CornellEngineering
Electrical and Computer Engineering

# UNREAL ENGINE AS A VISION BASED AUTONOMOUS MOBILE ROBOT SIMULATION TOOL

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted By:

Haritha Muralidharan (hm535)

MEng Field Advisor: Silvia Ferrari

Degree Date: January 2019

# ABSTRACT

**Master of Engineering Program**

**Cornell University**

**Design Program Report**

**Project:** Unreal Game Engine as a Vision Based Autonomous Mobile Robot Simulation Tool

**Author:** Haritha Muralidharan (hm535)

**Abstract:**

Testing vision-based decision-making algorithms in real-life is a difficult and expensive process due to the complex and diverse situations in which programs have to developed and tested. In recent years, with the development of high-performance graphics machines, it has become possible to test such diverse scenarios through simulated environments. In this project, we explore the possibility of using Unreal Engine, a hyper-realistic game development platform, as a simulation tool for testing motion detection through optical flow subtraction models.

## EXECUTIVE SUMMARY

While motion detection has been extensively studied, tracking an object of interest when the camera itself is in motion is still a challenge. This project considers optical flow models for identify moving objects of interest in dynamically changing scenes. Conventional cameras are monocular, and lose one dimension of information when capturing images or videos of the physical world. Therefore, geometry-based optical flow models based on stereo-vision are proposed for more estimating ego-motion more accurately. Stereo-cameras are composed of two monocular cameras, and are able to recover depth information through disparity calculation algorithms. The recovered depth information can then be used for analyzing changes in the scene – i.e. to identify changes caused by ego-motion, and subsequently to identify other objects in the scene moving independently of the camera.

In this project, the Unreal Engine (UE) gaming platform is used as the main development tool. The hyper-real graphics on UE make it possible to simulate real-life to a high degree of accuracy. A stereo-camera is modeled in the UE, and computer vision algorithms are applied directly to the images captured by the camera. Two different methods of disparity calculation were tested and compared to the ground truth provided by th UE. Subsequently, geometry-based methods were used for optical flow subtraction to identify and remove the effects of ego-motion.

Through the course of this project, various tools and libraries were implemented to enable future developers and researchers to use the optical flow calculation tools easily without having to re-implement them from scratch.

## Table of Contents

# 1 INTRODUCTION

## 1.1 Motivation

With recent advances in computer vision, it has become easier than ever to implement scene-based decision algorithms for autonomous mobile robots. Such algorithms are essential for a wide variety of applications, including autonomous vehicles, security and surveillance, and patient monitoring [1].

One of the primary research themes at the Laboratory of Intelligent Systems and Controls (LISC) is the development of target-tracking systems. However, given the complexity of the problem, and the multitude of environments required for testing, physical validation of such algorithms will be expensive, slow and challenging.  To address this, LISC proposes using the Unreal Engine (UE) game development platform as a simulation tool for development and testing of new algorithms. UE was chosen for its hyper-realistic graphics models that closely mirror real-life. It is possible to model visual and dynamic models on UE to a high degree of accuracy. Furthermore, UE is programmed with C++, making it for algorithms to be translated from the simulation to the real world.

One of the goals of this project is to model camera modes in UE such that they simulate the behavior of physical cameras. The virtual camera has easy-to-use interface such that users are able to toggle between camera modes without having to modify the source code. Furthermore, the camera model is sufficiently decoupled so that it can be attached to various vehicles, making them reusable between projects, without researchers having to start from scratch. Another goal of this project is to explore how the camera models can be used for motion detection.

## 1.2 Problem Statement

Detecting the motion of moving objects is integral for target-tracking; a mobile robot will have to be able to identify the location and direction of moving targets in order to be able to track it efficiently. When a camera is stationary, it is easy to identify moving targets; the robot simply has to identify the clusters of pixels that are moving from frame-to-frame. For example, the inert traffic camera, in Figure 1 can identify the locations of pedestrians by comparing pixel intensities of cluster between consecutive frames.

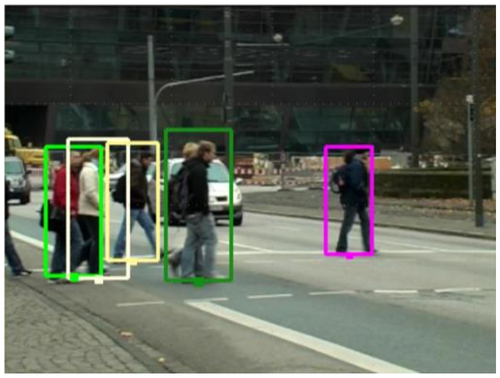


***Figure 1*** **–** Bounding boxes around moving pedestrians as identified by a stationary traffic camera.

However, when the robot itself is moving, then the whole scene would have appeared to move from the camera's perspective[1]. Therefore, when the robot is moving, it becomes difficult to identify the targets that are moving independently of the camera. Addressing this problem requires ego-motion subtraction; i.e. we need to identify the change in the scene caused by the camera and subtract it from the total observed change in the scene. The remaining components would identify the motion of objects moving independently of the camera. One way to solve the ego-motion subtraction problem would be through optical flow models.

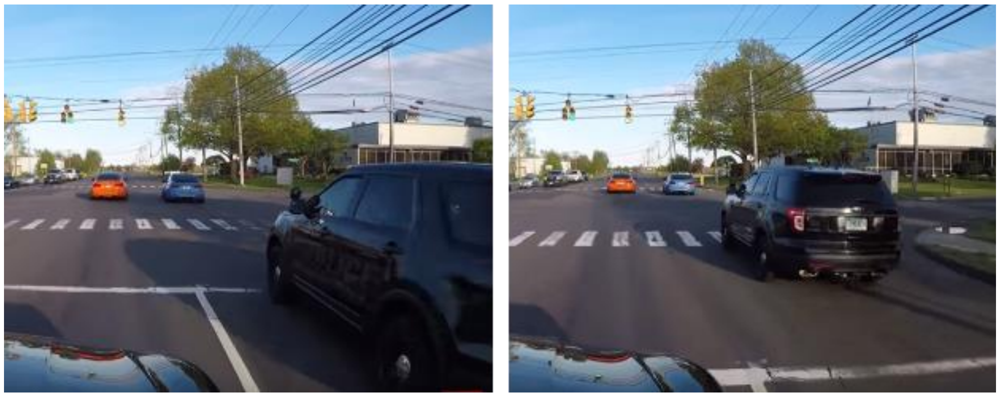


***Figure 2*** **–** Two images captured by a camera mounted on a moving car. In this scene, there are objects (other cars) that are moving independently of the car/camera. Identifying the motion of these independent objects is requires ego-motion subtraction.

## 1.3 Objectives and Scope

The main objective of this project was to create easy-to-use tools on the UE environment without having to redevelop programs from scratch. The tools created include camera models, optical flow models, and stereo-vision models. The models were based on previous work done in the lab, which includes the setting up of basic camera simulation in UE, as well as the derivation of the ego-motion subtraction algorithm.

[1] In other words, rotating a camera in a stationary world is equivalent to rotating the world around a stationary camera. A similar argument applies to translation as well.

## 1.4 Development Platform and Tools

### 1.4.1 Unreal Engine Simulation Environment

Unreal Engine provides a hyper-realistic graphics environment with a complete suite of programming tools for developers. The suite includes many high-quality toolboxes that render virtual bodies (such as cars, quadcopters, humans, etc.) that follow physical laws (including lighting, collision, gravity, etc.), which allows researchers to emulate diverse real-world situations with ease [2]. Furthermore, UE has multiple interfaces that allows researchers to develop with either C++ on Microsoft Visual Studios, or with a graphical programming interface (called Blueprints in UE). Figures 3 and 4 show two examples of simulated environments in UE.

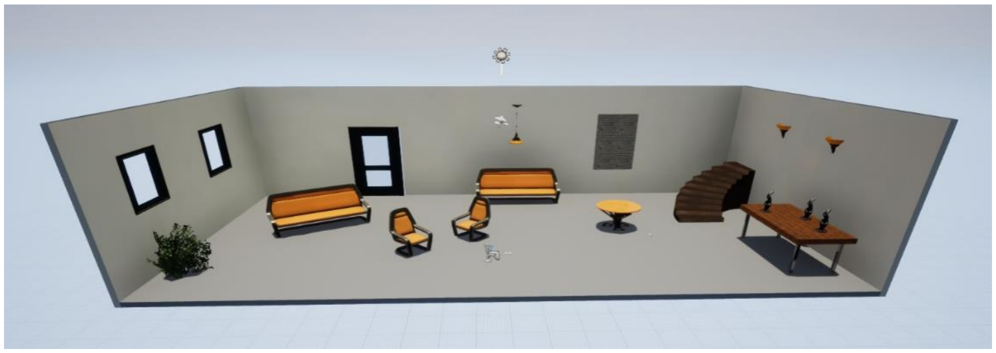


***Figure 3*** – A minimal environment of a room created on UE.

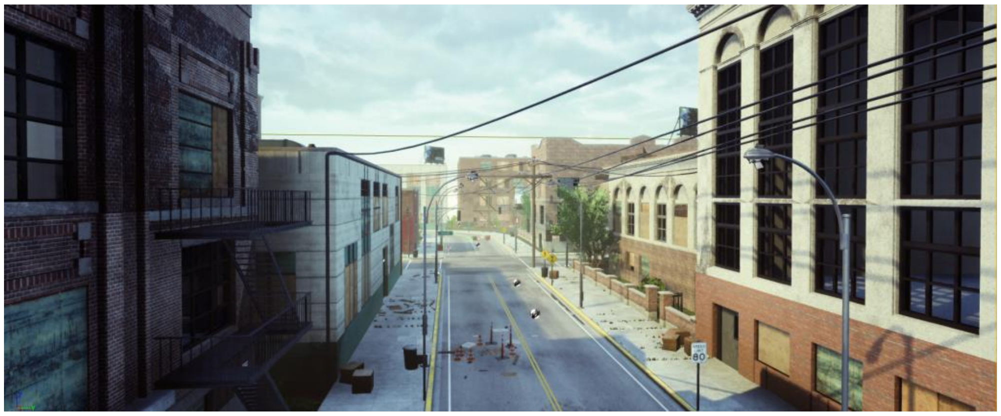


***Figure 4*** – An "Industrial City" environment created on UE.

### 1.4.2 OpenCV Toolbox

The opensource OpenCV libraries were added to UE as a third-party plugin for the development of computer vision algorithms. OpenCV has a large number of inbuilt functions for calculating vision-based optical flow, including the Lucas-Kanade and the Farneback algorithms. The toolbox also provides a simplified form of disparity calculation using the SGM method.

# 2 Camera Models

In order to formulate the model for ego-motion subtraction, it is necessary to first understand fundamental camera models, which will be described in this section.

## 2.1 Basic Camera Model

The fundamental camera projection model is described by Figure 5 . The perspective projection of a point $\mathcal{P}$ in the real world is represented by $p$ in the image plane.

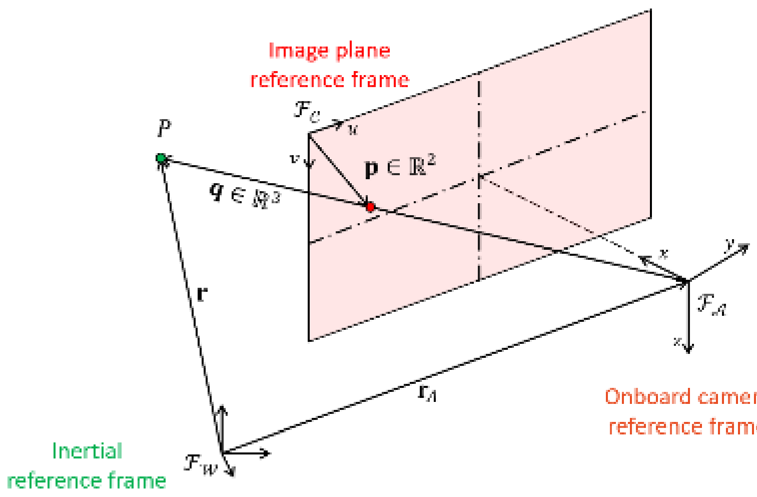


***Figure 5 –*** Projection model of a standard camera, with the focal axis denoted by the $\boldsymbol{x}$ axis.

The coordinates of $p$ can be calculated as:

$$p_y = \lambda \frac{Y}{X} \qquad \text{-----------------------------------------------(1)}$$

$$p_z = \lambda \frac{Z}{X} \qquad \text{-----------------------------------------------(2)}$$

where $[X, Y, Z]$ refer to the coordinates of $\mathcal{P}$ with reference to the camera frame, and $\lambda$ refers to the focal length of the camera [3]. Single monocular cameras lose one dimension (depth) of information, as a 3-dimensional (3-D) point $\mathcal{P}$ is reduced to a 2-D point $p$. In conventional models, the focal axis is typically the $z$ axis. In this report, the $x$ axis will be used as the focal axis to match the convention used in UE, so as to reduce the number of transformations required.

## 2.2 Stereo Camera Model

Stereo cameras attempt to regain depth information through the use of two monocular cameras, as shown in Figure 6. The two cameras are displaced along a baseline, such that their optical axes are parallel.

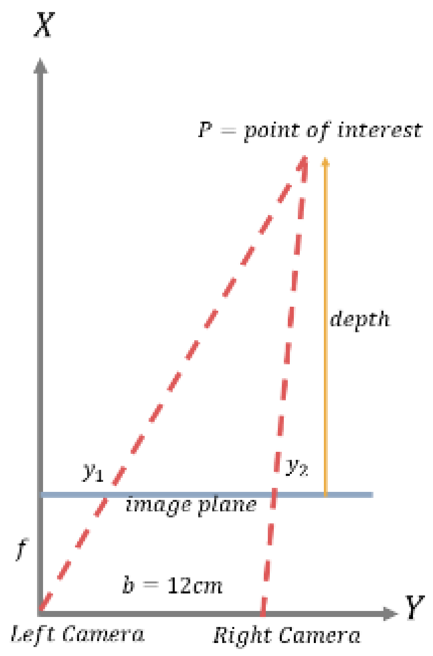


***Figure 6 –*** Geometric model of the triangulation model used for recovering depth in stereo-vision.

For a generic stereo-camera, the $x$ axis represents the baseline – i.e. the axis parallel to, and connecting both the monocular cameras; the $z$ axis represents the optical axis of either camera; and the $y$ axis is obtained using the right-hand rule. For a single physical point $P$, that appears in both the cameras' fields of view, the corresponding image coordinates will be displaced in the direction of the $x$ axis. The depth of point $\mathcal{P}$ can be recovered through the following equation:

$$Z = \frac{b \times \lambda}{x_1 - x_2} \qquad \text{---------------------------------------------(3)}$$

where $Z$ refers to the depth of the point of interest, $b$ refers to the baseline distance (the distance between the two cameras), and $x_1$ and $x_2$ refer the $p_x$ coordinated of the point in either image [4].

# 3 Optical Flow Models

## 3.1 Theory of Optical Flow

For target-tracking applications, we are interested in estimating how much an object of interest has moved in the real world, given its observed movement in the camera's observed scene, which is given by optical flow. Fundamentally, optical flow is simply the observed change in pixels in the image plane when the point of interest moves in the real world.

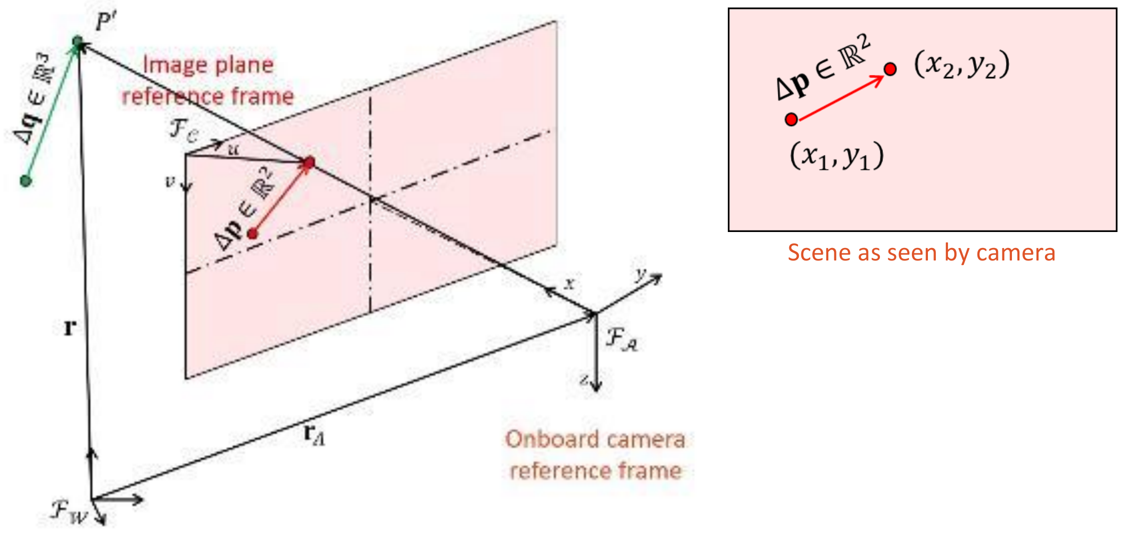

$P'$
Image plane reference frame
$\Delta \mathbf{q} \in \mathbb{R}^3$
$\mathcal{F}_{\mathcal{C}}$
$u$
$v$
$\Delta \mathbf{p} \in \mathbb{R}^2$
$\mathbf{r}$
$x$
$y$
$\mathcal{F}_{\mathcal{A}}$
$z$
$\mathbf{r}_{\mathcal{A}}$
Onboard camera reference frame
$\mathcal{F}_{\mathcal{W}}$
$\Delta \mathbf{p} \in \mathbb{R}^2$
$(x_2, y_2)$
$(x_1, y_1)$
Scene as seen by camera

***Figure 7*** **–** Optical flow due to the movement of a single point, as seen by a stationary camera.

In Figure 7 above, the symbols denote the following variables [5]:

i) $\mathcal{F}_{\mathcal{W}}$ is the inertial reference frame in $\mathbb{R}^3$

ii) $\mathcal{F}_{\mathcal{A}}$ is the body reference frame of the camera, which has a focal point $\mathcal{A}$.

iii) $\mathcal{F}_{\mathcal{C}}$ is the reference frame of the image plane, located at focal length $\lambda$ from $\mathcal{A}$ along the camera's optical axis. The origin of $\mathcal{F}_{\mathcal{C}}$ is given by point $\mathcal{C}$, located at the upper left corner of the image

iv) $\mathcal{P} \in \mathbb{R}^3$ refers to a point of interest in the real world

v) $r_{\mathcal{P}}$ is the position of $\mathcal{P}$ relative to $\mathcal{F}_{\mathcal{W}}$

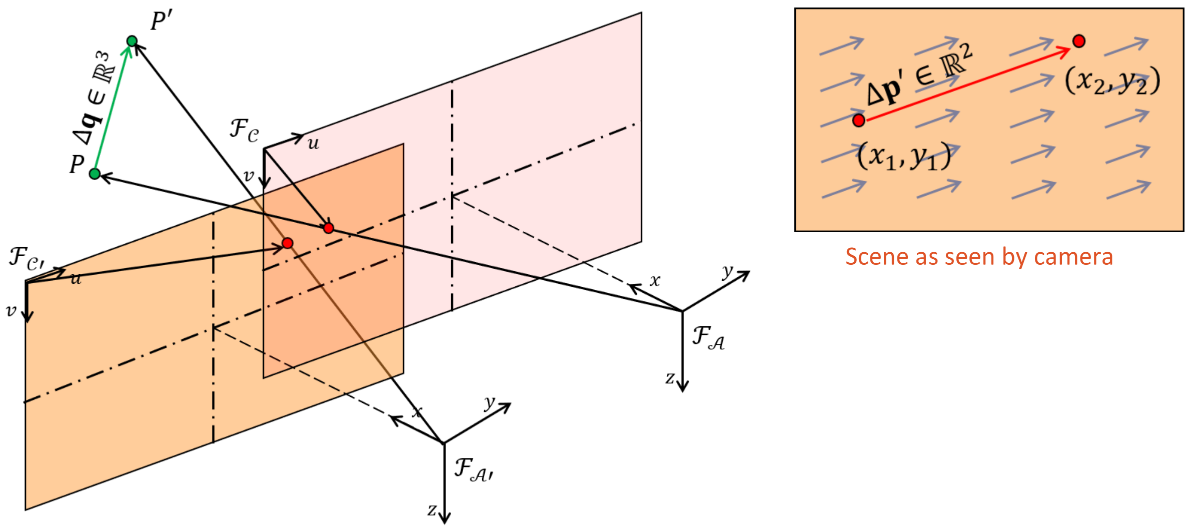
vi) $r_{\mathcal{A}}$ is the position of $\mathcal{A}$ relative to $\mathcal{F}_{\mathcal{W}}$

vii) $r_{\mathcal{C}}$ is the position of $\mathcal{C}$ relative to $\mathcal{F}_{\mathcal{W}}$.

viii) $q \in \mathbb{R}^3$ is the position of $\mathcal{P}$ with respect to $\mathcal{A}$ is given by $q = r_{\mathcal{P}} - r_{\mathcal{A}}$. $q$ with respect to $\mathcal{F}_{\mathcal{W}}$ is written as ${}^{\mathcal{W}}q$, and with respect to $\mathcal{F}_{\mathcal{A}}$ is written as ${}^{\mathcal{A}}q$.

ix) $p \in \mathbb{R}^2$ is the perspective projection of $\mathcal{P}$ onto the image. It is the point at which $q$ intersects the image plane.

If the point $\mathcal{P}$ (Figure 7) moves to point $\mathcal{P}'$ by an amount $\Delta q$ in the real world, then it moves by a corresponding amount $\Delta p$ in the image. In this example, optical flow simply refers to the $\Delta p$ in pixel coordinates. Since only a single pixel, or a single cluster of pixels is moving, it is easy to detect the motion of the moving object. $\Delta q$ can then recovered by inverting the perspective projection equation using $\Delta p$. However, if the camera is also moving, then flow observed by the camera is more complex, as shown in Figure 8 [5].



$P'$
$\Delta \mathbf{q} \in \mathbb{R}^3$
$P$
$\mathcal{F}_{\mathcal{C}}$
$u$
$v$
$\mathcal{F}_{\mathcal{C}'}$
$u$
$v$
$x$
$y$
$\mathcal{F}_{\mathcal{A}}$
$z$
$x$
$y$
$\mathcal{F}_{\mathcal{A}'}$
$z$
$\Delta \mathbf{p}' \in \mathbb{R}^2$
$(x_2, y_2)$
$(x_1, y_1)$
Scene as seen by camera

***Figure 8 –*** Optical flow due to the independent motions of a point ***P*** and the camera.

If the point $\mathcal{P}$ moves to point $\mathcal{P}'$ by an amount $\Delta q$ in the real world, and the camera independently moves from $\mathcal{F}_{\mathcal{A}}$ to $\mathcal{F}_{\mathcal{A}'}$, the observed flow as seen by the camera becomes compounded by both the motions. The observed flow $\Delta p'$ is a sum of both the flow caused by the movement of $\mathcal{P}$ and of the camera. As a result, $\Delta q$ cannot be recovered easily. Furthermore, the camera scene is also populated by the estimated flow of background pixels ($\Delta a$) even if those points are stationary in the real world. In order to recover $\Delta p$ from $\Delta p'$, we need to subtract the background flow from the total observed flow:

$$\Delta p = \Delta p' - \Delta a \qquad \text{------------------------------------------------}(4)$$

This is the fundamental model of ego-motion subtraction. $\Delta a$ is geometrically estimated using the mobile robot's body velocities, and the depth of the scene at each point, as discussed in Section 3.3 [5].

## 3.2 Observed Optical Flow

The observed optical flow refers to the $\Delta p'$ term in Equation 4, and is calculated through computer vision algorithms. It is defined as the distribution of apparent flow velocities of the movement of brightness patterns in an image. Assuming that the pixel intensity of an object of interest is constant over time, the shift in pixel intensities between frames would imply a shift in the object in the object itself; and therefore, can be used to describe the apparent motion of the object. The output of an optical flow calculation would be a vector field of gradients that indicate how the pixel values are shifting. Mathematically, this is represented by the following partial differential equation (PDE) [6]:

$$\frac{\partial I}{\partial y}\frac{\partial y}{\partial t}+\frac{\partial I}{\partial z}\frac{\partial z}{\partial t}+\frac{\partial I}{\partial t}=0 \qquad \text{----------------------------------------------(5)}$$

where $I$ refers to the pixel intensity values, $(y, z)$ are the pixel coordinates using the UE convention, and $t$ is the time-step. An additional smoothness constraint is imposed on the PDE to ensure a tangible solution is reached. The smoothness constraints imposes that the flow velocities of neighboring pixels must be similar. This is based on the assumption that objects of finite size undergoing rigid motion would translate to a group of neighboring pixels moving at similar velocities. While there are many proposed algorithms for estimating and smoothing observed flow; the Farneback and Lucas-Kanade methods are considered in this project.

### 3.2.1 Farneback Algorithm

The Farneback algorithm is a dense method that calculates the pixel velocity at every pixel location. The algorithm uses polynomial expansion, where the neighborhood of each pixel is approximated by a polynomial:

$$f(\mathcal{X}) \sim \mathcal{X}^T A \mathcal{X} + b^T \mathcal{X} + c \qquad \text{----------------------------------------------(6)}$$

where $\mathcal{X} = [y \quad z]^T$ is the $1 \times 2$ vector representing pixel locations, $A$ is a $2 \times 2$ symmetric matrix of unknowns capturing information about the even component of the signal, $B$ is a $2 \times 1$ vector of unknowns capturing information about the odd component of the signal, and $c$ is an unknown scalar [6].

### 3.2.2 Lucas-Kanade Algorithm

The Lucas-Kanade algorithm is a sparse method that calculates the pixel velocity only at the locations of salient features. The Lucas-Kanade method also uses a pyramidal implementation from that refines motion from coarse to fine, thus allowing movement both, large and small, movements to be captured.

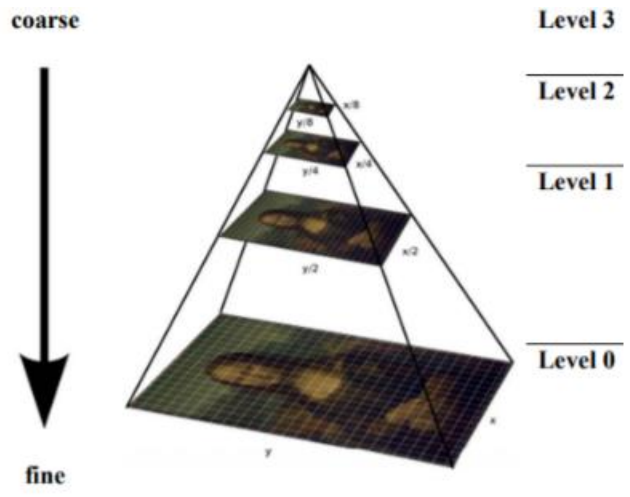


***Figure 9 –*** Image pyramid with four levels. At each level, the image is downsized. Optical flow is recursively calculated from the top of the pyramid (Level 3) and ends at the bottom (Level 0).

Both algorithms available in readily available in OpenCV toolboxes, and therefore do not have to be implemented from scratch [7].

## 3.3 Optical Flow from Ego-Motion

For subtraction of $\Delta a$ in Equation 4, we need to estimate the optical flow (termed geometric optical flow) that will be cause by the ego-motion of the camera. This section will derive the geometric optical flow.

First, we derive how the points in a stationary world move respective to the camera, when the robot is in motion. This movement is simply the time derivative of the vector $q$, and is denoted by $\dot{q}$. ${}^{\mathcal{A}}\dot{q}$ denotes the movement relative to the camera frame $\mathcal{F}_{\mathcal{A}}$. To obtain ${}^{\mathcal{A}}\dot{q}$ from $\dot{q}$, we make use of the transport theorem [5]:

$$ {}^{\mathcal{A}}\dot{q} = \frac{d}{dt}{}^{\mathcal{A}}q + {}^{\mathcal{A}}\omega_{\mathcal{A}/\mathcal{W}} \times {}^{\mathcal{A}}q \qquad \text{-----------------------------------------------(7)} $$

where ${}^{\mathcal{A}}\omega_{\mathcal{A}/\mathcal{W}} = [p \quad q \quad r]^T$ is the rotational velocity of $\mathcal{F}_{\mathcal{A}}$ relative to $\mathcal{F}_{\mathcal{W}}$ expressed in the frame of $\mathcal{F}_{\mathcal{A}}$. Letting ${}^{\mathcal{A}}q = [x \quad y \quad z]^T$, and $\frac{d}{dt}{}^{\mathcal{A}}q = [v_x \quad v_y \quad v_z]^T$. Equation 7 summarizes to:

$$ {}^{\mathcal{A}}\dot{q} = \begin{bmatrix} \dot{q}_x \\ \dot{q}_y \\ \dot{q}_z \end{bmatrix} = \begin{bmatrix} v_x - ry + qz \\ v_y + rx - pz \\ v_z - qx + py \end{bmatrix} \qquad \text{-----------------------------------------------(8)} $$

Next, we have to calculate the flow caused by ${}^{\mathcal{A}}\dot{q}$.

Based on Figure 5, the position of $\mathcal{C}$ with respect to $\mathcal{F}_{\mathcal{A}}$ is given by

$$ {}^{\mathcal{A}}r_{\mathcal{C}} = \begin{bmatrix} \lambda & -\frac{w}{2} & -\frac{h}{2} \end{bmatrix}^T \qquad \text{-----(9)} $$

where $\lambda$ is the focal length, $w$ and $h$ are the width and height of the image respectively. Therefore, to express $p$ with respect to $\mathcal{A}$:

$$ {}^{w}p = \begin{bmatrix} 0 & u & v \end{bmatrix}^T \qquad \text{-----(10a)} $$

$$ {}^{\mathcal{A}}p = {}^{\mathcal{A}}r_{\mathcal{C}} + {}^{\mathcal{C}}p = \begin{bmatrix} \lambda & u-\frac{w}{2} & v-\frac{h}{2} \end{bmatrix}^T \qquad \text{-----(10b)} $$

To express $p$ with respect to $\mathcal{C}$:

$$ {}^{\mathcal{C}}p = \begin{bmatrix} u-\frac{w}{2} & v-\frac{h}{2} \end{bmatrix}^T = \begin{bmatrix} \hat{u} & \hat{v} \end{bmatrix}^T \qquad \text{-----(11)} $$

This is also related to perspective projection Equations 1 and 2. The project equation rewritten in UE frame is given by [5]:

$$ \begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} = \lambda \begin{bmatrix} y/x \\ z/x \end{bmatrix} \qquad \text{-----(12)} $$

The derivative of Equation 12 (which is $\Delta a$ in Equation 4) is given by:

$$ \frac{d}{dt}\begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} = \lambda \frac{d}{dt} \begin{bmatrix} y/x \\ z/x \end{bmatrix} = \frac{\lambda}{x^2} \begin{bmatrix} \dot{q}_y x - \dot{q}_x y \\ \dot{q}_z x - \dot{q}_x x \end{bmatrix} \qquad \text{-----(13)} $$

Substituting in Equation 8, 13 can be rewritten as [5]:

$$ \frac{d}{dt}\begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} = \frac{\lambda}{x^2} \begin{bmatrix} -y & x & 0 & -zx & -zy & x^2+y^2 \\ -z & 0 & x & yx & -x^2-y^2 & yz \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ p \\ q \\ r \end{bmatrix} \qquad \text{-----(14a)} $$

Using the perspective projection equation:

$$\frac{d}{dt}\begin{bmatrix}\hat{u}\\ \hat{v}\end{bmatrix} = \lambda\begin{bmatrix} -\frac{\hat{u}}{x} & \frac{\lambda}{x} & 0 & -\hat{v} & -\frac{\hat{u}\hat{v}}{\lambda} & -\frac{(\hat{u}^2+\lambda^2)}{\lambda^2} \\ -\frac{\hat{v}}{x} & 0 & \frac{\lambda}{x} & \hat{u} & -\frac{(\hat{v}^2+\lambda^2)}{\lambda^2} & \frac{\hat{u}\hat{v}}{\lambda} \end{bmatrix}\begin{bmatrix} v_x \\ v_y \\ v_z \\ p \\ q \\ r \end{bmatrix} \qquad \text{-------------------------------(14b)}$$

Equation 14b gives the estimated geometric flow, which is the $\Delta a$ component in Equation 4. From the equation, we can see that the depth information ($x$) is required for ego-motion subtraction. However, since monocular cameras are not able to provide depth information, stereo-vision models that recover depth information are required for the subtraction model.

## 4 STEREO VISION MODELS

### 4.1 Theory of Stereo-Vision

In order to accurately recover the depth as described in Section 2.2, it is necessary to establish scene correspondence through feature matching. Given two images, one from each camera in the stereo model, we need to identify the corresponding coordinates of similar features in two stereo images. In stereo-vision, the disparity ($d$) refers to the difference between the coordinates of a single feature. Using the UE reference frame:

$$d = y_1 - y_2 \qquad \text{-------------------------------------------(15)}$$

where the two cameras are shifted along the $y$-axis. $d \in [0, D]$ where $D$ is the maximum expected disparity value that each pixel can take.

The disparity can then be inverted to obtain depth information. Intuitively, objects closer to the camera's focal point have a higher disparity, while objects further away have a lower disparity.

There are multiple proposed models for calculating the disparity of an image pair, and a current up-to-date list can be found on Middlebury or KITTI dataset websites. In this project, two popular models capable of delivering real-time information were considered.

### 4.2 Sparse Disparity Calculation

Sparse disparity calculation is the simplest way of recovering the depth information. It calculated the disparity only at key salient features. Given two images, the key features (using conventional corner detectors such as Scale-Invariant Feature Transform, SIFT) of either frame are first extracted. Each feature on the left frame is then matched with a feature from the right frame based on their similarity. The difference in coordinates between the matching pair is then used to calculate the disparity of that specific feature [4].

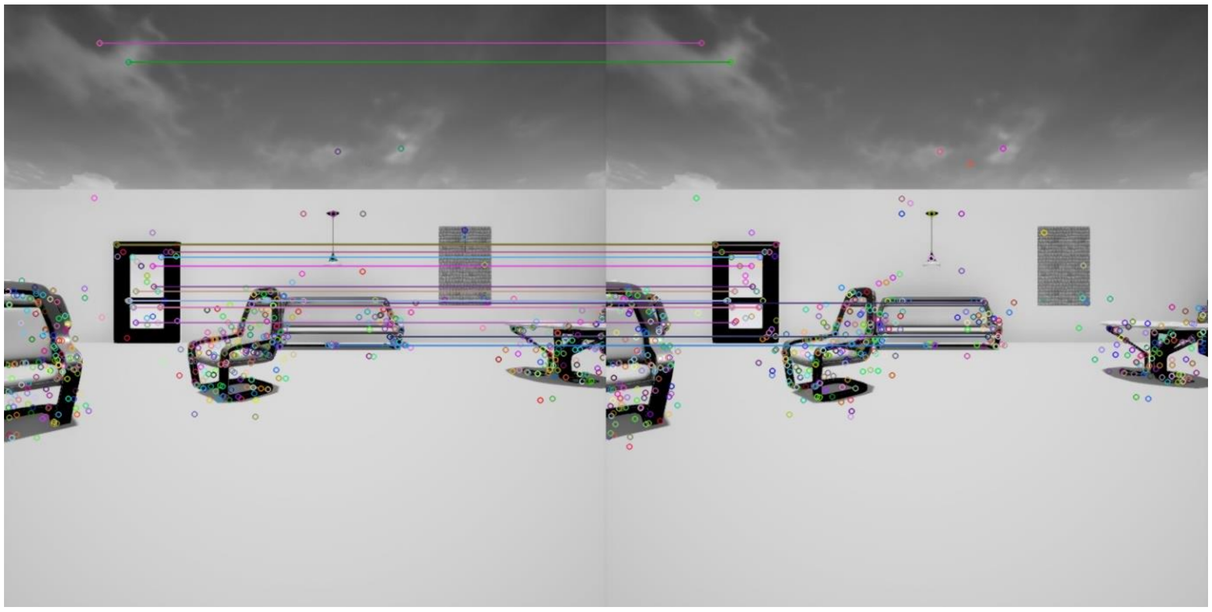


***Figure 10 –*** Sparse feature matching for disparity calculation. Each circle represent a matched features, while the lines connect 20 of the strongest matching pairs.

One clear disadvantage of this method is that disparity (and hence depth) information is not available at every pixel of the image. While this is sufficient for sparse optical flow calculation methods, such as the Lucas-Kanade method, it does not contain sufficient information for dense methods. Therefore, alternate algorithms capable of calculating dense disparity information are required. The challenge of calculating dense depth information lies in calculating the disparities in large uniform areas, such as a blank wall. Disparity calculation in uniform areas is difficult due to the lack of features, which are make it impossible to accurately establish scene correspondence between the left and right images of a stereo-camera. The address this, algorithms that impose a smoothness constraint based on known disparities of neighboring pixels are considered. The Semi-Global Matching (SGM) algorithm and the Patch-Match (PM) algorithm are two such algorithms that were implemented in this project.

## 4.3 Semi-Global Matching (SGM) Algorithm

The SGM method first calculates a pixelwise cost at each expected disparity using a user-defined cost metric. A common cost metric is the sum of absolute differences (proposed by Birchfield-Tomasi) [8]:

$$C(d) = |I_1(y,z) - I_2(y-d,z)| \qquad \text{-----------------------------------------------}(16)$$

where $I_1(y,z)$ is the pixel intensity of the left image, and $I_2(y-d,z)$ is the pixel intensity of the right image at the expected disparity. This cost is iteratively calculated for every value in $d \in [0, D]$.

We then impose a smoothness constraint by penalizing changes from neighbors' disparities. Each pixel is scanned in multiple directions to smooth against all neighbors [9].

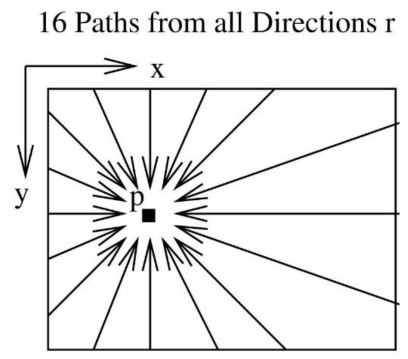


***Figure 11*** – In the smoothing function for the SGM algorithm, each pixel ***p*** is smoothed against all neighboring pixels.

Mathematically, the smoothness constraint is imposed by minimizing the following cost function [9]:

$$L_p(d) = C(d) + min(L_{p-r}(d), L_{p-r}(d-1) + S_M, L_{p-r}(d+1) + S_M, \underset{i}{min}(L_{p-r}(i) + S_L)) \qquad (17)$$

where $C(d)$ is the pixelwise cost for the current disparity being evaluated (from Equation 16), $L_{p-r}(d)$ refers to the neighbor pixel's cost at the current disparity, $L_{p-r}(d-1)$ and $L_{p-r}(d+1)$ is the neighbor pixel's cost at the neighboring disparities, and $L_{p-r}(i)$ is the neighbor pixel's cost at all other disparities. Small departures from the current disparity (i.e. $d-1$ and $d+1$) are penalized by a small penalty ($S_M$), while large departures from the current disparity are penalized by a large penalty ($S_L$). The current pixel's cost is then updated to include the minimum of all these costs to smooth against the incontinuities. The smoothing process is repeated for all pixels against every neighbor. To save on computation, we can choose to smooth against $4-$ or $8-$ connected neighbors instead of $16-$ connected neighbors. Finally, the disparity is chosen to be the value at which the smoothed cost is minimized. The pseudo-code for calculating the smoothed costs using the SGM algorithm is given in Code Block 1.

```
for each pixel in left_image:
    for each d in range[0, D]:
        calculate initial_pixel_cost()

for each pixel in left_image:
    for each neighbor in scanning_direction:
        for each d in range[0, D]:
            calculate smoothed_pixel_cost(current_pixel, neighbor_pixel)

def initial_pixel_cost():
    pixel_cost[d] = left_image(y, z) – right_image(y-d, z)

def smoothed_pixel_cost(current_pixel, neighbor_pixel):
    smooth_pixel_cost[current_pixel, d] = pixel_cost[current_pixel, d]
                              + min(pixel_cost[neighbor_pixel, d],
                              pixel_cost[neighbor_pixel, d-1] + small_penalty,
                               pixel_cost[neighbor_pixel, d+1] + small_penalty,
                              min( pixel_cost[neighbor_pixel, i] ) + large_penalty)
```

***Code Block 1 –*** Pseudo-code to calculate the smoothed disparity costs using the SGM method.

## 4.4 Patch-Match (PM) Algorithm

The PM method works through establishing correspondence through refined random searches. We first create a disparity image whose dimensions are the same as the images from the left/ right camera. The disparity image is seeded with uniform noise in the range $[0, D]$. The disparity image is then smoothed against each neighbor, similar to the SGM method. In the next iteration, the process is repeated by adding more noise to the disparity image, but at half the previous amplitude, i.e. at $[0, \frac{D}{2}]$. The iterations are repeated until convergence is met [10]. The pseudo-code for calculating the smoothed costs using the PM algorithm is given in Code Block 2.

```
disparity_image = uniform_noise(sizeof(left_image), disparity_range)

while not converged:
        for each pixel in disparity_image:
                calculate pixel_cost(disparity_image(d))

        for each pixel in disparity_image:
                for each neighbor in scanning_direction:
                        calculate smoothed_pixel_cost(current_pixel, neighbor_pixel)

        for each pixel in disparity_image:
                add uniform noise(disparity_range/2)

def pixel_cost(d):
        pixel_cost[d] = left_image(y, z) – right_image(y-d, z)

def smoothed_pixel_cost(current_pixel, neighbor_pixel):
        d1 = disparity at current_pixel
        d2 = disparity at neighbor_pixel
        c1 = cost at d1
        c2 = cost at d2
        if (c2 < c1):
                disparity at current pixel = d2
```

***Code Block 2 –*** Pseudo code to calculate the smoothed disparity costs using the PM method.

The dense disparity images are then used to calculate the depth through inverting Equation 3. The depth information is then used to calculate geometric optical flow as described in Section 4.1. Although not wholly accurate, the dense methods provide more information for optical flow subtraction than the sparse methods.

# 5 CAMERA SIMULATION IN UNREAL ENGINE

## 5.1 Basic Camera

UE provides in-built components for "Player Cameras". However, the player cameras, only act as a viewport to view the virtual world during simulation, and are unable to capture/ record images for computer vision processing. Furthermore, it is difficult to simulate multiple cameras required for stereo-vision with this tool. To address this, a tool called "Texture Render" was used. Texture renders are used to create in-game recording components, like simulated security cameras. For purposes of the project, each camera was modeled as a texture render. The images read by the UE are converted to OpenCV image formats for further processing. Figure 12 shows an example of an image frame captured in UE through this process.

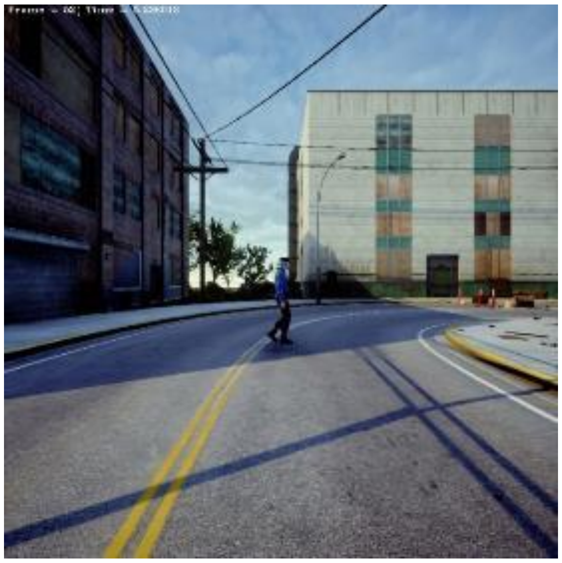


***Figure 12 –*** Sample image captured in UE through a texture render.

Texture renders are abstract components in UE, and do not behave as rigid bodies. In order to be able to control them through physical laws/ keyboard controls, it was necessary to attach them to a "Root Component" which is able behave like a rigid body.

## 5.2 Stereo Camera

Stereo-cameras are simply two basic cameras (or texture renders) attached to the same root component. In the model, the cameras are displaced from each other by 12cm. The rigid root is attached to the center-point of the two cameras, as described in Figure 13.

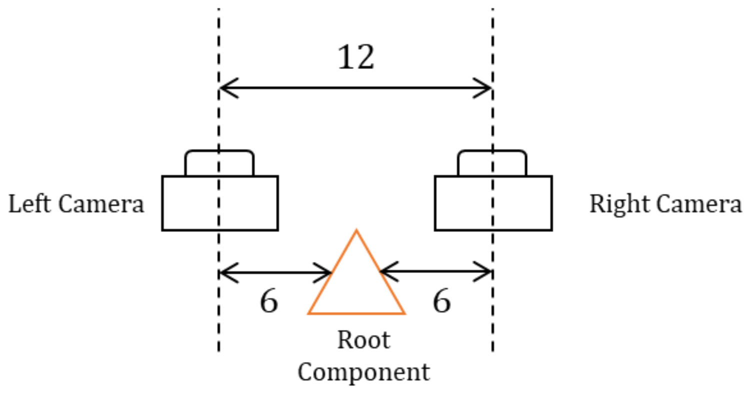


***Figure 13 –*** A visualization of how two virtual cameras are attached to a root component in UE. Dimensions are in cm.

By attaching the cameras to a single root (as opposed to two individual roots), we are able to ensure that both cameras move similar to a rigid body during translational and rotational motion. Figures 14 show the frames captured by either camera when the root is rotating in place.

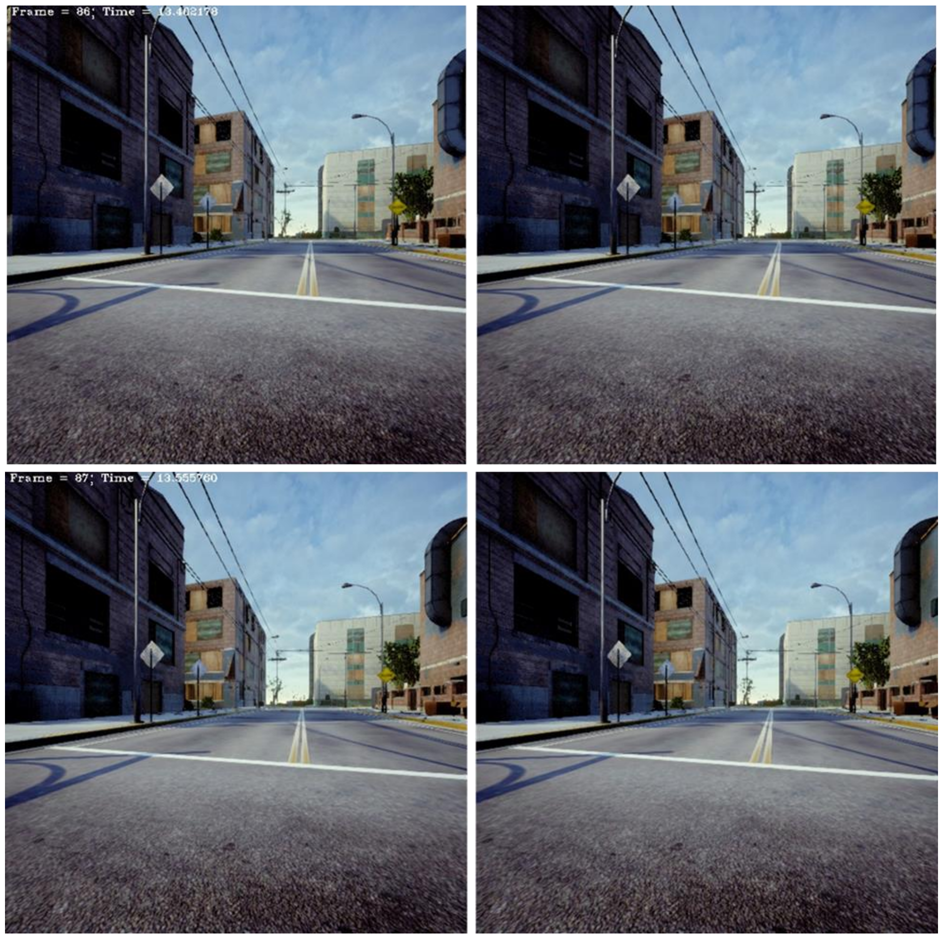


***Figure 14 –*** Frames from the left and right cameras when a stereo camera when it is rotating in place.

## 5.3 Stereo Vision Implementation

Two classes (one for SGM, and one for PM) were implemented with standard interfaces for calculating depth information. The source files can be included with any project (online or offline), and can be used as follows:

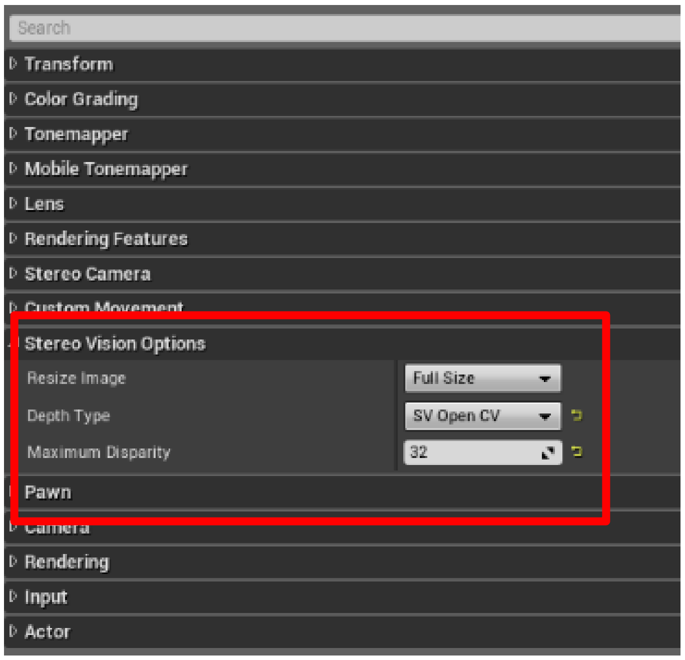
```
/* Semi Global Matching */
SemiGlobalMatching sgm( initialization_parameters )
sgm.set(left_frame, right_frame)
sgm.process()
sgm_disparity = sgm.get_disparity()

/* Patch Match */
PatchMatch pm( initialization_parameters )
pm.set(left_frame, right_frame)
pm.process()
pm_disparity = pm.get_disparity()

/* Getting Depth Information */
sgm_depth = getDepthFromDisparity(sgm_disparity)
pm_depth = getDepthFromDisparity(pm_distparity)
```

***Code Block 3 –*** Standardized interfaces for calculating depth information through the SGM or PM methods.

In order to enable researchers easily switch between the different methods, without having to modify and recompile the project code, the stereo camera was programmed to be editable in the graphical interface of UE. The different methods of calculation can be chosen from a drop-down menu between simulation runs, as shown in Figure 15. In addition to SGM and PM, users will also be able to choose the option of using the depth calculation method from OpenCV, as well as the ground truth information provided by UE. The user will also be able to set other parameters, such as the maximum expected disparity, and the option of scaling down the image for improved computational speed.



Resize Image: Full Size
Depth Type: SV Open CV
Maximum Disparity: 32

***Figure 15 –*** User interface in UE where users will be able to choose between the different modes of depth calculation.

## 5.4 Optical Flow Implementation

The OpenCV toolbox has readily available functions for calculating observed optical flow using the Farneback or Lucas-Kanade methods. However, the function call signatures, and the return parameters for either method are significantly different. The `cv::calcOpticalFlowFarneback()` function takes in two video frames, and a set of numerical parameters defining the smoothing settings used for calculating pixel flows, and returns a dense image matrix containing the gradient at each pixel location. Conversely, the `cv::calcOpticalFlowLucasKanade()` method requires the user to pass in a set of salient features (calculated through feature detection algorithms) for tracking in addition to the two video frames. The return argument is a sparse vector of pixel velocities at the locations of the salient features.

A function to calculate the geometric optical flow is not available in OpenCV, and was implemented from scratch. Given two consecutive video frames, the camera's body velocities, and a dense image matrix containing depth information, the function will return the expected flow at each pixel location.

For each optical flow method (Farneback, Lucas-Kanade, and Geometric), the input and output arguments are different. In order to standardize the interface for easily switching between implementations, a library of optical flow functions was implemented. The library allows the user to calculate optical flow using any method through a single function call, and an appropriate flag. The output arguments are also of an identical format for all three methods.

```
/* previous method of calculating functions */
// Farneback
dense_flow = calcOpticalFlowFarneback(frame1, frame2)
// Lucas Kanade
salient_features = goodFeaturesToTrack(frame1)
sparse_flow = calcOpticalFlowPyrLK(frame1, frame2, salient_features)
// Geometric – custom function
dense_flow = calcOpticalFlowGeometric(frame1, depth_frame, kinematic_data)

/* standardized function library */
// Intensity based methods – frames 1 and 2 are both rgb frames
flow = getOpticalFlow(FARNEBACK, frame1, frame2)
flow = getOpticalFlow(LUCAS_KANADE, frame1, frame2)
// Geometric method – frames 1 is rgb, and frame 2 is depth information
flow = getOpticalFlow(GEOMETRIC, frame1, frame2, kinematic_data)

/* optical flow subtraction */
subtracted_farne = subtractOpticalFlow(FARNEBACK, intensity_flow, geometric_flow)
subtracted_LK = subtractOpticalFlow(LUCAS_KANADE, intensity_flow, geometric_flow)

/* plotting optical flow */
plotOpticalFlow(FARNEBACK, intensity_flow)
plotOpticalFlow(GEOMETRIC, geometric_flow)
plotOpticalFlow(LUCAS_KANADE, subtracted_LK)
```

***Code Block 4 –*** Comparison of function calls before and after the implementation of the customized library, as well as the supplementary functions provided for subtracting and plotting optical flow with ease.

Optical flow subtraction is also performed differently for the Farneback and Lucas-Kanade methods, simply due to the fact that the former is dense, and the latter is sparse. For Farneback, subtraction is performed at every pixel. This is simply a process of iterating over every pixel in the image and subtracting the geometric from the total observed flow. For Lucas-Kanade, subtraction is performed only at the locations of salient features. In addition to standardizing the optical flow function calls, the library also provides a set of functions for performing subtraction with a single function call. Finally, a function to easily plot the flow vectors on the frame is also provided. The plotting function works on any type of optical flow, geometric, intensity-based, and even subtracted flows. This library can be imported directly into UE and used during simulation with minimal modification.

# 6 Results and Discussion

## 6.1 Ground Truth Depth Information

UE has a utility to obtain the ground truth in depth information as seen by the camera in the virtual world. For optical flow subtraction, this would provide the most accurate model for inferring depth of the scene. Figure 16 shows an example of the ground truth depth information. The depth information is obtained using the left camera as reference. During online computation, the depth information can be calculated in 16- or 32- bit formats. However, for visualization and storing, they are converted to 8-bit formats. In the image, lighter pixels represent points further from the camera, and darker pixels represent points closer to the camera. All pixels further than 5000cm will be white. The maximum distance "seen" by the camera can be tuned by the user.

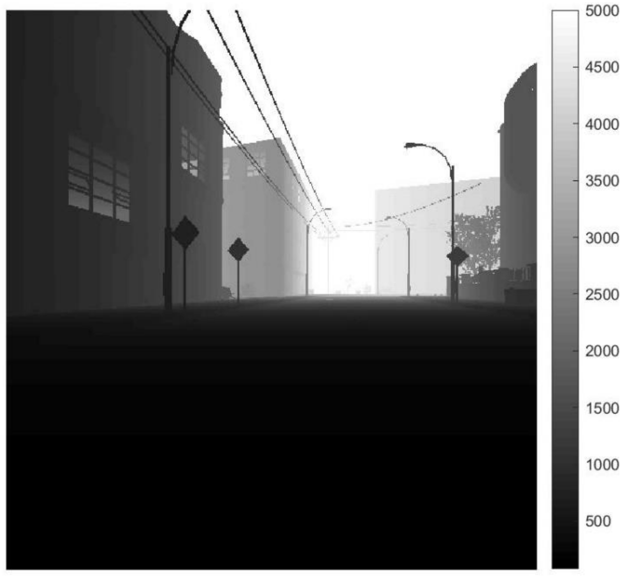


***Figure 16* –** Ground truth depth information obtained from UE. Lighter pixels represent points that are further away, and darker pixels represent points that are closer to the camera. All points further than 5000cm will be shown as white.

## 6.2 OpenCV Depth Calculation

The OpenCV toolbox provides a simplified disparity calculation algorithm. However, it has some loss of information (the disparity image loses information at the sides of the frames), and is often unable to handle larger disparity ranges. The average root mean square error computed for 250 frames at a maximum expected disparity value of 64, was 35.6%.

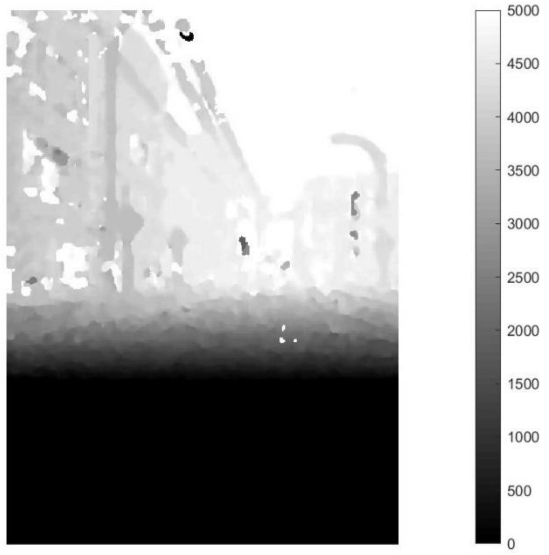


***Figure 17 –*** The OpenCV toolbox's implementation of disparity calculation. The left image shows the scene from Figure 16 calculated with a maximum expected disparity value of 64 pixels. Information at the edges of the image are lost during compression performed by the OpenCV tool.

## 6.3 SGM Depth Calculation

The SGM method was able to perform better than the OpenCV toolbox during offline computation. Figure 18 shows an example of the SGM algorithm with a normalized RMSE mean of 19.5% for 250 frames.

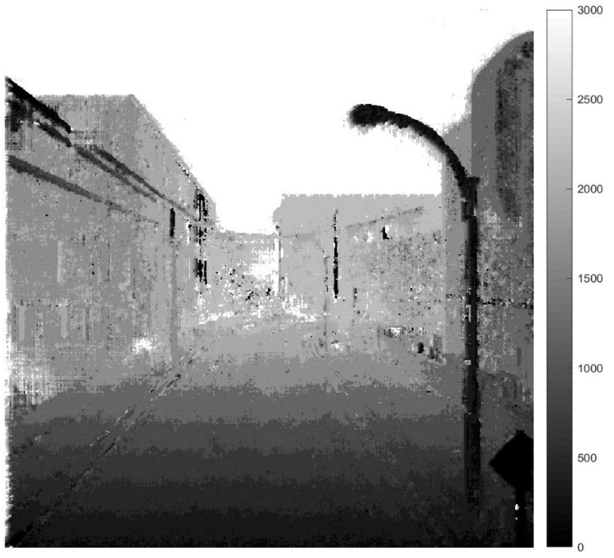


***Figure 18 –*** Offline performance of the SGM algorithm with optimal convergence. The normalized RMSE was 19.5%.

However, when the number of iterations was decreased to improve the speed of online computation, the performance of the SGM method deteriorated, as shown in Figure 19. With decreased iterations, the average normalized RMSE for the SGM method was 29.1%.

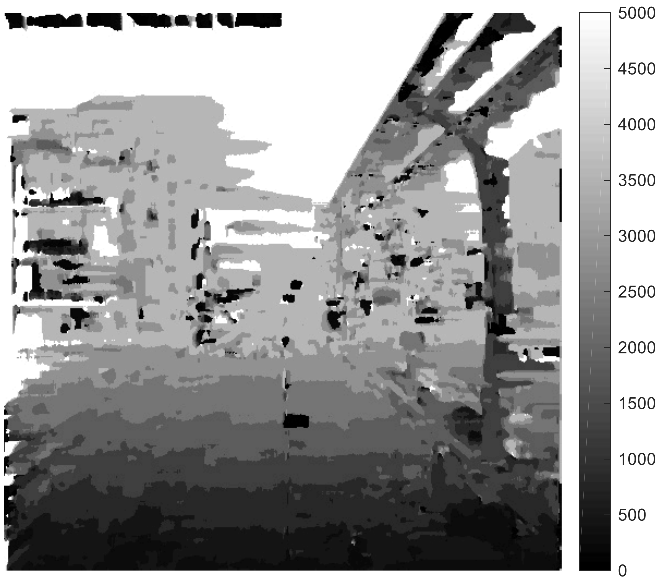


***Figure 19 -*** Online SGM with sub-optimal convergence (to reduce computation time). The average RMSE for online SGM was 29.1%.

One major disadvantage of the SGM method is that the computation speed increases linearly with the maximum expected disparity. Since the method the algorithm works by searching every possible disparity value in the range $d \in [0, D]$, increasing $D$ increases the time required for computation.

## 6.4 PM Depth Calculation

The PM method is slower than the SGM method during online computation. However, it is able to maintain constant time with respect to the maximum expected disparity. Since the algorithm works through random search, it does not analyze every possible value in the range $d \in [0, D]$. The number of iterations is user-defined, and therefore remains constant regardless of the size of the search range. The average error during runtime was similar to that of the SGM method, at 29.2%.

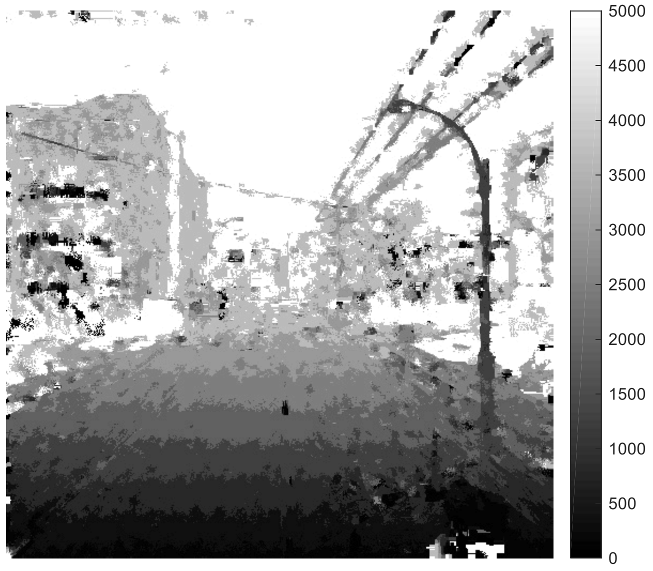


***Figure 20 –*** Disparity from PM algorithm after 1 iteration. The average RMSE was 29.2%

## 6.5 Observed Optical Flow

The optical flow library was able to calculate sparse and dense optical flows as expected. Figure 21 shows two samples of the sparse observed flow when the camera is in rotation (left) and in translation (right). Figure 22 shows the same two frames analyzed with the dense optical flow method.

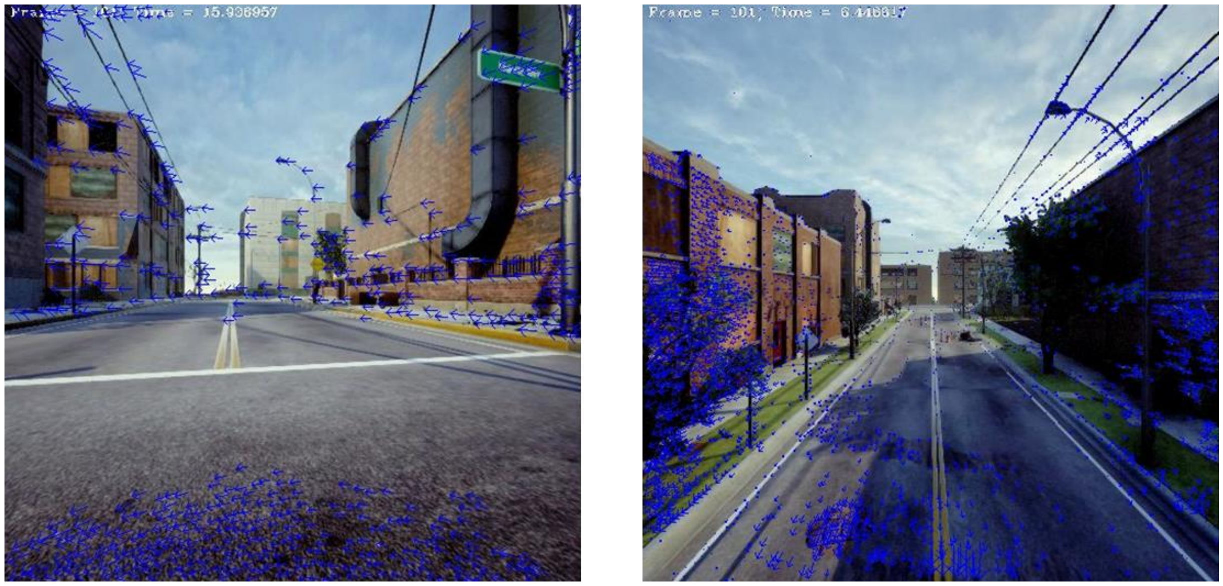


***Figure 21 –*** Sample frames from calculating sparse (Lucas-Kanade) optical flow for camera rotation (left) and translation (right) in stationary scenes.

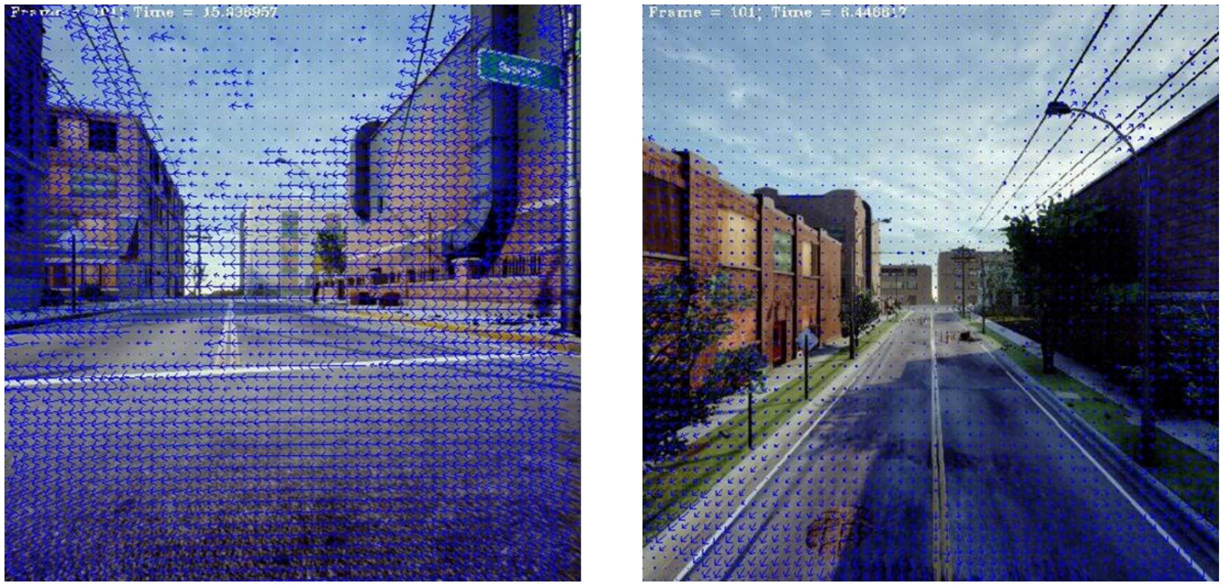


***Figure 22 –*** Sample frames from calculating the dense Farneback optical flow for camera rotation (left) and translation (right) in stationary scenes.

## 6.6 Geometric Optical Flow

The geometric optical flow algorithm derived in Section 4.1 was also able to estimate pixel velocities based on kinematic data from UE. Figure 23 shows the geometric optical flow calculated from a rotating (left) and a translating (right) camera. The translation aspect of the calculation still suffers from inaccuracies, particularly in regions of low texture. The rotation component is relatively less affected as it does not directly depend on depth information.

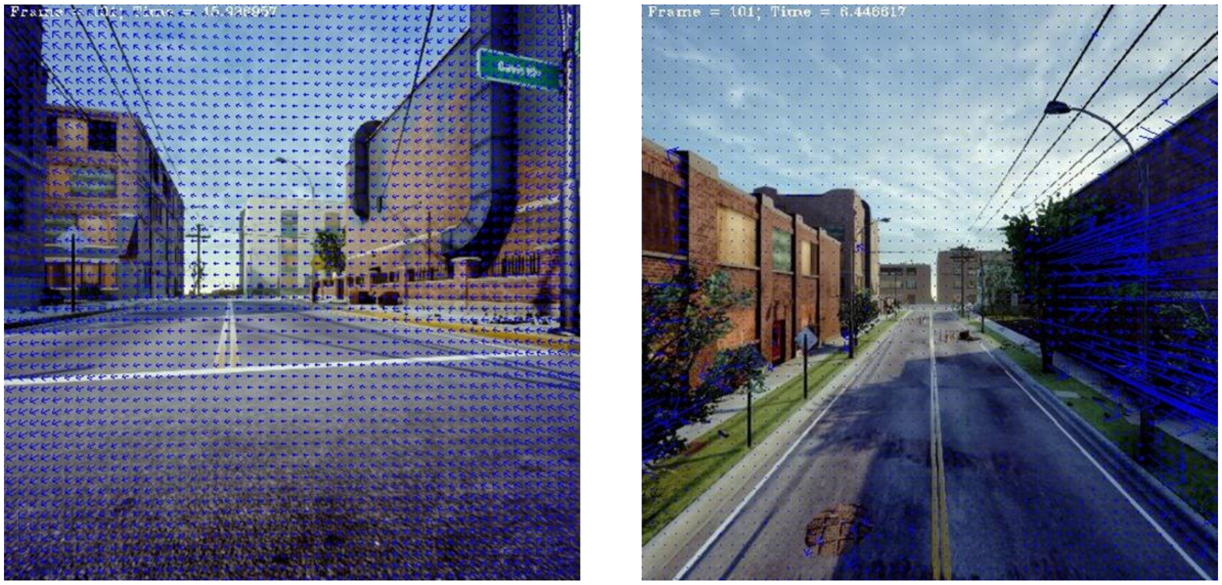


***Figure 23 -*** Geometric Optical Flow for rotating (left) and translating (right) camera motion in stationary scenes.

## 6.7 Optical Flow Subtraction

As of the time of writing this report, the primary challenge to optical flow subtraction is the lack of camera parameters on UE. The camera's focal length is a key parameter in depth calculation as well as in the calculation of geometric optical flow. However, in UE, due to the "virtual" state of rendering a camera, it does not share many properties with physical cameras, including the focal length. As a result, the flow subtraction is still an approximation at best, and still suffers from inaccuracies. Nonetheless, the subtraction program is able to produce detect a moving object (Figure 23).

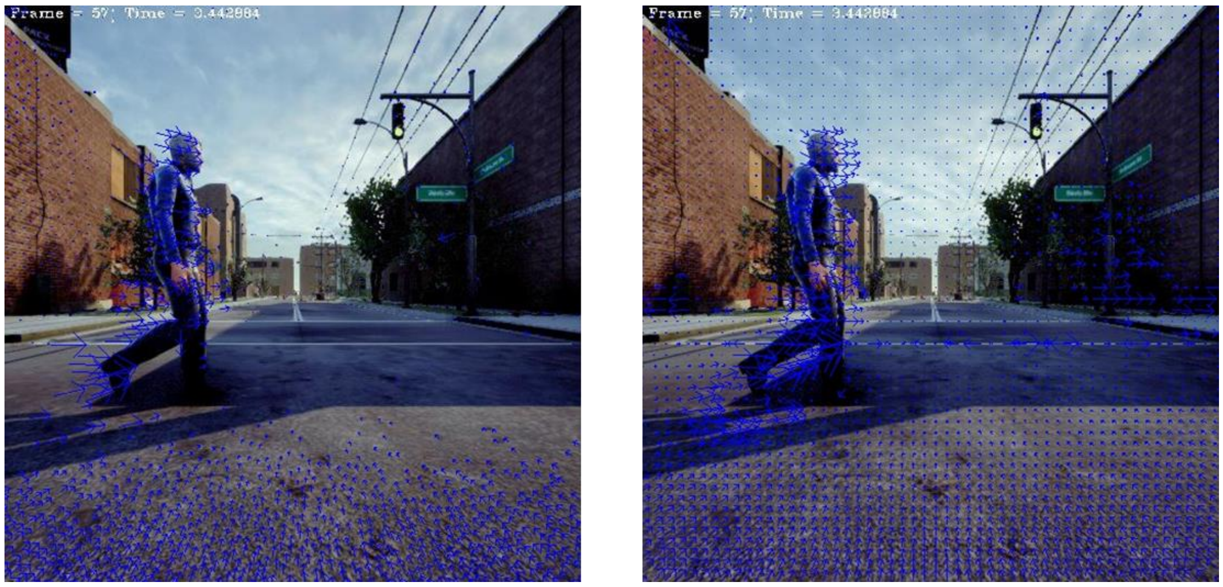


***Figure 24 –*** Optical flow subtraction performed with the sparse Lucas-Kanade method (left image), and with the dense Farneback method (right image).

In the video sequence from which Figure 23 was obtained, the virtual man is walking across the scene of the camera, while the camera itself is translating backwards. Larger flow vectors are observed around the moving person, while smaller flow vectors (indicating subtraction) are observed in the background pixels.

## 7 Conclusion and Summary

Overall, the goal of the project was to implement toolboxes that make it easier for researchers and developers to use typical algorithms without having to repeatedly implement common functions. I was able to implement a set of standard functions for depth calculation, observed optical flow calculation, as well as geometric optical flow calculation and subtraction, which were able to perform as expected.

## 8 Acknowledgements

I would like to thank Prof Silvia Ferrari for providing me with the opportunity to work on this exciting project with her team at LISC. I would also like to thank Jake Gemerek for his extended guidance and support throughout the course of the project.

## REFERENCES

[1] J. R. Gemerek, S. Ferrari, M. Campbell and B. Wang., "Video-guided Camera Control and Target Tracking using Dense Optical Flow," Cornell University, Ithaca, 2017.

[2] Epic Games, "What is Unreal Engine 4," 2018, [Online]. Available: www.unrealengine.com/. [Accessed 10 12 2018].

[3] K. Hata and S. Silvio, Course Notes 1: Camera Models, Stanford University.

[4] D. Hoiem, Projective Geometry and Camera Models, Univeristy of Illinois, 2011.

[5] J. Gemerek, "Optical Flow Ego-motion Subtraction," Cornell Univerisity, Ithaca, New York, 2018.

[6] G. Farneback, "Two-Frame Motion Estimation Based on Polynomial Expansion," Linkoping Univeristy, Linkoping, Sweden, 2003.

[7] J.-Y. Bouguet, "Pyramidal Implementation of the Lucas Kanade Feature Tracker - Description of the Algorithm," Intel Corporation, 2000.

[8] S. Birchfield and C. Tomasi, "Depth Discontinuities by Pixel-to-Pixel Stereo," Stanford University, Stanford, California, 1997.

[9] H. Hirschmuller, "Stereo Processing by Semiglobal Matching and Mutual Information," IEEE Transactions on Pattern Analysis and Machine Itelligence, vol. 30, no. 2, pp. 328-341, 2008.

[10] M. Bleyer, C. Rhemann and C. Rother, "PatchMatch Stereo - Stereo Matching with Slanted Support Windows," Microsoft, Cambridge, UK, 2011.

[11] B. J. L and T. N. A, "Tutorial: Computing 2D and 3D Optical Flow," University of Manchester, Manchester, UK, 2005.

[12] J. Barron, "The 2D/3D Differential Optical Flow," in Canadian Conference on Computer and Robot Vision, Kelowna, British Columbia, 2009.