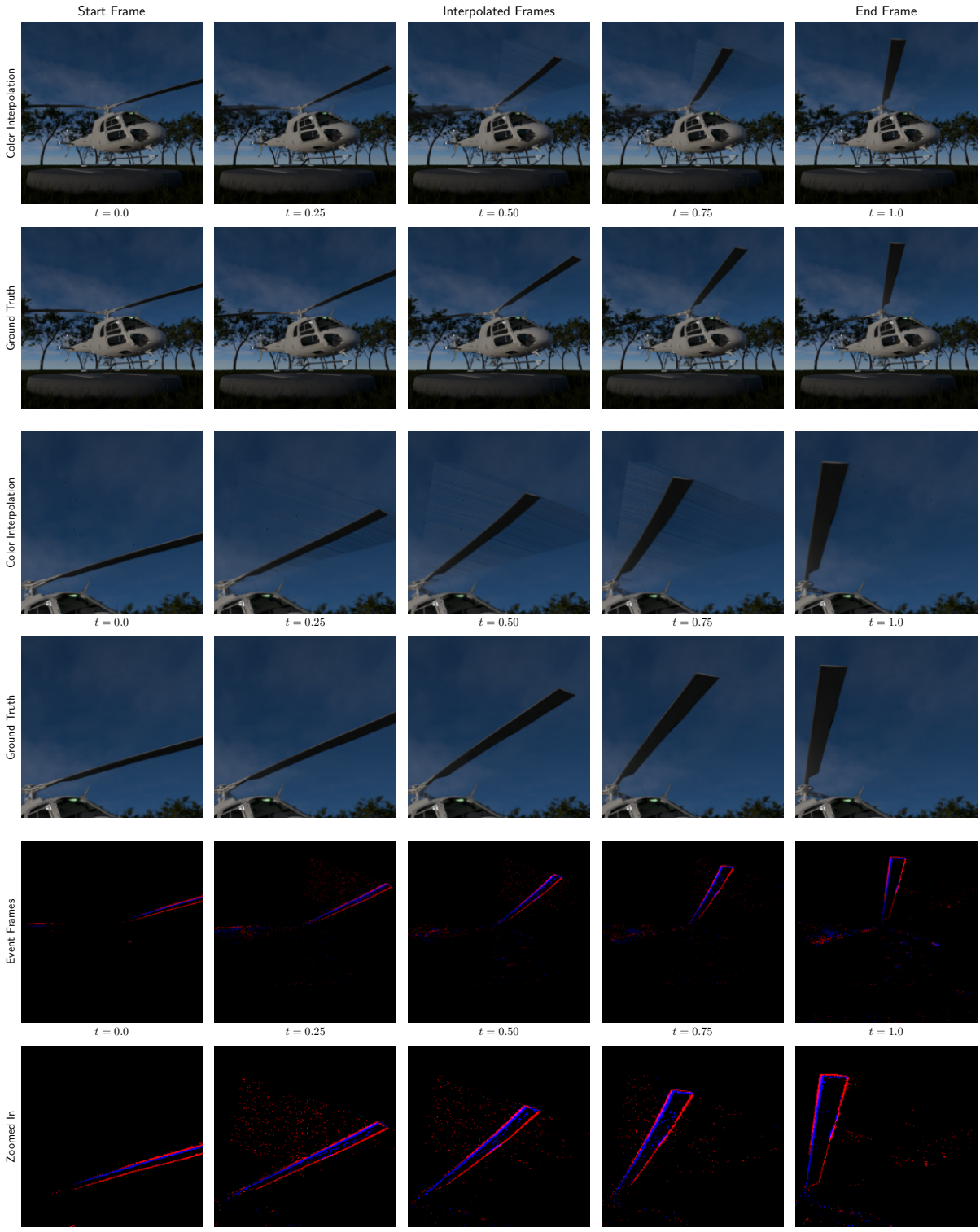



**Figure 6.5:** *Color interpolation and event generation of a helicopter with moving rotor, compared to rendered high frame-rate ground truth. The middle two rows show a cropped view of the rotor blade, where artifacts appear due to the linear optical flow model. Interpolation and events generated by the C++ version.*

## 6.2. Quantitative Analysis of Grid-Based Frame Interpolation

The performance of the new method is quantified against ESIM [3] in two different scenarios. The ESIM method is defined as a linear interpolation between two key-frames, chosen at a set optical flow distance from each other. For a camera with a fixed velocity, this is approximated with a fixed key-frame difference. Important for the comparison between the methods is that this key-frame interval is the same between the methods, not that the optical flow is never higher than a set value, because this analysis compares the quality of the results as a function of the amount of samples needed, and thus of the computational burden of the method. In the first scenario, the camera is rotated around a panoramic image, projected onto a sphere of equal depth everywhere. This means that there are no occlusions. In the second scenario, the camera is translated and rotated through a photorealistic room rendered in Blender. In this case, occlusions do occur. In both cases, the RMSE between the interpolated frames and the ground truth is determined for multiple sample intervals.

### 6.2.1. Panoramic Case

In the panoramic case, the camera was rotated around the yaw axis for $360$ degrees with steps of constant angular distance, varied between $0.9$ degrees and $9.0$ degrees. This results in optical flow distances from 10 px to 100 px. In the ESIM method, the colour is linearly interpolated between each sample. In Fig. 6.6, an example can be seen for one frame pair in the sequence that is spaced $9.0$ degrees apart. In this example, the ghosting artifacts caused by linear interpolation between frames with too much optical flow can be seen in the ESIM row, while our method is still able to model the intensity between the frames. These samples are compared against the ground truth render from the panoramic image. The mean RMSE is obtained from the full rotation, seen in Fig. 6.7 for this sequence. It can be seen that both methods line-up with the ground truth at each sampled frame, but deviate while interpolating. The maximum RMSE occurs in the middle between two frames.

This data is used to compare the mean RMSE of ESIM and FastESIM in Fig. 6.9a. The full data can be found in Table A.1 in Appendix A. For the same amount of sampled frames, the mean RMSE of our method is much lower, outperforming the original ESIM method. Since the computational complexity of our method is much lower than the that of rendering a frame, it is beneficial to render less frames. In order for ESIM to achieve the same RMSE, many more frames would have to be sampled: In this case, the scene would have to be sampled $8000$ times to achieve a maximum optical flow of 0.5 px recommended by the ESIM method.

Yet, there is still a difference between the FastESIM method and the ground truth, which is caused by two sources: the error due to the interpolation itself and the error due to the optical flow model. The optical flow is represented linearly, with a constant velocity over its entire length. However, as is illustrated in Fig. 6.8, the real optical flow due to the rotation of the camera is not linear and does not have a constant velocity. The last row shows the result of ideal sampling, with the true, non-linear, optical flow. The difference between our FastESIM method and this result shows the error due to linear optical flow and the remaining error is the fundamental interpolation error.

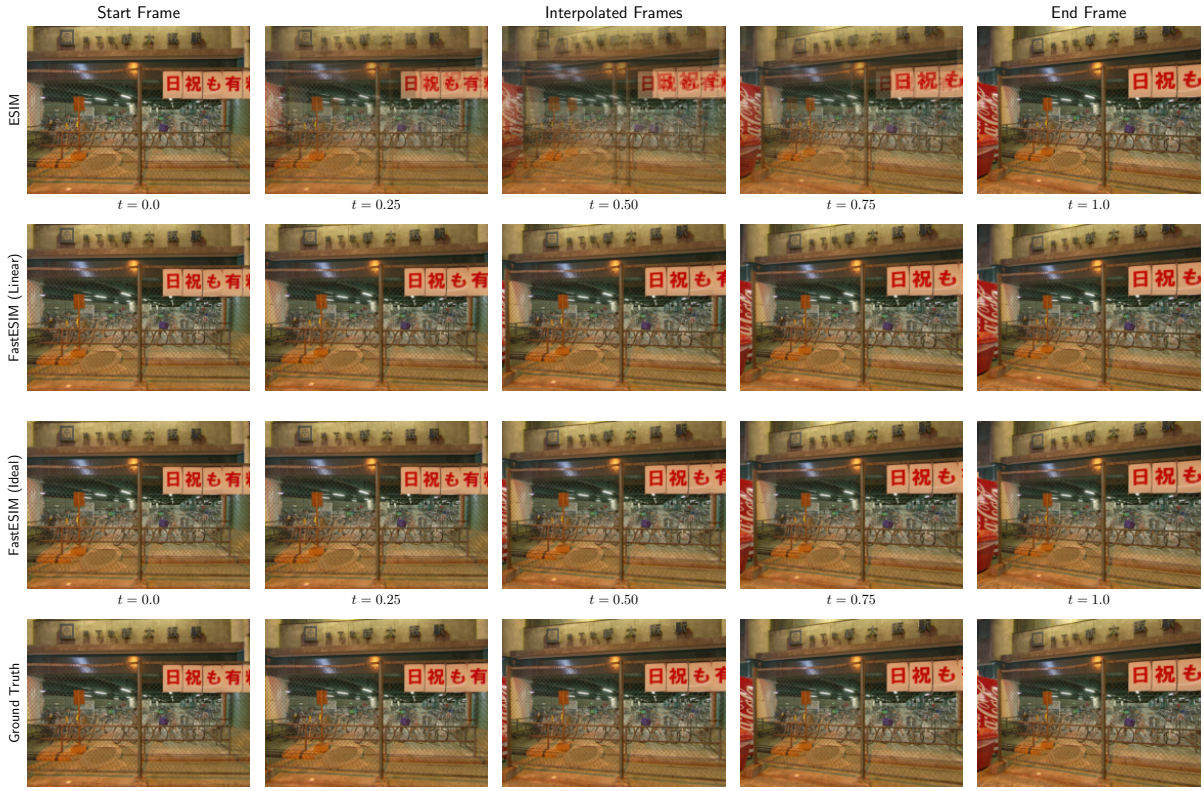



**Figure 6.6:** *Interpolated frames used for the panoramic analysis. In this example, the sampled frames are* $9$ *degrees apart. Interpolation generated by the C++ version.*

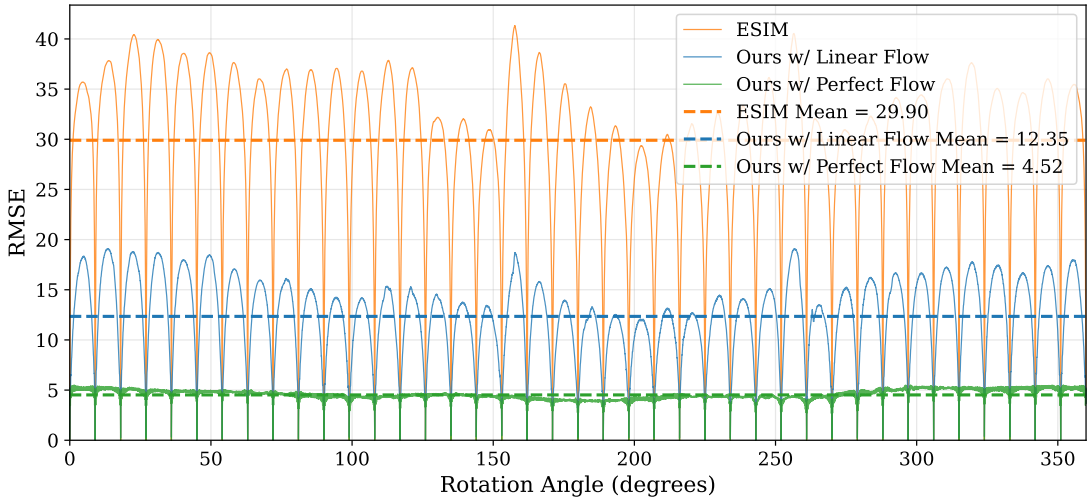



**Figure 6.7:** *Example of RMSE over the rotation in the panoramic sequence for the sequence with frames spaced* $9$ *degrees apart. Each frame, the interpolation methods are equal to the ground truth, in-between both diverge. When there is more detail in the current frame, the RMSE of the interpolation is higher. The RMSE is shown here in the range* $0..255$.

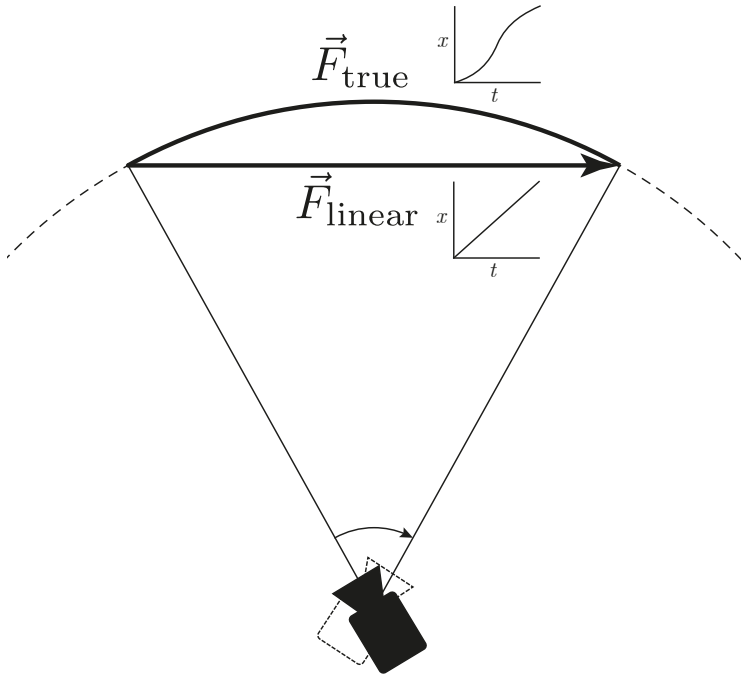
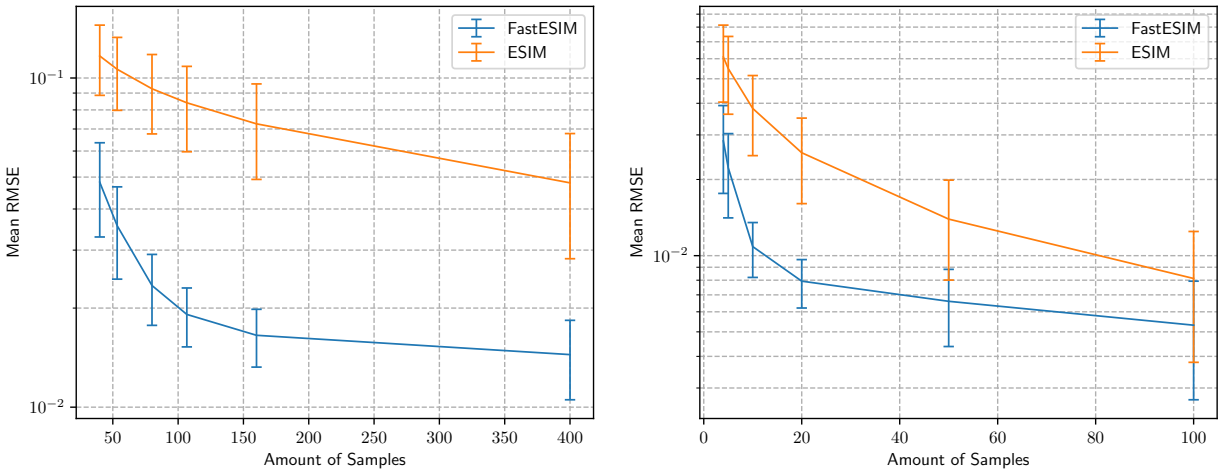



**Figure 6.8:** *The difference between the true and linear optical flow in the panoramic case. When projected to the screen, the linear flow has a constant speed in screen coordinates. The projection of the true optical flow will move slower at the edges and faster in the middle, due to motion perpendicular to the screen.*




**(a)** *Panoramic Comparison* **(b)** *Realistic Room Scene Comparison*

**Figure 6.9:** *The comparison of mean RMSE between ESIM and FastESIM for the Panoramic and Realistic Room sequence. The RMSE is normalised to the range* $0..1$.

### 6.2.2. Realistic Room Case

Next, a more complicated scene is tested. In this scene the camera translates and rotates through a square room filled with furniture and other details. Other than the panoramic image projected onto a sphere, this scene is fully 3D, with depth and changes of perspective. A view into this room can be seen in Fig. 6.10. A trajectory through this room was rendered with a high frame-rate, such that the motion between each frame is very small. Then, six key-frame interval settings are selected. As the key-frame interval increases, the optical flow between each key-frame is increased. Just like in the panoramic case, for each setting, the time between the key-frames is interpolated with the ESIM method and the new FastESIM method. Each frame in between the key-frames is regenerated and compared with the ground truth frames. Additionally, events are generated using the same trajectory, which are analysed against ground truth events.



**Figure 6.10:** *A view of the Realistic Room scene, rendered in Blender.*

**Interpolation Analysis** In Fig. 6.11, an example can be seen for a key-frame pair that is spaced $125$ frames apart. The top row shows the linear interpolation of the ESIM method. It can be seen that linear interpolation is not suitable for such a large optical flow magnitude, and a much lower magnitude should be selected when using this method. In the next row, it can be seen that our FastESIM method is able to interpolate this large magnitude correctly: It retains the structure of the scene and does not show any duplicates. It also correctly handles occlusions. Due to the linear optical flow representation, the interpolated frame does not always line up with the ground truth, which is the main cause of error in the ground truth comparison. In Fig. 6.12 the error per frame can be seen for three key-frame interval settings. The behaviour is similar to the panoramic case, both methods diverge in-between frames, but with different magnitudes. The maximum absolute error in the frame is dependent on the content and complexity of the current frame.

This results in the comparison seen in Fig. 6.9b. The full data can be seen in Table A.2 in Appendix A. This data shows the same result as for the panoramic case: For the same amount of rendered frames, the quality of our FastESIM method is better, which means that this method provides a better result for the same computational budget. When increasing the rendering budget, at some point to the right of the plotted region, the original ESIM method will become as accurate as our method.

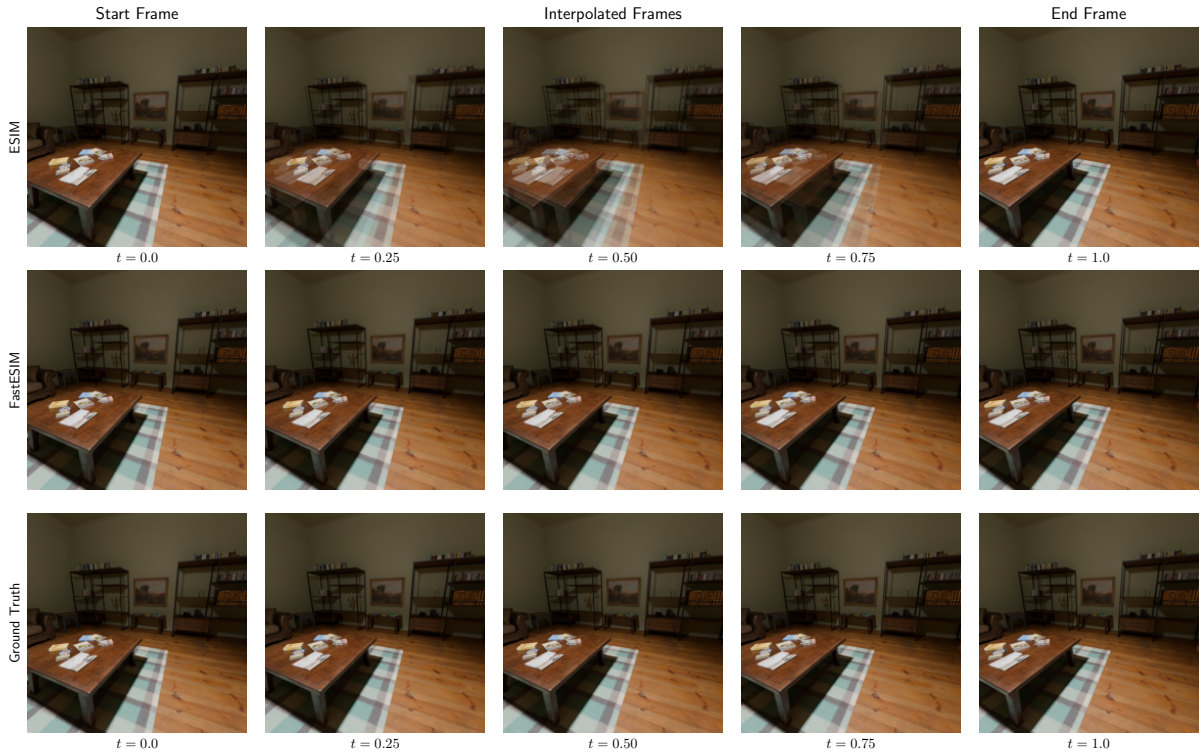
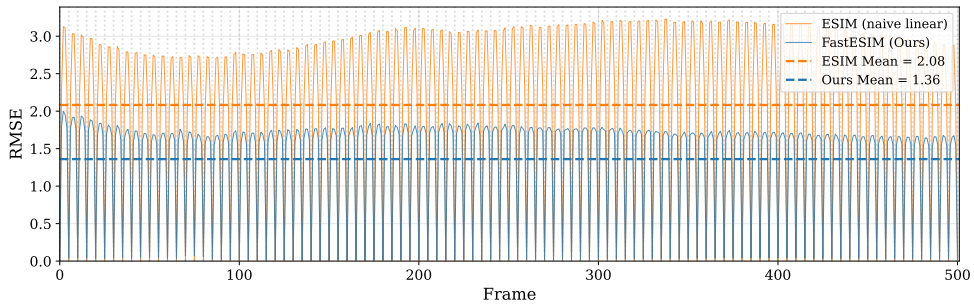
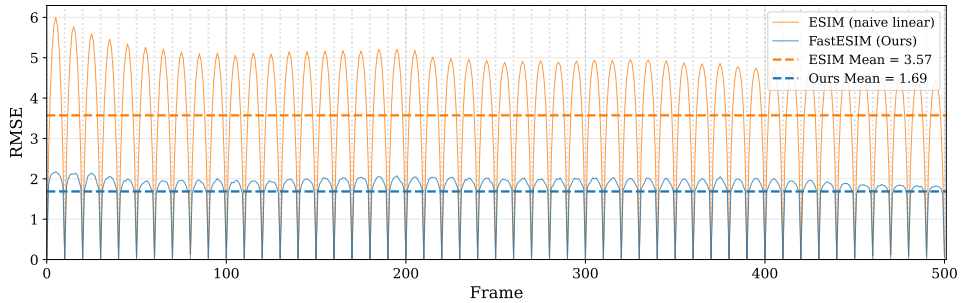



**Figure 6.11:** *Interpolated frames used for the Realistic Room analysis. In this example, a key-frame spacing of* $125$ *frames is shown. Interpolation generated by the C++ version.*

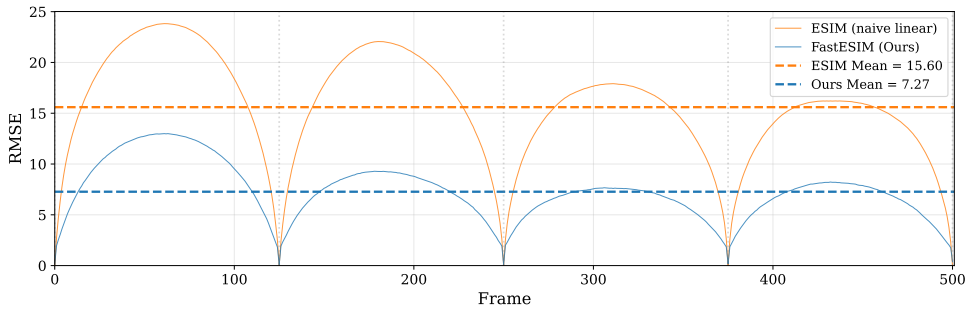



**(a)** *RMSE with key-frame interval of* $5$ *frames.*




**(b)** *RMSE with key-frame interval of* $10$ *frames.*

**Figure 6.12:** *Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of* $500$ *frames. The RMSE is shown here in the range* $0..255$.




**(c)** *RMSE with key-frame interval of* $125$ *frames.*

**Figure 6.12:** *Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of* $500$ *frames. The RMSE is shown here in the range* $0..255$*.*

**Event Analysis** For the realistic room case, events have been generated using the interpolated intensity curve from all six key-frame interval settings. This curve was interpolated both with linear interpolation (ESIM) and with our FastESIM method. Each frame of the rendered sequence was set to be spaced 2 ms apart and the generated events are accumulated into event frames with a 10 ms width. The events are generated with a symmetric contrast threshold of $0.1$ and $0.05$ on the normalised signal. In Fig. 6.13, the event frames for the different key-frame interval settings can be compared. It can be seen that the linearly interpolated method generates the correct events when the key-frame interval is small, but quickly deteriorates when this interval is increased. For large key-frame intervals, the output is completely wrong and the events are very unrealistic. On the other hand, our FastESIM method maintains the correct structure of the scene, even with large key-frame intervals. While the events in these cases still look realistic, the error with the ground truth does increase, because, due to the discrepancy between the linear optical flow model and the true optical flow, the position of the interpolated structures in the scene is not correct. So, while our method correctly assembles the scene at each point in time, it does not always do so in the correct place.

In Fig. 6.14, this result can also be seen. The full data can be found in Tables A.3 to A.6 in Appendix A. Fig. 6.14a and Fig. 6.14b show the RMSE of the event frames compared to the ground truth, for each contrast threshold. From this data, it is visible that the ESIM method deteriorates as the key-frame interval increases, in agreement with the analysis of the event frames. The quality of the FastESIM method also decreases as the amount of samples is reduced, with a clear drop-off visible, when the true non-linear trajectory becomes too different from the linear optical flow used. In the RMSE data, it can be seen that, while the event frames for large key-frame intervals still look realistic, they are not correct when compared to the ground truth.

Because the event frames contain mostly inactive pixels, an analysis with sparse metrics is also performed. In Fig. 6.14c and Fig. 6.14d, the F1 metric for both methods and thresholds can be seen. This metric is the geometric mean between the precision, how many of the events in the ground truth also appear in the interpolation, and the recall, how many of the events in the interpolation also appear in the ground truth. This value, a ratio between $0$ and $1$, shows the quality of the predicted events, where a value of $1$ would be a perfect prediction. The results show that our FastESIM method can predict event with much greater accuracy for the same rendering budget.

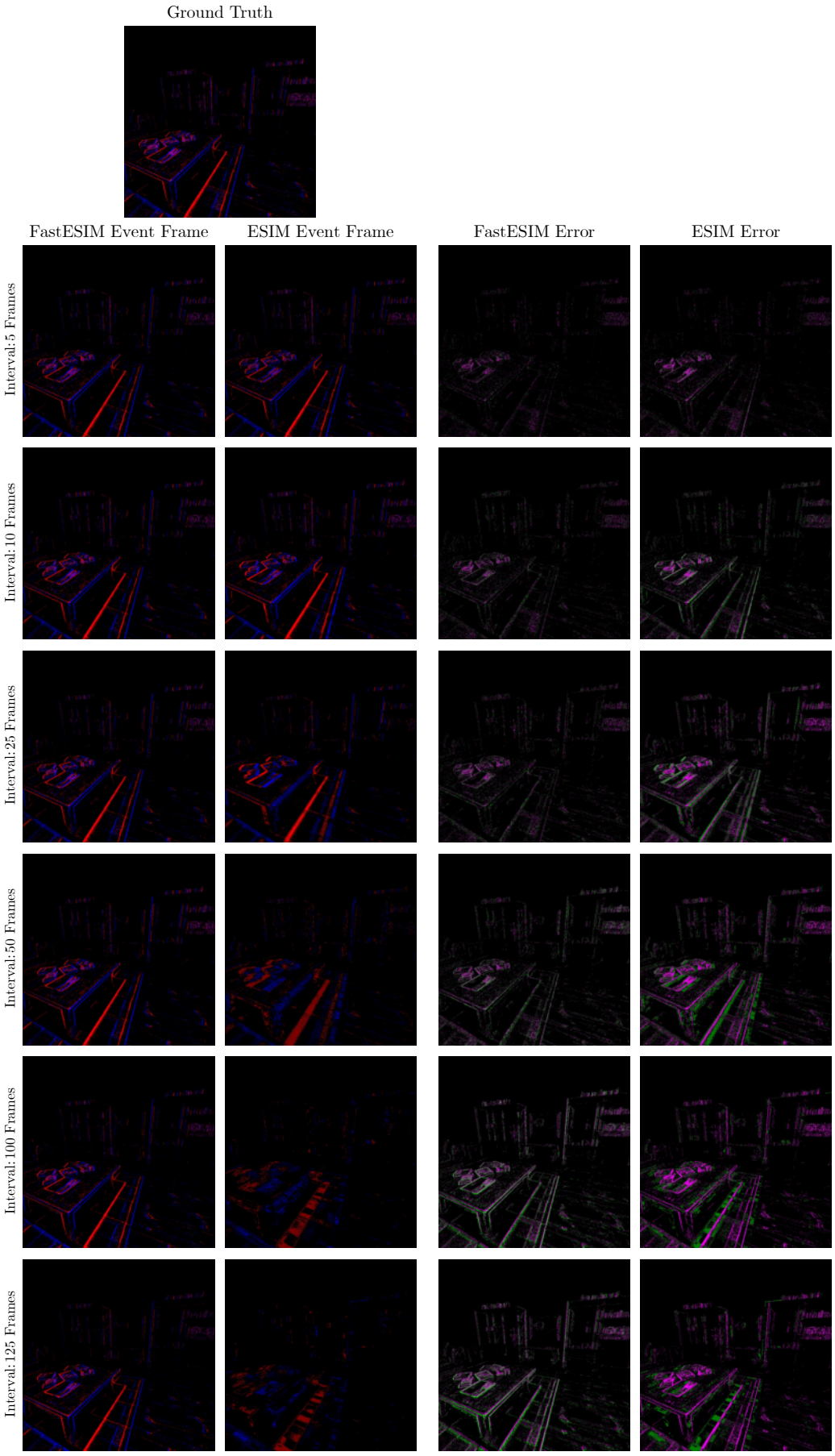
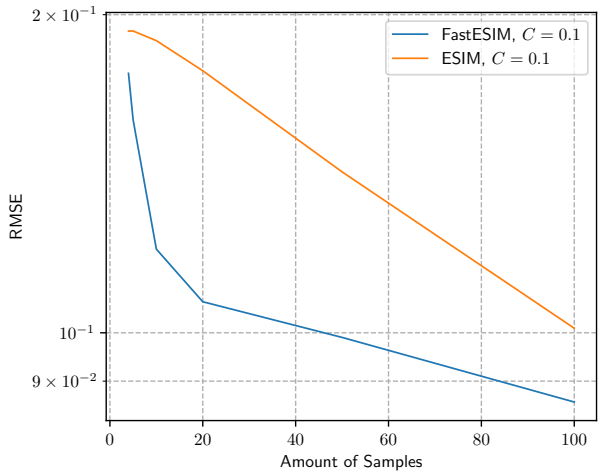
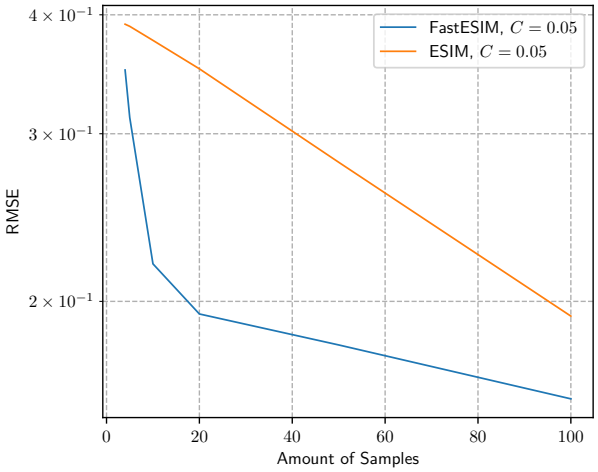


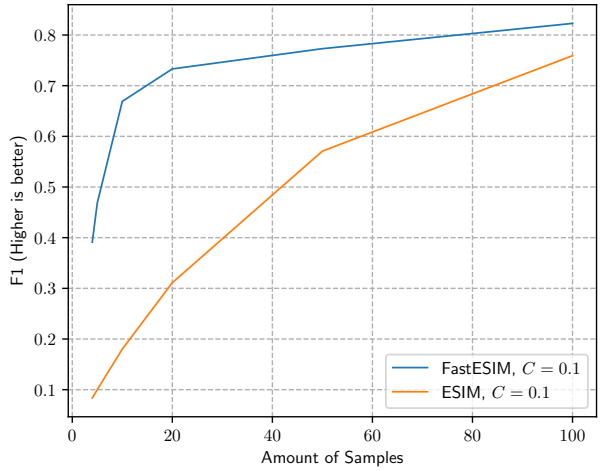

**Figure 6.13:** *The event frames of the Realistic Room scene, generated by ESIM and FastESIM with a contrast threshold of* $0.05$ *and accumulation time of* 10 ms*. In the two left columns, the event frames can be seen, while the error of both methods is shown in the right columns.*



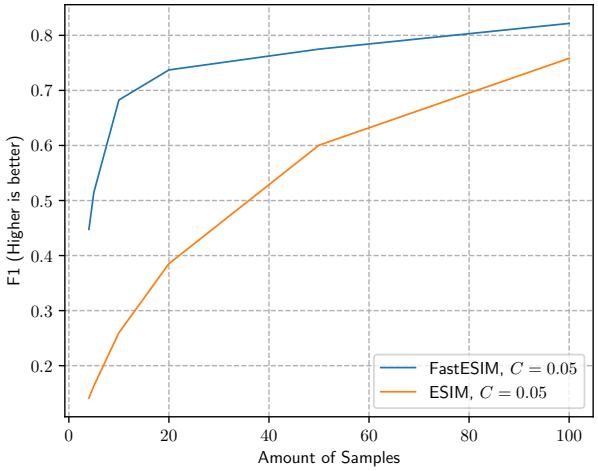

**(a)** *The RMSE of the Realistic Room with* $C = 0.1$.




**(b)** *The RMSE of the Realistic Room with* $C = 0.05$.




**(c)** *The F1 of the Realistic Room with* $C = 0.1$.




**(d)** *The F1 of the Realistic Room with* $C = 0.05$.

**Figure 6.14:** *Event Frame metrics for the Realistic Room scene for two contrast thresholds:* $C = 0.1$ *and* $C = 0.05$*. The top row shows the mean RMSE of the whole event frames compared to the ground truth, while the bottom row uses a sparse-aware metric to quantify the quality of the prediction. The F1 metric is the geometric mean of the precision and recall of the prediction. The precision quantifies how many of the ground truth events are predicted and the recall quantifies how many of the predicted events were correct. A higher F1 value is better.*

## 6.3. Performance of Unreal Engine Plug-in

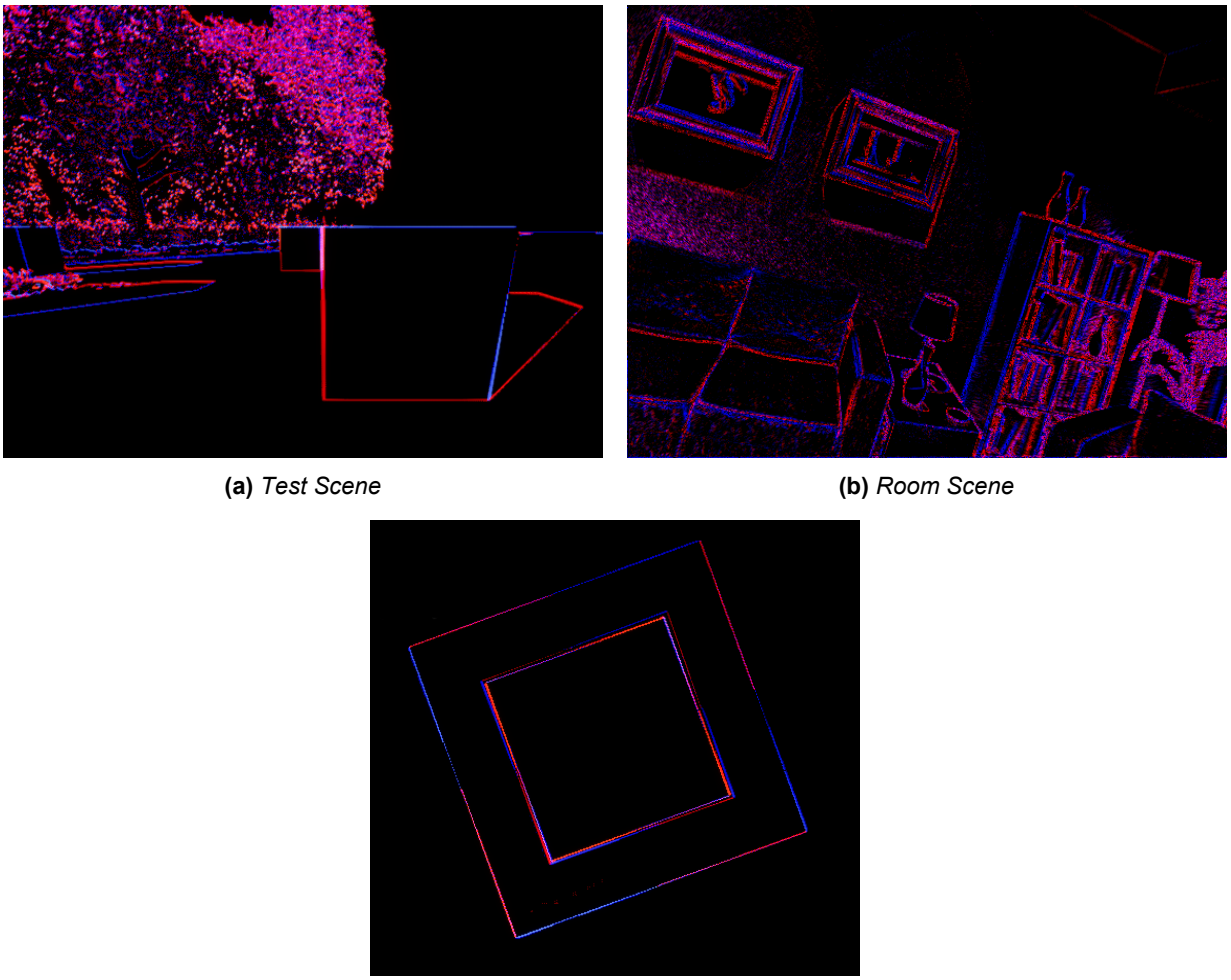
The developed plug-in for Unreal Engine 5.6 has two modes: ESIM mode and FastESIM mode. ESIM mode re-implements the ESIM algorithm [3] directly in Unreal Engine. Where the original implementation of ESIM was limited to rendering frames from Unreal at 20 fps, this new implementation uses a tight integration with the Unreal rendering pipeline, to render around 80 fps, on the same hardware. With more performance tweaks, this could be increased further. Additionally, the original implementation used the LDR image from Unreal, while this new version uses the HDR version before tone-mapping. This means that more detail is available for generating events in low and high light situations. In order to generate realistic events, ESIM needs to limit the optical flow of the scene to 0.5 pixels, which means that the motion needs to be slowed down. Depending on the velocity of the camera and the depth of the scene, the real-time factor adapts accordingly. For a camera moving at 1 m/s towards a wall at 2 m away, the real-time factor is 0.005. For the original implementation of ESIM, this would be approximately 0.00125.

The second mode uses the new FastESIM method to convert events. Fig. 6.15 shows three event frames rendered with the new method. The first scene is a simple test scene, that contains cubes and moving trees. The second is an existing scene of a photorealistic room. This scene shows that this method can handle complex geometry and lightning. The last scene consist of the camera moving towards a gate, while rolling. This scene will be used in further research to evaluate the use of the plug-in for training behaviour on event camera input. The new method can render realistic events with much higher optical flow magnitudes between frames, which allows this method to convert much more scene time into events with each rendered frame. This results in a much higher conversion factor for FastESIM. The current implementation can support optical flow magnitudes of up to 20 pixels, which results in a real-time factor of 0.5 for the same camera moving at 1 m/s towards a wall at 2 m. In this case, the factor is still dependent on the velocity of the camera and the depth of the scene, since the maximum flow is still regulated.

**Table 6.1:** *Render FPS and conversion factor comparison between the different simulators. The conversion factor is defined as the time unit of events that is generated per time unit required to render those events.*

| | Original ESIM | ESIM in Unreal | FastESIM |
|---|---|---|---|
| Maximum Optical Flow (px) | 0.5 | 0.5 | 20 |
| Render FPS | 20 | 80 | 50 |
| Theoretical Speed-up from ESIM | - | 4× | 100× |
| Measured Real-time factor[1] | 0.00125 | 0.005 | 0.5 |
| Measured Speed-up from ESIM | - | 4× | 400× |

[1] For a camera moving at 1 m/s towards a wall at 2 m.



**(a)** *Test Scene*

**(b)** *Room Scene*

**(c)** *Moving towards gate*

**Figure 6.15:** *Event Frames generated using the FastESIM Unreal Engine plug-in.*

# Part III

## Closure

# 7

# Conclusions

In order to fulfil the research objective, five research questions were posed. The answers to the first three research questions guide the development of the goal of a fast event simulator and the last two will allow for the evaluation of this simulator.

**Research Question 1**

How can modelled pixel attributes be represented between frames?

The first research question concerns the format of the model of the pixel attributes between frames. These attributes include mainly intensity and colour of pixels, but can also include other properties. In this new method, these properties are linearly interpolated between pixels and thus also modelled linearly between collisions. This results in a list of control points for each pixel, with each control point defining the current value at a certain time. Using these control points, a piecewise linear function can be defined that can describe any attribute between two frames. Since these control points are only defined for the exact times that a interpolated pixel switches from one triangle to the next, they are only stored when the slope of the function can change, not at a fixed interval, which is the case with existing frame interpolation techniques. Additionally, this is determined individually for every pixel, such that more active regions generate more data, without storing unnecessary information for inactive regions.

**Research Question 2**

How can continuous-time pixel attributes be modelled from low frequency frames and data generated by a rendering engine?

The results show that the developed method can model the pixel attributes between given frames accurately. It makes efficient use of the additional data that is available in rendering engines, such as the optical flow and depth passes. In order to continuously model the pixel attributes, the backward optical flow functions $\vec{F}'_{t\to 0,1}(t)$ need to be solved for the entire time domain from $t_0$ to $t_1$. By displacing the triangle grid, at any $t$, this method can provide a mapping between the current location and the original location in the source frame: which is the backward flow. By solving this problem in a continuous way, a continuous model of the pixel attributes is found.

**Research Question 3**

How can continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be computed in (near) real-time?

The analysis in Chapter 6 shows that the new method can generate more accurate frames and events compared to previous methods for the same rendering budget. The implementation as an Unreal Engine Plug-in, where the method is implemented as a Compute Shader to run on the GPU, shows that the method can be used to compute event frames much faster than previous event simulation techniques. Because the new method can take much bigger steps between frames, the required rendering budget needed to

generate realistic events is much lower. The new method can generate events much faster, however, it does not yet reach real-time rates in a typical scene. Additionally, the quality of generated events with the current implementation in Unreal Engine is worse than the previous method. This is not a result of the new method, which is capable of generating events at the same quality achieved by previous methods, but with the implementation in Unreal Engine. The full potential of the method is not yet reached, because only the interpolation pass forwards in time is implemented. Additionally, the amount of control points is fixed to 35 control points per pixel in this implementation, which limits the total optical flow that this implementation can handle to 20 pixels. This method has the potential to model pixel attributes continuously at real-time rates.

**Research Question 4**

Can the continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be used to upsample the time resolution of video accurately?

From the analysis of the interpolated frames, it can be seen that the developed method can reconstruct frames with a higher precision given the same rendering budget. Challenging regions in the frames, such as regions of occlusion and accretion, are handled correctly. Due to the linear optical flow model used in this method, movement in the scene that is very non-linear causes errors in the interpolation.

**Research Question 5**

Can the continuous-time pixel attributes modelled from low frequency frames and data generated by a rendering engine be used to synthesise events accurately?

The continuous model of the intensity over time allows for direct generation of events, without having to sample many frames. The analysis shows that the event frames accumulated from these events are accurate, and require much less rendered frames compared to previous methods. This shows that accurate events can be generated using the pixel intensity modelled with the developed method.

**Research Objective**

In this thesis, a faster, real-time, method of simulating Event Cameras will be developed.

The developed method allows for the interpolation of pixel attributes in continuous-time, based on the optical flow and depth information provided by the rendering engine. This allows for VFI and event generation without any learned stages in the pipeline, which allows this method to be independent of training datasets. Compared to other frame interpolation methods, which invoke the method for every frame that is generated between two frames, this method can sample as many frames as are requested after generating the continuous model of pixel attributes between two frames once. However, typical problems from linear VFI techniques also appear in this method, such as artifacts in objects that are not visible in either frame, problems with large motion and varying lighting conditions. Additionally, the current method uses linear optical flow, which introduces errors when the movement in the scene is not linear.

The implemented method is shown to be $400\times$ faster than existing methods of generating events. While not achieving real-time rates in standard scenes, the event generation real-time factor of $0.5\times$, where previous methods generated events with a factor of $0.00125\times$, allows this method to be used in controller-in-the-loop training and experimentation with event cameras in simulation. This implementation shows many areas that can be improved, in order to achieve real-time event generation using this method.

The method developed in this thesis shows that it is possible to generate events from low frequency frames that are spaced apart with a significant optical flow magnitude, by interpolating based on the information that rendering engines use to generate frames. By modelling the intensity of each pixel in-between frames continuously, instead of sampled at a high frequency, a more efficient way of generating events is achieved.

# 8

# Recommendations

During the development of the new grid-based interpolation and event simulator method, various topics came to light that are not yet answered. Some of these topics were planned features that are not yet implemented and others are improvements to the core method that can be implemented in future research. This chapter contains these recommendations.

**Improved Merging of Forward and Backward Passes** The current scheme that merges the passes forward and backward in time is not optimal, resulting in artifacts due to missing information in one pass and stretched boundaries. Information that is missing in one pass should be overwritten by the detail in the other, but due to the factor of the weight that determines that a interpolation is more valuable when closer to its origin frame, these triangles are given too much priority in the merging. This often happens in regions that overstretch. In the triangle grid, the triangles in this region become very large. When triangles become too small and their area becomes negative, they are ignored. Similarly, when these triangles become too big, they should also be ignored. One way of finding these triangles, is by reversing the optical flow. In this case, these triangles that would become large, now have a negative area. These triangles can be invalidated in the actual frame at the time that their area becomes negative.

**Unreal Engine Flow Recalculation** During the development of the Unreal plug-in, it was found that the optical flow generated by Unreal Engine is not always correct. Future research should investigate how Unreal calculates the optical flow pass and change the interpretation in the plug-in, or should change the plug-in to recalculate the optical flow from the depth pass and the motion of the camera. Improving the optical flow input to the FastESIM method will improve the quality of the events generated in Unreal.

**Improvement of Unreal Engine Plug-in** In the current version of the Unreal plug-in, only the forward pass is implemented. Future research should also implement the reversed time pass using the reverse time flow. This flow is not available in Unreal Engine, and thus has to be calculated. Then, the forward and backward merging strategy should be implemented in Unreal. The C++ version calculates the window size for multiple sections in the frame, such that regions with large flow have a large search window, but the search window of other sections can remain small. This should also be implemented in Unreal Engine. The amount of control points per pixel should be dynamically allocated, such that higher maximum optical flow values can be supported by the Unreal plug-in. Finally, when frames with optical flow and thus with larger delta time can be converted, splitting one converted frame into multiple event frames should be implemented, as this allows for larger steps in time.

The current performance metrics are generated using the in-editor Play function of Unreal Engine. By compiling the program to a stand-alone game, with less debug functionality enabled, an even higher performance gain could be achieved.

**Implementation in CUDA** During this thesis, it has been chosen to implement the method on the GPU as a plug-in to Unreal Engine. This allows for integration in training pipelines, using Unreal Engine as the rendering engine. For offline event generation, in future research, a stand-alone CUDA version should also be implemented. This version should be able to generate control points and events from input frames on the GPU, allowing for fast event generation from existing images, or integration into other rendering engines and pipelines.

**Validation of the Generated Events** Future research should show that training on events generated with this method works and helps when transferring to a real-life scenario. The effectiveness of this training should be compared to training with ESIM, taking the difference of rendering time into account.

**Investigate effect of collision window size** The window size has been fixed at $128 \times 128$ in this work. Future work should investigate the effect of different window sizes on the performance of the method.

**Grid and Control Point Decimation** Large sections of solid colours are represented in our method with the same grid size as very textured areas. These solid areas produce the same amount of collisions and require the same amount of processing, while not providing any more information. Multiple triangles of the same color could be combined into larger triangles without loss of precision. This would result in a grid with fewer edges that have to be checked for collisions, which would results in shorter processing time.

Similarly, the control points that describe the $I/t$ curve as a piecewise linear line use the same amount of control points, based on the amount of collisions, even when the resulting $I/t$ is straight. In order to reduce the amount of control points stored, they can be simplified by only adding new control points when the derivative is significantly different and moving the last existing control point otherwise, or using other simplification techniques, such as the Ramer–Douglas–Peucker algorithm [35].

**Robustness to Non-Linear Optical Flow** The currently implemented method uses a linear model for the optical flow between two frames. However, the motion between frames is often non-linear, due to accelerations, decelerations, rotations, or directional changes. An example of non-linear flow due to rotation can be seen in Fig. 6.5 in Chapter 6, where the interpolated trajectory of the rotor blades differs significantly from their true trajectory. In order to solve this, a non-linear optical flow model needs to be generated and used throughout the method. This non-linear flow could be implemented as piecewise linear segments or Bézier curves. If piecewise linear segments are used, the same collision solutions can be used on the individual sections, but if a higher order model of the optical flow is used, the collisions also need to be solved in a higher order that linear.

Rendering engines do not natively provide this non-linear optical flow, as it is not used in the standard rendering pipeline. However, it is still possible to be obtained from information present in the rendering engine. From either the continuous or quickly sampled camera and object positions in the scene, a non-linear optical flow model can be determined.

**Add Event Noise Patterns to the New Simulator** Various other works since the original ESIM event simulator [3] have focussed on the realism of the generated events. Real-life event cameras have many noise patterns and effects driven by the physics of the event-based sensor. These noise effects can have a big impact of the performance of trained methods that use event vision and their portability from the simulator to real-life. This work does not consider these noise effects when simulating events, instead focussing on rendering idealised events as fast as possible. The inclusion of these noise effects can improve the realism of the generated events.

**Native Event Rendering Engine** The next step is to not generate frames at all, but directly calculate continuous $I/t$ curves using the rendering equation[1] from the scene information. In this method, a grid of triangles is generated from rendered pixels that is projected from the camera using depth values. A traditional frame based rendering engine also uses triangles made of vertices to render individual frames. The developed method can also be used in this way. Instead of moving pre-rendered pixels, the triangles from the objects in the scene are used to generate the collisions with the pixels and rendered at these collisions. While a scene would contain many of these triangles, they would on average be much bigger than the triangles generated from the pixel grid, which could result in a significant speed-up.

In order for this native event rendering engine to be useful, textured triangles need to be implemented. In the current method, each vertex can have a solid color, and the colours on a triangle are interpolated from its corners. Instead, this rendering engine has to implemented textured triangles, from which the path that the pixel travels can be sampled to generate correct events. This would result in a rasterising rendering engine that can render continuous intensity information and generate events directly from a scene.

[1] Eq. (2.1), See Section 2.3.

# References


[1] Bharatesh Chakravarthi et al. “Recent Event Camera Innovations: A Survey”. In: (Aug. 2024). URL: http://arxiv.org/abs/2408.13627.

[2] Tomas Akenine-Möller et al. *Real-Time Rendering Fourth Edition*. Fourth Edition. CRC Press Taylor & Francis Group, 2018.

[3] Henri Rebecq et al. “ESIM: an Open Event Camera Simulator”. In: *Proceedings of Machine Learning Research*. Vol. 87. 2018.

[4] Berthold K.P. Horn et al. “Determining optical flow”. In: *Artificial Intelligence* 17.1-3 (Aug. 1981), pp. 185–203. DOI: 10.1016/0004-3702(81)90024-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0004370281900242.

[5] Michael A. Gennert et al. “Relaxing the Brightness Constancy Assumption in Computing Optical Flow”. In: (June 1987). URL: https://dspace.mit.edu/handle/1721.1/6477.

[6] Dahyeon Kye et al. “AceVFI: A Comprehensive Survey of Advances in Video Frame Interpolation”. In: (June 2025). URL: https://arxiv.org/pdf/2506.01061.

[7] Simon Niklaus et al. “Softmax Splatting for Video Frame Interpolation”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Mar. 2020), pp. 5436–5445. DOI: 10.1109/CVPR42600.2020.00548. URL: https://arxiv.org/pdf/2003.05534.

[8] Huaizu Jiang et al. “Super SloMo: High Quality Estimation of Multiple Intermediate Frames for Video Interpolation”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2018. DOI: 10.1109/CVPR.2018.00938.

[9] Matt Pharr et al. *Physically Based Rendering: From Theory to Implementation*. Mar. 2023. URL: https://pbr-book.org/.

[10] Pradeep Gupta. *CUDA Refresher: Reviewing the Origins of GPU Computing | NVIDIA Technical Blog*. Apr. 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/.

[11] Guillermo Gallego et al. “Event-Based Vision: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (2022). DOI: 10.1109/TPAMI.2020.3008413.

[12] Henri Rebecq et al. “Real-time Visual-Inertial Odometry for Event Cameras using Keyframe-based Nonlinear Optimization”. In: (Sept. 2017). URL: https://rpg.ifi.uzh.ch/docs/BMVC17_Rebecq.pdf.

[13] Alex Zihao Zhu et al. “EV-FlowNet: Self-Supervised Optical Flow Estimation for Event-based Cameras”. In: *Robotics: Science and Systems* (Aug. 2018). DOI: 10.15607/RSS.2018.XIV.062. URL: http://arxiv.org/abs/1802.06898%20http://dx.doi.org/10.15607/RSS.2018.XIV.062.

[14] Alex Zihao Zhu et al. “Unsupervised Event-based Learning of Optical Flow, Depth, and Egomotion”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2019-June (Dec. 2018), pp. 989–997. DOI: 10.1109/CVPR.2019.00108. URL: https://arxiv.org/pdf/1812.08156.

[15] Amos Sironi et al. “HATS: Histograms of Averaged Time Surfaces for Robust Event-based Object Classification”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Mar. 2018), pp. 1731–1740. DOI: 10.1109/CVPR.2018.00186. URL: https://arxiv.org/pdf/1803.07913.

[16] R Wes Baldwin et al. “Inceptive Event Time-Surfaces for Object Classification Using Neuromorphic Cameras”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial*

*Intelligence and Lecture Notes in Bioinformatics)* 11663 LNCS (Feb. 2020), pp. 395–403. DOI: 10.1007/978-3-030-27272-2{_}35. URL: http://arxiv.org/abs/2002.11656%20http://dx.doi.org/10.1007/978-3-030-27272-2_35.

[17] Ryad Benosman et al. “Event-based visual flow”. In: *IEEE Transactions on Neural Networks and Learning Systems* 25.2 (Feb. 2014), pp. 407–417. DOI: 10.1109/TNNLS.2013.2273537. URL: https://www.researchgate.net/publication/258355234_Event-Based_Visual_Flow.

[18] R Wes Baldwin et al. “Time-Ordered Recent Event (TORE) Volumes for Event Cameras”. In: (2022). DOI: 10.1109/TPAMI.2022.3172212. URL: www.ieee.org/publications/rights/index.html.

[19] Daniel Gehrig et al. “End-to-End Learning of Representations for Asynchronous Event-Based Data”. In: *Proceedings of the IEEE International Conference on Computer Vision* (Apr. 2019), pp. 5632–5642. DOI: 10.1109/ICCV.2019.00573. URL: https://arxiv.org/pdf/1904.08245.

[20] Waseem Shariff et al. “Event Cameras in Automotive Sensing: A Review”. In: *IEEE Access* 12 (2024), pp. 51275–51306. DOI: 10.1109/ACCESS.2024.3386032. URL: https://ieeexplore.ieee.org/abstract/document/10494342.

[21] Xu Zheng et al. “Deep Learning for Event-based Vision: A Comprehensive Survey and Benchmarks”. In: 14.8 (Feb. 2023). URL: https://arxiv.org/pdf/2302.08890.

[22] Jesse Hagenaars et al. “Self-Supervised Learning of Event-Based Optical Flow with Spiking Neural Networks”. In: (Oct. 2021). URL: http://arxiv.org/abs/2106.01862.

[23] Elias Mueggler et al. “The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and SLAM”. In: *The International Journal of Robotics Research* 36.2 (Feb. 2017), pp. 142–149. DOI: 10.1177/0278364917691115. URL: https://journals.sagepub.com/doi/10.1177/0278364917691115.

[24] Daniel Gehrig et al. “Video to Events: Recycling Video Datasets for Event Cameras”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Dec. 2019), pp. 3583–3592. DOI: 10.1109/CVPR42600.2020.00364. URL: https://arxiv.org/pdf/1912.03095.

[25] Yuhuang Hu et al. *v2e: From Video Frames to Realistic DVS Events*. Tech. rep. URL: https://github.com/SensorsINI/v2e.

[26] Damien Joubert et al. “Event Camera Simulator Improvements via Characterized Parameters”. In: *Frontiers in Neuroscience* 15 (2021). DOI: 10.3389/fnins.2021.702765.

[27] Alexander Amini et al. “VISTA 2.0: An Open, Data-driven Simulator for Multimodal Sensing and Policy Learning for Autonomous Vehicles”. In: *2022 International Conference on Robotics and Automation (ICRA)* (Nov. 2021), pp. 2419–2426. URL: http://arxiv.org/abs/2111.12083.

[28] Songnan Lin et al. “DVS-Voltmeter: Stochastic Process-Based Event Simulator for Dynamic Vision Sensors”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13667 LNCS. 2022, pp. 578–593. DOI: 10.1007/978-3-031-20071-7{_}34. URL: https://link.springer.com/10.1007/978-3-031-20071-7_34.

[29] Zhongyang Zhang et al. “V2CE: Video to Continuous Events Simulator”. In: *Proceedings - IEEE International Conference on Robotics and Automation* (Sept. 2023), pp. 12455–12461. DOI: 10.1109/ICRA57147.2024.10609864. URL: https://arxiv.org/pdf/2309.08891.

[30] CARLA. *CARLA: DVS Sensor*. 2023. URL: https://carla.readthedocs.%20io/en/latest/ref_sensors/#dvs-camera.

[31] PROPHESEE. *PROPHESEE: Video to Event Simulator*. 2023. URL: https://docs.prophesee.ai/stable/samples/modules/core_ml/viz_video_to_event_simulator.html.

[32] Anish Bhattacharya et al. “EvDNeRF: Reconstructing Event Data with Dynamic Neural Radiance Fields”. In: *Proceedings - 2024 IEEE Winter Conference on Applications of Computer Vision, WACV*

*2024* (Oct. 2023), pp. 5834–5843. DOI: 10.1109/WACV57701.2024.00574. URL: https://arxiv.org/pdf/2310.02437.

[33] Joachim Ott et al. “Text-to-Events: Synthetic Event Camera Streams from Conditional Text Input”. In: (June 2024). URL: https://arxiv.org/pdf/2406.03439.

[34] D. J. Butler et al. “A naturalistic open source movie for optical flow evaluation”. In: *European Conf. on Computer Vision (ECCV)*. Ed. by A. Fitzgibbon et al. (Eds.) Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, pp. 611–625.

[35] Wenzhong Shi et al. “Performance evaluation of line simplification algorithms for vector generalization”. In: *Cartographic Journal* 43.1 (Mar. 2006), pp. 27–44. DOI: 10.1179/000870406X93490; PAGE:STRING:ARTICLE/CHAPTER. URL: https://www.tandfonline.com/doi/pdf/10.1179/000870406X93490.

# Appendix

# A

# Additional Results

In this chapter of the appendix, additional results are shown. The data that is plotted in Chapter 6 is shown here in Tables A.1 to A.6. Additional interpolated frames and events are shown in Figs. A.1 to A.6.

**Table A.1:** *Panoramic comparison between ESIM and FastESIM for varying optical flow. The RMSE is normalised to the range* $0..1$.

| Angle step | Optical flow | Amount of | ESIM | | FastESIM | |
|---|---|---|---|---|---|---|
| (deg) | (px) | samples | Mean RMSE | (Std) | Mean RMSE | (Std) |
| 0.900 | 10 | 400.0 | 0.0480 | (0.0198) | 0.0144 | (0.0039) |
| 2.250 | 25 | 160.0 | 0.0726 | (0.0234) | 0.0165 | (0.0033) |
| 3.375 | 37 | 106.7 | 0.0841 | (0.0244) | 0.0191 | (0.0039) |
| 4.500 | 50 | 80.0 | 0.0927 | (0.0251) | 0.0234 | (0.0057) |
| 6.750 | 75 | 53.3 | 0.1063 | (0.0265) | 0.0356 | (0.0111) |
| 9.000 | 100 | 40.0 | 0.1167 | (0.0281) | 0.0482 | (0.0153) |

**Table A.2:** *Realistic Room scene comparison between ESIM and FastESIM for varying optical flow. The RMSE is normalised to the range* $0..1$.

| Key-frame | Amount of | ESIM | | FastESIM | |
|---|---|---|---|---|---|
| Interval | frames | Mean RMSE | (Std) | Mean RMSE | (Std) |
| 5 | 100 | 0.0081 | (0.0043) | 0.0053 | (0.0026) |
| 10 | 50 | 0.0139 | (0.0059) | 0.0066 | (0.0022) |
| 25 | 20 | 0.0255 | (0.0095) | 0.0079 | (0.0017) |
| 50 | 10 | 0.0382 | (0.0133) | 0.0109 | (0.0027) |
| 100 | 5 | 0.0548 | (0.0186) | 0.0222 | (0.0081) |
| 125 | 4 | 0.0609 | (0.0205) | 0.0284 | (0.0108) |

**Table A.3:** *Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.1$ on the range $0..1$. All metrics show the same order.*

| Method | Key-frame Interval | RMSE (counts) | RRMSE (% mean) | NRMSE (% range) | MAE (counts) |
|---|---|---|---|---|---|
| FastESIM | 5 | 0.086 | 398.9% | 1.4% | 0.007 |
| FastESIM | 10 | 0.099 | 454.6% | 1.6% | 0.010 |
| ESIM | 5 | 0.101 | 468.7% | 1.7% | 0.010 |
| FastESIM | 25 | 0.107 | 491.6% | 1.8% | 0.011 |
| FastESIM | 50 | 0.120 | 548.8% | 2.0% | 0.014 |
| ESIM | 10 | 0.142 | 653.5% | 2.4% | 0.017 |
| FastESIM | 100 | 0.159 | 727.7% | 2.6% | 0.023 |
| FastESIM | 125 | 0.176 | 797.7% | 2.9% | 0.026 |
| ESIM | 25 | 0.177 | 811.2% | 2.9% | 0.026 |
| ESIM | 50 | 0.189 | 863.1% | 3.1% | 0.028 |
| ESIM | 100 | 0.193 | 878.9% | 3.2% | 0.029 |
| ESIM | 125 | 0.193 | 879.3% | 3.2% | 0.029 |

**Table A.4:** *Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.05$ on the range $0..1$. All metrics show the same order.*

| Method | Key-frame Interval | RMSE (counts) | RRMSE (% mean) | NRMSE (% range) | MAE (counts) |
|---|---|---|---|---|---|
| FastESIM | 5 | 0.158 | 232.8% | 1.3% | 0.023 |
| FastESIM | 10 | 0.180 | 264.6% | 1.4% | 0.029 |
| ESIM | 5 | 0.193 | 284.9% | 1.5% | 0.031 |
| FastESIM | 25 | 0.194 | 285.7% | 1.6% | 0.033 |
| FastESIM | 50 | 0.219 | 322.6% | 1.7% | 0.041 |
| ESIM | 10 | 0.280 | 412.8% | 2.2% | 0.053 |
| FastESIM | 100 | 0.312 | 457.0% | 2.5% | 0.067 |
| FastESIM | 125 | 0.350 | 512.3% | 2.8% | 0.077 |
| ESIM | 25 | 0.351 | 514.8% | 2.8% | 0.073 |
| ESIM | 50 | 0.376 | 551.7% | 3.0% | 0.080 |
| ESIM | 100 | 0.389 | 569.2% | 3.1% | 0.082 |
| ESIM | 125 | 0.391 | 572.0% | 3.1% | 0.082 |

**Table A.5:** *Sparse-Aware Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.1$ on the range $0..1$. All metrics show the same order.*

| Method | Key-frame Interval | Active RMSE | Precision | Recall | F1 |
|---|---|---|---|---|---|
| FastESIM | 5 | 0.614 | 0.8602 | 0.7896 | 0.8229 |
| FastESIM | 10 | 0.689 | 0.8144 | 0.7368 | 0.7730 |
| ESIM | 5 | 0.711 | 0.8130 | 0.7136 | 0.7592 |
| FastESIM | 25 | 0.734 | 0.7773 | 0.6945 | 0.7331 |
| FastESIM | 50 | 0.798 | 0.7118 | 0.6318 | 0.6690 |
| ESIM | 10 | 0.937 | 0.6391 | 0.5295 | 0.5708 |
| FastESIM | 100 | 0.971 | 0.5004 | 0.4416 | 0.4685 |
| FastESIM | 125 | 1.034 | 0.4193 | 0.3684 | 0.3913 |
| ESIM | 25 | 1.098 | 0.3839 | 0.2803 | 0.3111 |
| ESIM | 50 | 1.165 | 0.2444 | 0.1529 | 0.1801 |
| ESIM | 100 | 1.209 | 0.1525 | 0.0783 | 0.1001 |
| ESIM | 125 | 1.221 | 0.1342 | 0.0634 | 0.0835 |

**Table A.6:** *Sparse-Aware Event Metrics of the Realistic Room scene. Generated with a contrast threshold of $C = 0.05$ on the range $0..1$. All metrics show the same order.*

| Method | Key-frame Interval | Active RMSE | Precision | Recall | F1 |
|---|---|---|---|---|---|
| FastESIM | 5 | 0.740 | 0.8658 | 0.7824 | 0.8217 |
| FastESIM | 10 | 0.822 | 0.8225 | 0.7334 | 0.7750 |
| ESIM | 5 | 0.903 | 0.8272 | 0.7013 | 0.7583 |
| FastESIM | 25 | 0.873 | 0.7884 | 0.6928 | 0.7372 |
| FastESIM | 50 | 0.953 | 0.7328 | 0.6393 | 0.6826 |
| ESIM | 10 | 1.209 | 0.6796 | 0.5467 | 0.6005 |
| FastESIM | 100 | 1.212 | 0.5550 | 0.4815 | 0.5150 |
| FastESIM | 125 | 1.317 | 0.4839 | 0.4180 | 0.4476 |
| ESIM | 25 | 1.422 | 0.4698 | 0.3403 | 0.3853 |
| ESIM | 50 | 1.511 | 0.3472 | 0.2191 | 0.2598 |
| ESIM | 100 | 1.590 | 0.2522 | 0.1258 | 0.1636 |
| ESIM | 125 | 1.613 | 0.2324 | 0.1049 | 0.1412 |

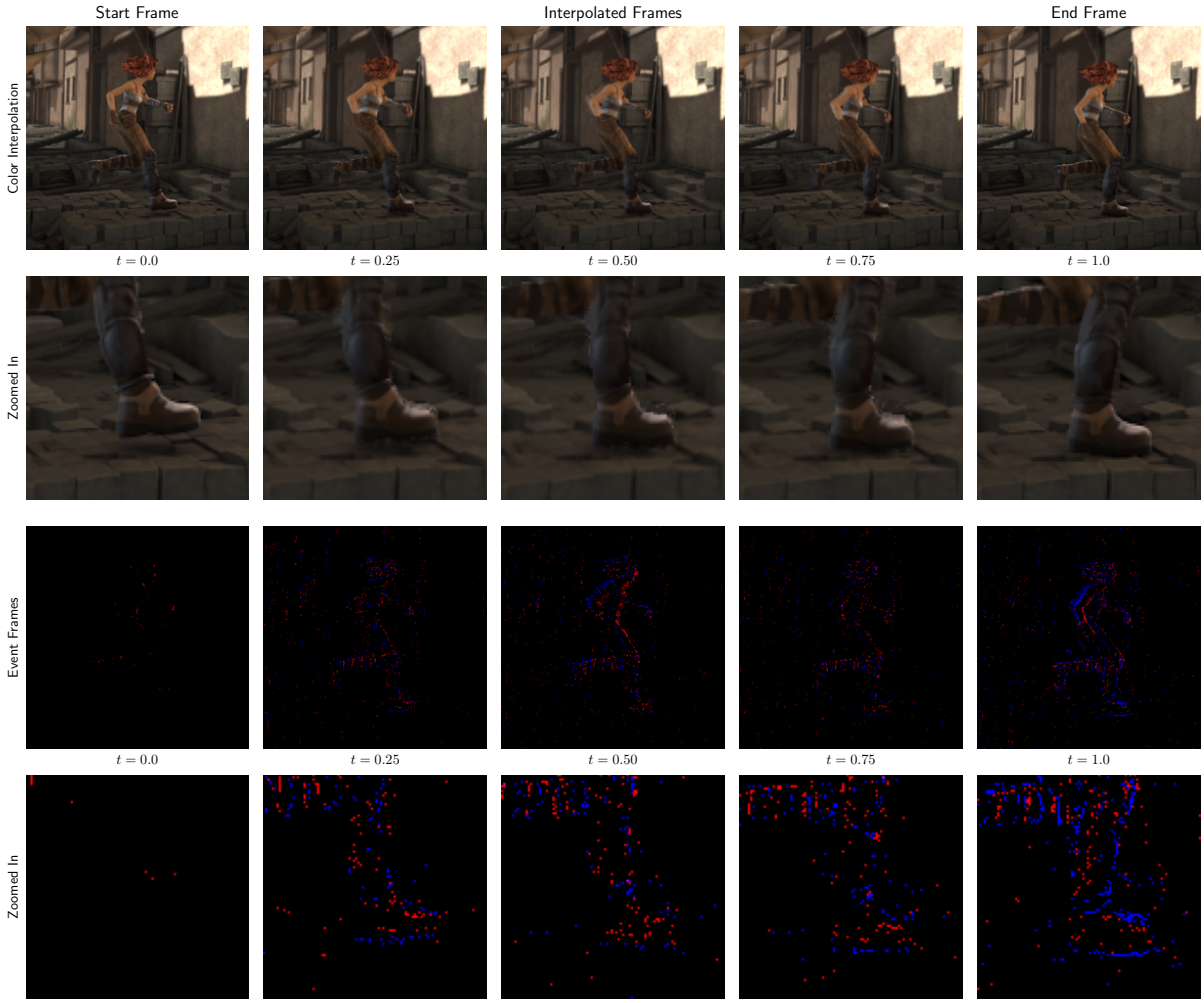



**Figure A.1:** *Alley 2 scene with generated event frames, from the Sintel Dataset. Interpolation and Events generated by C++ version.*

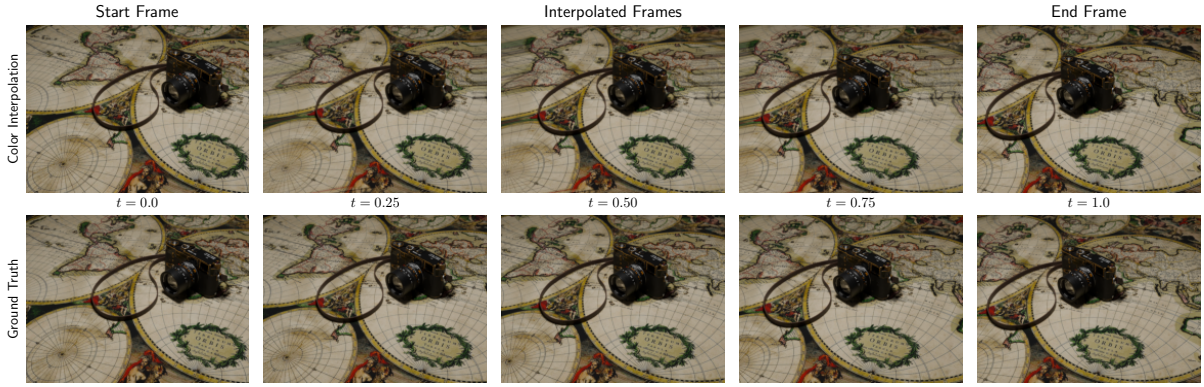



**Figure A.2:** *A scene with a camera. Interpolation generated by C++ version.*

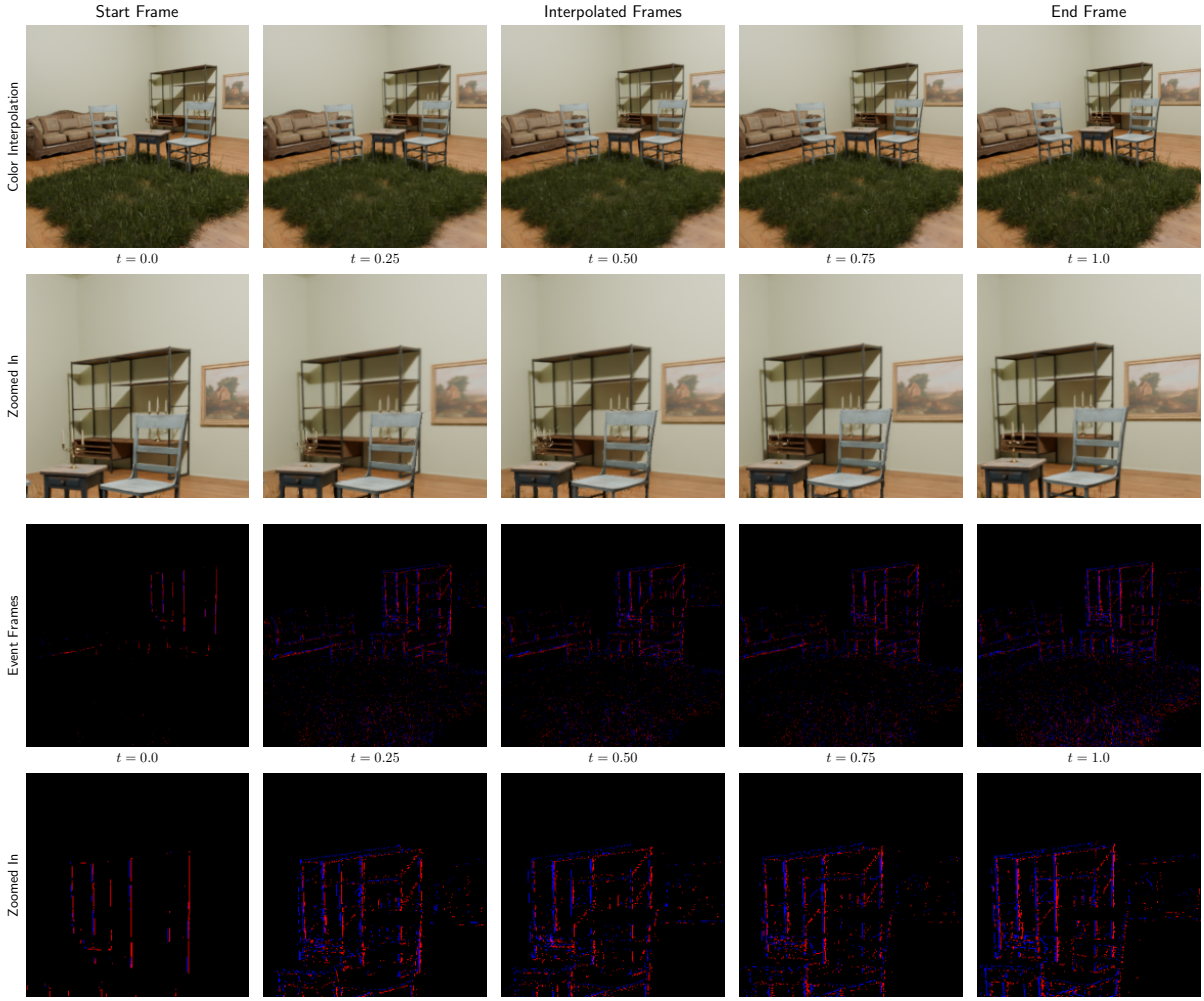
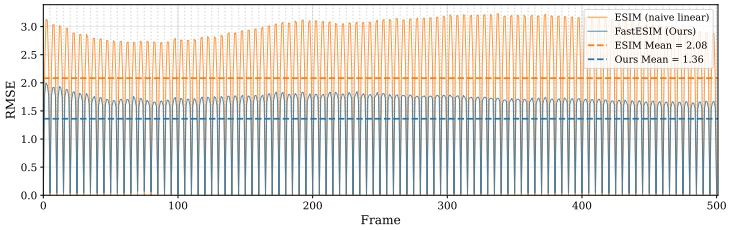
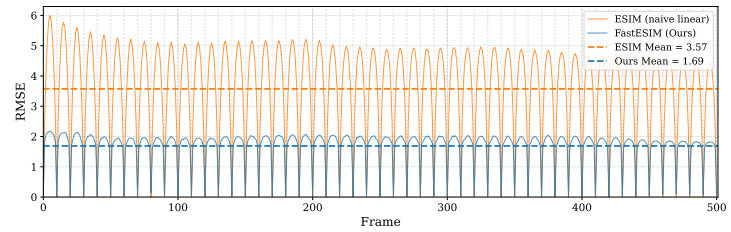


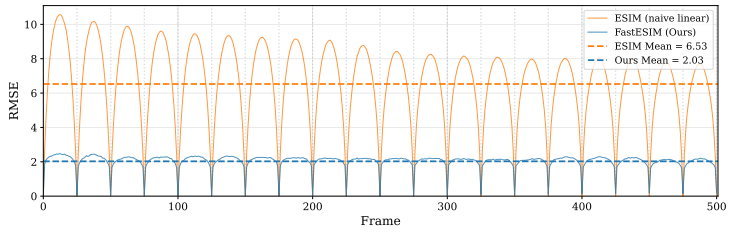

**Figure A.3:** *Complex house scene rendered in Blender. Interpolation and Events generated by C++ version.*



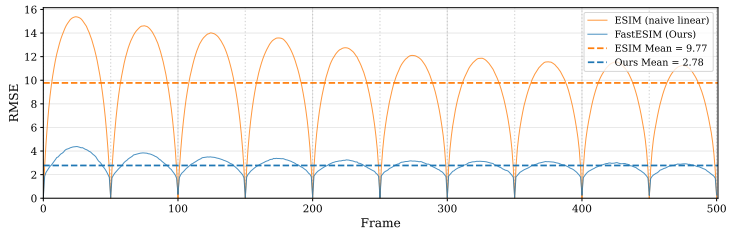

**(a)** *RMSE with key-frame interval of* $5$ *frames.*



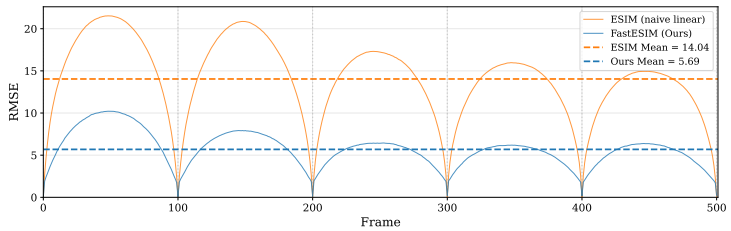

**(b)** *RMSE with key-frame interval of* $10$ *frames.*



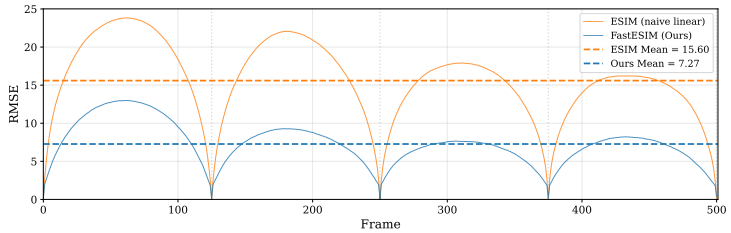

**(c)** *RMSE with key-frame interval of* $25$ *frames.*




**(d)** *RMSE with key-frame interval of* $50$ *frames.*

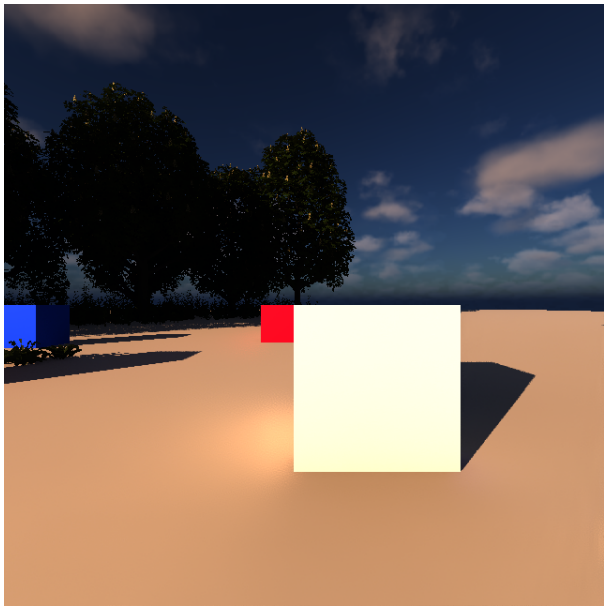



**(e)** *RMSE with key-frame interval of* $100$ *frames.*

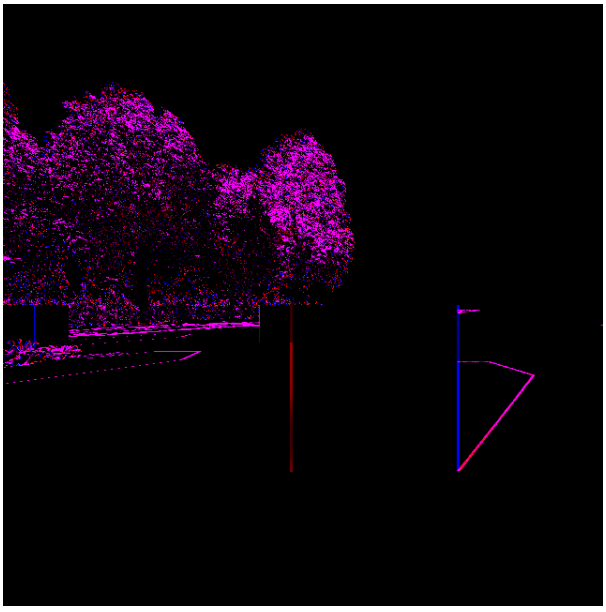



**(f)** *RMSE with key-frame interval of* $125$ *frames.*

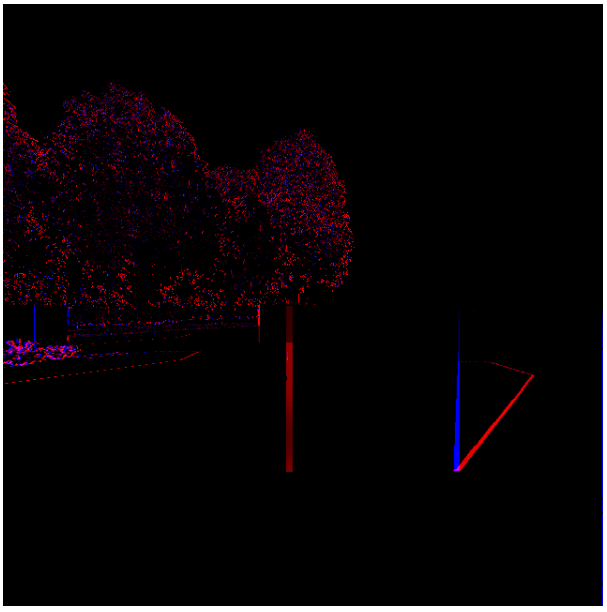
**Figure A.4:** *Comparison between the ESIM and FastESIM method for different key-frame intervals in the same sequence of* $500$ *frames. The RMSE is shown here in the range* $0..255$.



**(a)** *Color Frame*



**(b)** *ESIM Frame*

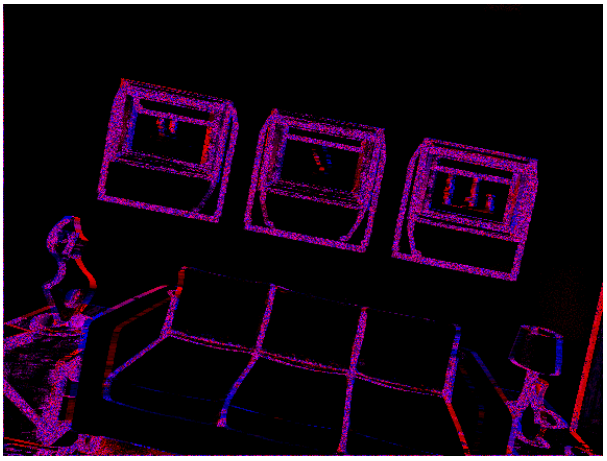


**(c)** *FastESIM Frame*

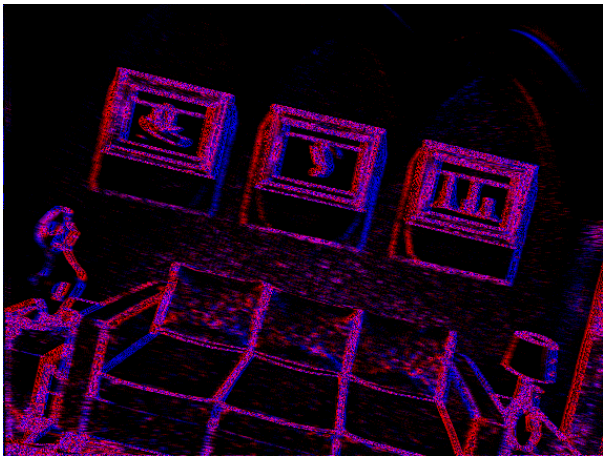
**Figure A.5:** *Test Scene rendered in Unreal Engine. Events generated by Unreal plug-in.*



**(a)** *Color Frame*



**(b)** *ESIM Frame*



**(c)** *FastESIM Frame*

**Figure A.6:** *Photorealistic room scene rendered in Unreal Engine. Events generated by Unreal plug-in.*

# B

# Source Code of Implemented Methods

## B.1. Implementation of Stand-Alone Version

This implementation of the novel FastV2E method calculates the events that occur between two given frames. It can be used as a command line tool or as a python package. The full source code is available on github.com/ReinierZwikker/FastEventSim.

**Table B.1:** *The source files used in the C++ version of the FastEventGenerator.*

| Filename | Purpose |
|---|---|
| `main_cmd.h/.cpp` | Provides command line interface to FastV2EFrame |
| `main_lib.h/.cpp` | Provides binding of FastV2EFrame to python |
| `FastV2EFrame.h/.cpp` | Contains the FastV2EFrame class, that contains two grids and executes the pipeline. |
| `pipeline/cpu_pipeline.h/.cpp` | Contains the pipeline implemented on CPU. This implementation is multi-threaded on the CPU, and sequences the vertex and edge collisions, chaining, and merging. |
| `cpu/vertex_collisions.h/.cpp` | Calculates the vertex collisions. |
| `cpu/edge_collisions.h/.cpp` | Calculates the edge collisions. |
| `cpu/intersect.h/.cpp` | Provides functions to calculate vertex and edge intersections. |
| `both/collision.h` | Implements Collision struct. Works for both CPU and CUDA implementations. |
| `both/leaving_entering.cuh` | Provides functions to calculate leaving and entering triangles for collisions. Works for both CPU and CUDA implementations. |
| `both/pipeline_utils.h` | Provides Directions struct. |
| `cpu/chaining.h/.cpp` | Converts collisions into control points. |
| `cpu/events.h/.cpp` | Generates events from control points. |
| `grid/grid_parts.h` | Provides declarations for all parts of the grid. |
| `grid/Grid.h/.cpp` | Implements the Grid class, that contains Vertices, Edges, and Triangles. |
| `grid/Vertex.h/.cpp` | Implements the Vertex class. |
| `grid/Edge.h/.cpp` | Implements the Edge class. |
| `grid/Triangle.h/.cpp` | Implements the Triangle class. |
| `grid/Chain.h/.cpp` | Implements the Chain class. |
| `utils/Color.h` | Provides color handling utilities. |
| `utils/constants.h` | Provides constants and tolerance definitions. |
| `utils/Image.h/.cpp` | Implements the Image class, that can read, create, and write raster images. |
| `utils/interpolate.h` | Provides interpolation functions. |
| `utils/MathVector.h` | Provides a two-dimensional vector template, for use in vector math. |
| `utils/lib_utils.h/.cpp` | Provides conversion function for the python binding. |

## B.2. Implementation of Unreal Engine Plug-In

This implementation of original ESIM method [3] and the novel FastESIM method calculates the events using Unreal Engine. It is integrated into an Unreal Engine project. The full source code is available on github.com/ReinierZwikker/FastEventGeneratorPlugin.

**Table B.2:** *The source files used in the Unreal Engine version of the FastEventGenerator.*

| Filename | Purpose |
|---|---|
| *Game Thread* | |
| `FastEventController.h/.cpp` | Controls the simulator settings and game world, and handles API calls. |
| `FastEventCharacter.h/.cpp` | Holds the camera object. |
| `DebugHUD.h/.cpp` | Displays information about the simulator on the screen. |
| *Rendering Thread* | |
| `FastEventShader.h/.cpp` | Handles the Plug-In module startup and shutdown. |
| `FastEventViewExtension.h/.cpp` | Requests resources from and queues the shaders into the RDG. |
| `ZMQSenderThread.h/.cpp` | Handles the sending frames over the ZeroMQ socket. |
| *Shaders* | |
| `ESIM/ESIMShader.usf` | Implements the ESIM simulation approach. |
| `ESIM/MaxStrideShader.usf` | Computes one stride to find to maximum flow. |
| `FastESIM/FastESIMShader.usf` | Implements the FastESIM simulation approach. |
| `FastESIM/VelocityMagnitudeShader.usf` | Pre-Processes the velocity pass for the FastESIM shader. Outputs control points and event frames. |
| `FastESIM/MaxDirectionalStrideShader.usf` | Computes one stride to find to maximum flow and maximum directional flow. |
| `FastESIM/EventsShader.usf` | Converts control points from the FastESIM shader into events. |
| `FastESIM/EventFrameShader.usf` | Converts control points from the FastESIM shader into event frames. |
| `General/ConvertDepthShader.usf` | Converts depth into world coordinates for exporting. |
| `General/ConvertVelocityShader.usf` | Converts velocity into screen space coordinates. |
| `General/ExposureShader.usf` | Converts exposure from HDR image to true brightness for exporting. |
| `Debug/WindowExtendDebugShader.usf` | Displays geometric information about the current search window on the screen. |

# C
# Interfaces

The interface to use each implementation is shown here. More information is available in the documentation of each implementation.

## C.1. Command-Line Interface

The command-line interface provides a direct way of using the stand-alone version. It converts a pair of frames, with flow and depth information, to 100 color and 100 events frames, spaced evenly between the two input frames. The interface can be used as follows:

```
./FastEventGenerator
  <Color Frame 0>.png <Color Frame 1>.png
  <Flow Frame 0 to 1>.flo <Flow Frame 1 to 0>.flo
  <Depth Frame 0>.dpt <Depth Frame 1>.dpt
  <output directory>
```

The `.flo` and `.dpt` files are read using the Sintel dataset format. The file consists of a 32-bit float verification number, in this case $202021.25$, the width and height of the image as a 32-bit integer and the frame data, encoded as 32-bit floating points numbers.

## C.2. Python Package Interface

The python binding provides an easy way to integrate the method into other image processing pipelines. This implementation is more configurable and allows for sampling frames, generating control points, or simulating events directly.

```
import fast_v2e

frame = fast_v2e.FastV2EFrame(<frame height>, <frame width>)

frame.set_data_bidirectional_numpy(
    <Color Frame 0>,
    <Flow Frame 0 to 1 X>, <Flow Frame 0 to 1 Y>,
    <Depth Frame 0>,
    <Color Frame 1>,
    <Flow Frame 1 to 0 X>, <Flow Frame 1 to 0 Y>,
    <Depth Frame 1>
  )

frame.calculate_frame(<Merge Bidirectional Enabled>,
                      <CUDA Enabled (Unimplemented)>)

frames = frame.sample_frames(<List of times to sample between 0 and 1>)
    or
control_points = frame.get_control_points()
    or
events = frame.get_events()
```

## C.3. Unreal Engine Plug-In Interface

Finally, the Unreal Engine 5.6 plug-in is installed by placing it in the `plugin` folder in an Unreal project. From here, it can be enabled as the world game-mode and controlled via the API using the python client or by using a custom client that communicates via ZeroMQ with the plug-in. From here, the simulation can be controlled and the velocity of the camera can be determined. Event frames, as well as color, depth, and optical flow frames can be retrieved from the plug-in over ZeroMQ.