9-5-2024

# Study of the Application of Blender for Simulation of a Closed-loop Image-based Greenhouse Supplemental Lighting Control

Kip Nieman
*Department of Chemical Engineering and Materials Science, Wayne State University, Detroit, MI*,
kip.nieman@wayne.edu

Helen Durand
*Department of Chemical Engineering and Materials Science, Wayne State University, Detroit, MI*,
helen.durand@wayne.edu



## Recommended Citation

Nieman, K.; Durand, H. Study of the Application of Blender for Simulation of a Closed-loop Image-based Greenhouse Supplemental Lighting Control. IFAC Papers OnLine 58(14), 2024, 519-524. https://doi.org/10.1016/j.ifacol.2024.08.389

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnline CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 58-14 (2024) 519–524

# Study of the Application of Blender for Simulation of a Closed-loop Image-based Greenhouse Supplemental Lighting Control

**Kip Nieman** * **Helen Durand** **

* *Wayne State University, 42 W. Warren Ave. Detroit, MI 48202, USA (e-mail: kip.nieman@wayne.edu).*

** *Wayne State University, 42 W. Warren Ave. Detroit, MI 48202, USA (e-mail: helen.durand@wayne.edu)*

**Abstract:** Image-based control and sensing has been applied in a wide variety of next generation manufacturing fields. Utilizing methods of simulating closed-loop image-based control may be advantageous for improving control performance and design without the need for an experimental setup. One software capable of these simulations is the open-source 3D modeling software Blender, which has many capabilities aided by a Python API. This work explores the use of Blender as an image-based control test bed, where both the process and the controller are simulated, in the context of a greenhouse supplemental lighting control system.



*Keywords:* Simulation tools, Greenhouse control, Model predictive and optimization-based control, Digital twin development, Modeling, Blender, Ray tracing, 3D imaging

## 1. INTRODUCTION

Next-generation manufacturing strategies can take advantage of new sensing technologies, such as image-based sensors and models based on image data, to improve process operation. Image-based sensing and control has been important in applications in a variety of fields, including in robotics (Mezouar and Chaumette (2002)), automotive (Trivedi et al. (2007)), aerospace (Scorsoglio et al. (2022)), and process systems (Yan et al. (2014)) applications. In these contexts, sensors and image-based controllers are studied to allow for improved process performance on real systems. Also desirable are methods of simulating closed-loop test beds of image-based control, where both the controller and system are simulated together. Simulations like this have been applied in many fields, such as, for example, autonomous vehicles (Aradi (2020)).

One potential software for test beds of closed-loop image-based controllers is Blender, which is a open-source 3D computer graphics software with several useful features. These include that the code is maintained and easy to download, a user interface for creation and modification of a virtual 3D geometry, and the inclusion of a Python application programming interface (API) for Python coding and interfacing Blender geometries. The use of Blender in the process systems engineering field for closed-loop image-based control has been studied in several contexts. This includes investigating image-based proportional-integral (PI) control of the level in a tank (Oyama et al. (2022a)), a study of cyberattacks on image-based control systems (Oyama et al. (2022b)), and a path-finding study for use of image-based control of an actuated nanorod as a precursor to image-based control of self-assembly (Leonard et al. (2024)). Such simulations may help in probing the effects of modeling and control strategies and investigating the use of different computing platforms on control operation.

This work seeks to demonstrate and expand upon the capabilities of Blender with the Python API as a test-bed for closed-loop image-based control through the study of a greenhouse supplemental lighting control system. The overall problem involves the study of a controller that determines the optimal powers of individual light-emitting diodes (LEDs), considering sunlight, for a greenhouse that considers a virtual representation of the process. To represent the virtual geometry, individual plant leaves are represented using the Blender image creation tools. Following this, the Python API is used to gather point cloud information (representing a 3D imaging method), create a mesh from the point cloud using a triangulation algorithm, perform ray tracing to model energy absorbed by each leaf from each LED and the sun, represent growth of the plant geometry, and solve an optimization problem that minimizes energy usage and maximizes plant growth by selecting optimal LED powers. The purpose of these coding objectives is to demonstrate the broad capabilities of Blender to capture a variety of dynamics, and to demonstrate how Blender could be used to explore novel control objectives and strategies. We begin by detailing the Blender and Python API test bed, explaining an overview of the workflow. This includes descriptions of the creation of a sample geometry, the point cloud data collection and triangulation algorithms, the optimization problem formulation, the ray tracing model, and a plant growth strategy. Following this, we discuss the selection of simulation parameters for the ray tracing algorithm, where we determine the minimum number of vectors for independent results (ensuring enough vectors are simulated so that the results do not depend on the number of vectors). Finally,


10.1016/j.ifacol.2024.08.389

simulation results from the Blender test bed are presented and discussed.

## 2. SIMULATION METHODS

The greenhouse lighting control test bed was created in Blender and utilizes Python coding. An overview of the objectives of the code are as follows:

(1) A mock plant geometry was created with the Blender interface.
(2) A Python code reads node data from the geometry to create the kind of data that might be gathered from a 3D imaging technique (e.g., a point cloud).
(3) A Python triangulation code converts the point cloud data into a triangular mesh consisting of a series of connected triangles.
(4) A Python code solves an optimization problem to determine the optimal LED powers based on an objective function, using a simplified ray tracing model (denoted as the 'reduced' ray tracing model) to estimate the energy absorbed and growth of the plant.
(5) Using the LED values from the optimization problem, a triangulated mesh of the actual geometry, and a full version of the ray tracing algorithm (the 'process' model), the geometry of the process is updated.

The following sections explain these aspects in more detail.

### *2.1 Plant Geometry, Point Cloud Data, and Triangulation*

An arbitrary plant was manually created using the Blender interface (though the creation of plant geometries can be automated using the Python API). In this work, the geometry was created with seven leaves, each created using sections of ico-spheres that were stretched and manipulated into leaf-like shapes (see the left image in Fig. 1 for a rendering of the geometry from Blender). The result was a geometry with 171 vertices in total. The geometry was also designed to fit inside a cube representing the walls of the greenhouse, with dimensions (in meters) of $4 \times 4 \times 4$ (within coordinates $-2 \leq x \leq 2$, $-2 \leq y \leq 2$, $0 \leq z \leq 4$). A wireframe representing the plant geometry is shown in the middle image of Fig. 1.

Next, a section of the code was designed to write the vertex coordinates from the geometry in Blender to a text file. The coordinates are taken directly from the Blender geometry by first selecting each leaf using

```
leafObj = bpy.data.objects["leaf1"]
```

in Python, which allows for each leaf to be stored as a variable (in this case, leaf1 is now represented using leafObj). Then it is possible to iterate through the vertices by creating a "for" loop starting with

```
for local in leafObj.data.vertices:
```

and print the coordinates to a text file. Note that it is necessary to convert the coordinates to the global reference frame before printing to a text file, using the command

```
global = leafObj.matrix_world @ local.co
```

where 'global' contains the global coordinates within the "for" loop. By including an additional command to print

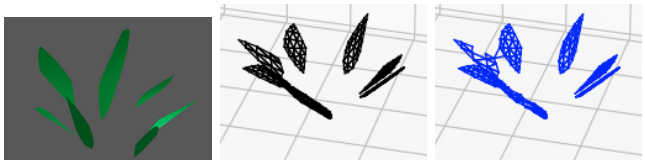

Fig. 1. LEFT: Plant with seven leaves rendered in Blender. MIDDLE: Geometry of the actual leaf, represented as a triangular mesh. RIGHT: The triangulated geometry created using the strategy involving the 'Delaunay' function.

to a text file, this process results in a text file. In this work, the text file was structured so that each line contains the $(x, y, z)$ coordinates of every vertex in the plant, without any information representing the connectivity. For simplicity, this is treated as a representation of a point cloud. While this method is not a model of a 3D imaging method (we are merely reading vertices directly from the Blender geometry), the flexibility of the Python API means it would be possible to implement many different kinds of simulations of 3D imaging hardware and software.

Following this, a section of the code determines the connectivity of the vertex data, connecting nearby points to create a triangular mesh. This is accomplished by projecting each vertex on the xy-plane and then using the 'Delaunay' function from the Python Scipy package, which is designed to create a connected triangular 2D mesh from point data. The resulting connected vertices are then reunited with the z-coordinates to create the 3D mesh. After this, two filtering functions were written that iterate though every triangle and remove certain triangles that exceed certain properties. These include triangles that contain an edge length longer than $L_{max}$ and triangles with a height less than $H_{min}$. This is necessary because, without removing these triangles, all leaves become fused together (with oddly shaped triangular elements connecting adjacent leaves together, in this case making the geometry bowl-shaped). The results of this process can be seen in the right image of Fig. 1. Note that triangles were additionally included to represent the walls, floor, and ceiling of the growth area (though these are not displayed in Fig. 1). The resulting plant mesh is not identical to the 'actual' geometry and contains some errors introduced by the filtering functions failing to remove all of the unwanted elements without error. Overall, even though we have not simulated 3D imaging, the differences in the mesh and the actual geometry would also occur in a 3D imaging method (given the general nature of the Python API, 3D imaging could be implemented as models in Blender).

### *2.2 Optimization Problem Formulation*

The controller formulation is implemented in the Python API and solves the following optimization problem:

$$\min_{x_i,\ i=1,2,\ldots,n_{\text{LED}}} \left[ C_1 \sum_{i=1}^{n_{\text{LED}}} x_i - C_2 \frac{\sum_{j=1}^{n_{\text{tri}}} p_j}{\sum_{j=1}^{n_{\text{tri}}} a_j} \right]$$

$$s.t. \quad \sum_{j=1}^{n_{\text{tri}}} p_j \geq E_{\min}$$

$$0 \leq x_i \leq x_{\max},\ i = 1, 2, \ldots, n_{\text{LED}}$$

where $x_i, i = 1, 2, ..., n_{\text{LED}}$ are decision variables of the optimization problem representing the power of each of the $n_{\text{LED}}$ LEDs (the $x_i$ values are also the manipulated inputs computer by the controller). The objective function contains two terms, each preceded by constants, $C_1$ and $C_2$, which are used to specify the relative importance of each term. The first term in the objective function $\sum_{i=1}^{n_{\text{LED}}} x_i$ represents the total amount of energy used by the supplemental lighting system, which is minimized. The second term $\frac{\sum_{j=1}^{n_{\text{tri}}} p_j}{\sum_{j=1}^{n_{\text{tri}}} a_j}$ represents the overall energy absorbed per unit area of the plant, which is used as a metric to estimate plant growth in the optimization problem, and is subtracted so it is maximized. The absorbed energy of each of the $n_{\text{tri}}$ triangles is represented as $p_j, j = 1, 2, ..., n_{\text{tri}}$, and the area of each triangle is $a_j, j = 1, 2, ..., n_{\text{tri}}$. The value of $\sum_{j=1}^{n_{\text{tri}}} p_j$, which is found using the ray tracing algorithm, is representative of the total energy absorbed by the plant, and $E_{\min}$ represents a desired minimum amount. Finally, $x_{\max}$ represents the maximum possible power applied to each LED. To determine the solution of this optimization problem, the 'minimize' function from the Python Scipy package was used with the 'SLSQP' (Sequential Least Squares Programming) algorithm. Finally, no specific units of energy are specified in this work. Instead, the value given to energy is relative to $x_{\max}$ (so if $x_{\max} = 1$ and the plant absorbs an energy of, for example, 2.5, this indicates that the amount of energy absorbed by the plant is 2.5 times the amount of energy emitted by an LED).

### *2.3 Ray Tracing*

The purpose of this section is to describe the mathematics behind the ray tracing simulation, which was implemented using the Blender Python API. This implementation demonstrates one possible mathematical framework that can be integrated into a Python simulation, though given the freedom introduced by the Python API, just about any mathematics can be implemented. The ray tracing simulation in this work assumes that a number of point light sources exist that each emit light with some radius. The following discusses the vector mathematics involved in this ray tracing simulation.

Each vector has a starting point $A = (x_0, y_0, z_0)$ and an end point of $B = (x_f, y_f, z_f)$. The vector is defined as $\overrightarrow{AB} = [x_f - x_0 \;\; y_f - y_0 \;\; z_f - z_0]^T$, the vector magnitude is $||\overrightarrow{AB}|| = \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2 + (z_f - z_0)^2}$, and the unit vector in the direction of $\overrightarrow{AB}$ is $\hat{\overrightarrow{AB}} = \frac{\overrightarrow{AB}}{||\overrightarrow{AB}||}$. The dot product of two vectors $\overrightarrow{A} = [a_1 \;\; a_2 \;\; a_3]^T$ and $\overrightarrow{B} = [b_1 \;\; b_2 \;\; b_3]^T$ is $\overrightarrow{A} \cdot \overrightarrow{B} = a_1b_1 + a_2b_2 + a_3b_3 = ||\overrightarrow{A}||\,||\overrightarrow{B}|| \cos\theta$ and the cross product is $\overrightarrow{A} \times \overrightarrow{B} = [a_2b_3 - a_3b_2 \quad a_3b_1 - a_1b_3 \quad a_1b_2 - a_2b_1]^T = ||\overrightarrow{A}||\,||\overrightarrow{B}|| \sin\theta$. To represent a vector in three-dimensional space, the parametric equation of a line is applied:

$$x = x_L + \alpha t, \quad y = y_L + \beta t, \quad z = z_L + \gamma t \tag{1}$$

where $(x_L, y_L, z_L)$ is a point on the line, $[\alpha \;\; \beta \;\; \gamma]^T$ is the direction of the line, and $t$ is the parameter. Each triangle lies in a two-dimensional plane that can be described by the following equation:

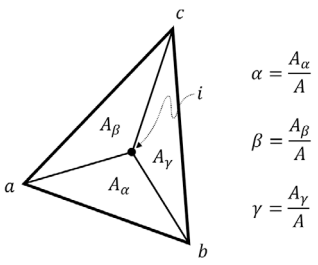


Fig. 2. Diagram of determining if a vector intersects a plane inside of a triangle, utilizing three subtriangles $A_\alpha$, $A_\beta$, and $A_\gamma$.

$$n_1x + n_2y + n_3z + k = 0 \tag{2}$$

where $[n_1 \;\; n_2 \;\; n_3]^T$ is the direction of a vector normal to the plane, $k = -(n_1x_P + n_2y_P + n_3z_P)$ is a constant, and $(x_P, y_P, z_P)$ is a point on the plane. For each triangle, the normal vector can be found by taking the cross product of any two edges. For example, if the three vertices of a triangle are $A$, $B$, and $C$, the normal vector can be found as $\overrightarrow{N} = \overrightarrow{AB} \times \overrightarrow{AC}$. Additionally, the area of a triangle can be found as Area $= \frac{1}{2}||\overrightarrow{N}||$.

Combining Eq. 1 and Eq. 2, the following equation for the parameter $t$ can be found, which represents the value of the parameter when the vector defined by Eq. 1 intersects the plane given in Eq. 2:

$$t = -\frac{[n_1 \;\; n_2 \;\; n_3]^T \cdot [x_L \;\; y_L \;\; z_L]^T + k}{[n_1 \;\; n_2 \;\; n_3]^T \cdot [\alpha \;\; \beta \;\; \gamma]^T} \tag{3}$$

where $[n_1 \;\; n_2 \;\; n_3]^T$ is the direction of a normal vector to the plane and $(x_L, y_L, z_L)$ is a point on the line that may intersect the plane. If the vector reaches the specified plane, the value of $t$ given by Eq. 3 will be greater than zero. Using Eq. 3, the point where the vector intersects each plane (if it exists) can be found by substituting the $t$ value into Eq. 1.

Next, it is necessary to determine if the vector intersects the plane within the triangle. One algorithm that accomplishes this (see Shirley et al. (2009)) involves dividing a triangle into three subtriangles as shown in Fig. 2 (which shows the case where the intersection lies within the triangle). In Fig. 2, the triangle vertices are denoted as $a$, $b$, and $c$, and the intersection point is denoted as $i$. The areas of the subtriangles, $\triangle abi$, $\triangle aci$, and $\triangle bci$, are denoted as $A_\alpha$, $A_\beta$, and $A_\gamma$ respectively. Then the ratio of each subtriangle to the area of $\triangle abc$ is found as $\alpha = \frac{A_\alpha}{A}$, $\beta = \frac{A_\beta}{A}$, and $\gamma = \frac{A_\gamma}{A}$, where $A$ is the area of $\triangle abc$. For the point $i$ to lie within $\triangle abc$, the expressions in Eq. 4 must hold:

$$\alpha, \beta, \gamma \in [0, 1], \quad \alpha + \beta + \gamma = 1 \tag{4}$$

Following this, the next step is to calculate the direction of the reflected vector. To accomplish this, first define the incident vector as $\overrightarrow{I}$, the unit normal vector to the plant as $\hat{\overrightarrow{N}}$, and the reflected vector as $\overrightarrow{R}$. Then the direction of the reflected vector is:

$$\overrightarrow{R} = \overrightarrow{I} - 2(\overrightarrow{I} \cdot \hat{\overrightarrow{N}})\hat{\overrightarrow{N}} \tag{5}$$

Fig. 3 demonstrates how Eq. 5 is derived. First, as shown in the top-right image in Fig. 3, the projection of $\overrightarrow{I}$ onto

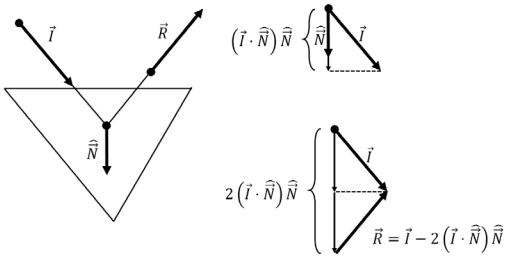


Fig. 3. Diagram representing the process to find the direction of the reflected vector $\vec{R}$. The projection of $\vec{I}$ onto $\hat{\vec{N}}$ is doubled, and then vector subtraction is used to find $\vec{R} = \vec{I} - 2(\vec{I} \cdot \hat{\vec{N}})\hat{\vec{N}}$ (Dorst et al. (2009)).

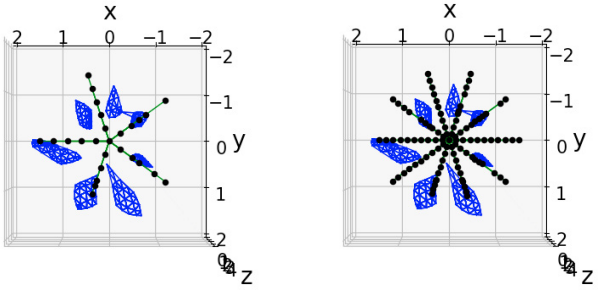


Fig. 4. Display of vectors emanating from one LED, displayed with the camera positioned directly above the plant and rotated so that the view is facing the ground. LEFT: $n_r = 5$, $n_\theta = 5$ RIGHT: $n_r = 10$, $n_\theta = 10$

$\hat{\vec{N}}$ is found as $(\vec{I} \cdot \hat{\vec{N}})\hat{\vec{N}}$. The bottom-right image in Fig. 3 shows how, if the projection is doubled and the result is subtracted from $\vec{I}$, the result is the equation of the reflected vector as shown in Eq. 5.

Each LED emits a number of vectors defined by a number of radial divisions $n_r$ (up to a specified radius of $r_{LED}$, which is defined on the $z = 0$ plane) and angular divisions $n_\theta$ (where the radius and angle are determined by projecting the vector onto the $xy$-plane). This results in each LED emitting a total of $n_r n_\theta$ vectors, where each vector carries a power of $x_i/(n_r n_\theta)$ ($i = 1, 2, ..., n$ represents the number corresponding to each LED). Two examples of vectors emanating from a single LED are shown in Fig. 4.

An additional concern is representing the energy that is absorbed by a surface (of leaves and the walls of the growth area, which are represented using triangles). To account for this, the power of each LED ($E_i$) is divided by the number of vectors emitted by that LED ($n_r n_\theta$). When the vector intersects a triangle, the energy the triangle absorbes is increased by $\alpha x_i/(n_r n_\theta)$, where $\alpha$ is the absorptivity assumed to take a value of $\alpha = 0.5$ (as assumed in Susorova et al. (2013)). The reflected vector is then assigned a power of $(1 - \alpha)x_i/(n_r n_\theta)$.

The sun was accounted for by creating a series of vectors starting on a plane defined by $z = 4$, $-2 - H \leq x \leq 2 + H$, $-2 - H \leq y \leq 2 + H$ (units in meters), where $H$ represents

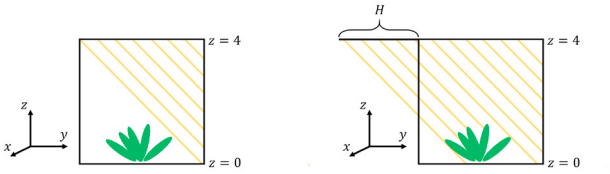


Fig. 5. LEFT: Side view of simulated greenhouse with no overhang. Note how most of the vectors representing sunlight do not reach the growing area. RIGHT: Image with overhang in one direction, now with most of the growing area now illuminated by sunlight.

the amount the plane extends beyond the size of the growing area. In this work, a value of $H = 7$ was selected. This overhang $H$ is necessary especially when the sunlight approaches at an angle, as otherwise much of the growing area will not be exposed to sunlight (see Fig. 5). Each sun vector is given a direction by first defining two angles $\theta_{sun_x}$ and $\theta_{sun_y}$, which are used along with the sun vector start point $(x_0, y_0, 4)$, to determine the point representing where the sun vector will intersect the floor $(x_f, y_f, 0)$ (assuming it does not intersect a leaf first). Using trigonometry, $x_f$ and $y_f$ can be written as $x_f = x + 4\tan(\theta_{sun_x})$ and $y_f = y + 4\tan(\theta_{sun_y})$. The direction of each sun vector is then found as $[x_f - x_0 \quad y_f - y_0 \quad -4]^T$.

### *2.4 Plant Growth*

A simple plant growth representation was also implemented by utilizing the Python API. Plant growth was assumed to be equal to the energy flux applied to each leaf (while this has no basis in reality, actual plant growth methods could also be simulated in the Python API). To implement plant growth in Blender, first the optimal LED energies were determined by utilizing the optimization problem with the reduced ray tracing model. Next, the full process ray tracing model with independent values of $N_R$, $n_r$, $n_\theta$, $n_x$, and $n_y$ (discussed in the following section) is used to determine the energy flux applied to each individual leaf (denote this as $E_{\text{leaf}_i}$). The amount each leaf is scaled is based on the following equation:

$$S_{\text{leaf}_i} = S_{\text{leaf}_i} + E_{\text{leaf}_i} \tag{6}$$

where $S_{\text{leaf}_i}$ is the current scaling factor of each leaf. This can be accomplished by first selecting each leaf in the geometry using the Python command

```
obj = bpy.data.objects["leaf1"]
```

where leaf1 is the name assigned to a part of the geometry representing a leaf. Following this, the leaf can be scaled by entering

```
obj.scale[i] = obj.scale[i] + scaleVal
```

where 'scaleVal' $= E_{\text{leaf}_i}$ and $i = 0, 1$, or 2 (corresponding to the x, y, or z-direction respectively). In this work, all three directions are scaled by the same value of $E_{\text{leaf}_i}$.

### *2.5 Independence Testing*

This section utilizes the Blender API to verify that the simulation uses a sufficient number of vectors and number of reflections. This was accomplished by running a series

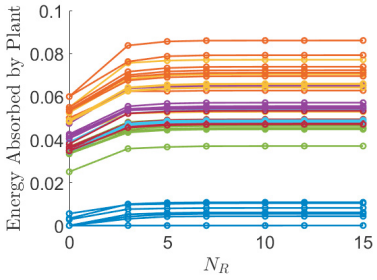


Fig. 6. Independence testing for the number of reflections, $N_R$.

| Test Number | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $N_R$ | 0 | 3 | 5 | 7 | 10 | 12 | 15 |
| $n_r$ | 3 | 5 | 10 | 15 | 20 | 25 | 30 |
| $n_\theta$ | 3 | 5 | 10 | 20 | 30 | 40 | 50 |

Table 1. Values of the number of reflections ($N_R$), the number of radial divisions in vectors generated for the LED ($n_r$), and the number of angular divisions in vectors generated for the LED ($n_\theta$) used for the independence testing in Fig. 6 and Fig. 7. The number of sun vectors was set to $n_x = n_y = 10$ for these tests.

of ray tracing simulations. The objective was to determine appropriate numbers of vectors from each LED ($n_r$ and $n_\theta$), the number of reflections ($N_R$), and the number of sun vectors ($n_x$ and $n_y$). Utilizing too few vectors will result in inconsistent results. For example, if each LED only uses 4 vectors, it is possible that none of them will intersect any leaves. A simulation using too many vectors will lead to excessive computation time. Here, the number of vectors / reflections is increased and results are plotted. When the results stop changing (indicated by a 'leveling off' on the plots), the corresponding number of vectors or reflections is deemed independent. This means that the results are independent of the number of vectors or reflections used (hence why this is called 'independence testing').

We begin by determining how many vectors each LED should emit and how many times these vectors should reflect. This was accomplished using the geometry from the right image in Fig. 1 and assuming a single ($N_{LED} = 1$) LED located at coordinates $(x, y, z) = (0, 0, 4)$. A set of ray tracing simulations was performed using all combinations of parameters listed in Table 1. In these simulations, the number of sun vectors was set to $n_x = n_y = 10$ with a total energy of 1. For each combination of $n_r$ and $n_\theta$ in Table 1, a plot was created, where the energy absorbed by the plant is plotted against the number of reflections $N_R$. The plots for all combinations are shown in Fig. 1. While Fig. 1 does not demonstrate which $n_r$ and $n_\theta$ values are independent, it is helpful for determining an independent value of $N_R$. To see this, note that for every combination of $n_r$ and $n_\theta$ in Fig. 1, the amount of energy absorbed by the plant approaches a constant value as $N_R$ increases at approximately $N_R = 7$ (a conservative estimate). For this reason, $N_R = 7$ is selected as an independent value.

Next, for determining the number of vectors $n_r$ and $n_\theta$ per LED, the data found using all combinations of parameters listed in Table 1 was analyzed. This time, for a given value of $N_R$, the energy absorbed by the plant versus number

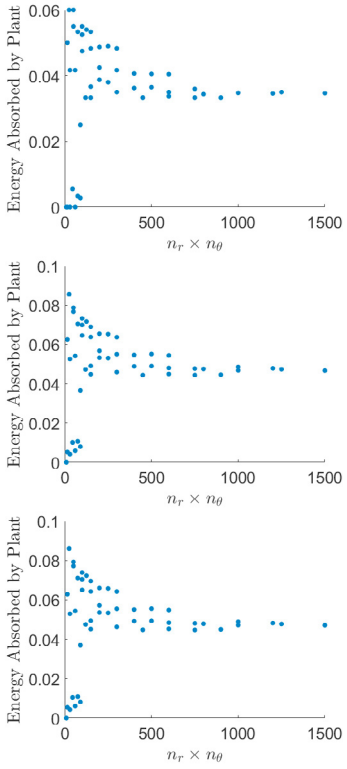


Fig. 7. Independence testing for the number of vectors from one LED. FROM TOP TO BOTTOM: $N_R = 0, 5, 10$.

of vectors per LED ($n_r \times n_\theta$) was plotted. The absorbed energy results, which are shown in Fig. 7 for the $N_R = 0$, $N_R = 5$, and $N_R = 15$ cases, tend to all approach a similar value at approximately $n_r \times n_\theta = 1000$ (again chosen conservatively). While there multiple possible values of $n_r$ and $n_\theta$ which have a product of 1000, in this work, values of $n_r = 20$ and $n_\theta = 50$ were selected as independent.

Finally, independence tests were performed to determine the number of sun vectors in the x-direction ($n_x$) and y-direction ($n_y$). For these simulations, it was assumed that $n_x = n_y$ with LED parameters of $N_R = 0$, $n_r = 3$, and $n_\theta = 3$ (small values were chosen because these values act independent of the number of sun vectors). Values of $n_x = n_y = 5$ to $n_x = n_y = 200$ were considered, using a total sun energy of 1, with the resulting energy absorbed by the plant versus $n_x \times n_y$ plotted in Fig. 8. Looking at Fig. 8, it can be seen that the value of the energy absorbed levels off as $n_x \times n_y$ increases. The independence occurs at approximately $n_x \times n_y = 2 \times 10^4$ (again chosen conservatively), which corresponds to $n_x = n_y = 142$ (rounded up to the nearest integer). For this reason, $n_x = n_y = 142$ is used as an independent value.

In summary, Table 2 lists the values for the numbers or vectors and reflections that were selected as independent (which are used in the full ray tracing model representing the actual process), and parameters used in the reduced ray tracing model (which are used as the model in the controller).

## 3. SIMULATED CLOSED-LOOP CONTROL RESULTS

Using the independent simulation parameters determined in the previous section (see Table 2) and the initial

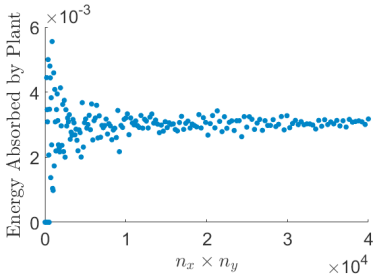


Fig. 8. Independence testing for the number of sun vectors

| | Full Process Model (Independent Results) | Reduced Model |
|---|---|---|
| $N_R$ | 7 | 0 |
| $n_r \cdot n_\theta$ | $20 \cdot 50 = 1000$ | $7 \cdot 15 = 105$ |
| $n_x \cdot n_y$ | $142 \cdot 142 > 2 \times 10^4$ | $30 \cdot 30 = 900$ |

Table 2. Simulation parameters selected for the reduced and full process ray tracing models.

geometry shown on the right in Fig. 1, an optimal set of LED powers were determined to achieve a desired absorbed energy by the plant under selected sunlight conditions. The parameters used in this simulation are shown in Table 3. The results of a simulation are listed

| | |
|---|---|
| Lower bound of absorbed energy | $E_{\min} = 0.2$ |
| Leaf absorptivity | $\alpha = 0.5$ |
| Number of LEDs | $N_{LED} = 9$ |
| Maximum LED power | $x_{max} = 1$ |
| Radius of LED light at $z = 0$ | $r_{LED} = 1.5$ |
| Total power of sun | $E_{sun} = 10$ |
| Angle of sun vectors | $\theta_{sun_x} = \theta_{sun_y} = 30°$ |
| Maximum triangle edge length | $L_{max} = 0.3$ |
| Minimum triangle height | $H_{min} = 0.01$ |
| Scaling constant for energy term | $C_1 = 2.5$ |
| Scaling constant for growth term | $C_2 = 1$ |

Table 3. Simulation parameters.

in Table 4. The overall power required to achieve at least $E_{\min} = 0.2$ energy absorbed by the plant is 1.286, of which 0.200 (in the reduced model in the controller) and 0.219 (in the full process model) is absorbed by the plant. Additionally, the energy per unit plant area was 0.111 (in the reduced model in the controller) and 0.113 (in the full process model). Finally, Fig. 9 shows the plant before and after the growth stage.

| LED Number | LED Coordinates | Optimal LED Power |
|---|---|---|
| 1 | $(-1, -1, 4)$ | 0.000 |
| 2 | $(-1, 0, 4)$ | 0.000 |
| 3 | $(-1, 1, 4)$ | 0.000 |
| 4 | $(0, -1, 4)$ | 0.000 |
| 5 | $(0, 0, 4)$ | 0.000 |
| 6 | $(0, 1, 4)$ | 0.286 |
| 7 | $(1, -1, 4)$ | 1.000 |
| 8 | $(1, 0, 4)$ | 0.000 |
| 9 | $(1, 1, 4)$ | 0.000 |

Table 4. Optimal LED energies rounded to the nearest 0.001. The total power usage is 1.286.

## ACKNOWLEDGEMENTS

Financial support from the Air Force Office of Scientific Research (award number FA9550-19-1-0059), Michigan Space Grant Research Consortium (award number 80NSSC20M0124), the National Science Foundation (award numbers CNS-1932026, CBET-1839675, and CBET-2143469), and Wayne State University are gratefully acknowledged.

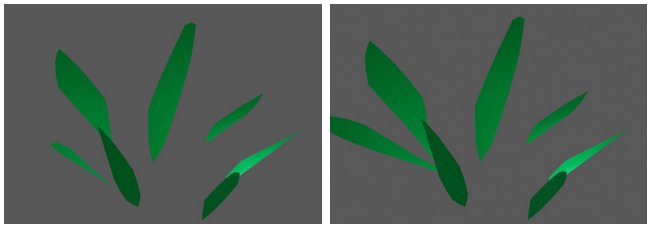

Fig. 9. Render of the plant geometry before (left) and after (right) applying plant growth.

## REFERENCES

Aradi, S. (2020). Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2), 740–759.

Dorst, L., Fontijne, D., and Mann, S. (2009). *Geometric algebra for computer science (revised edition)*. Elsevier.

Leonard, A.F., Gjonaj, G., Rahman, M., and Durand, H.E. (2024). Virtual test beds for image-based control simulations using blender. *Processes*, 12(2), 279.

Mezouar, Y. and Chaumette, F. (2002). Path planning for robust image-based control. *IEEE transactions on robotics and automation*, 18(4), 534–549.

Oyama, H., Leonard, A.F., Rahman, M., Gjonaj, G., Williamson, M., and Durand, H. (2022a). On-line process physics tests via lyapunov-based economic model predictive control and simulation-based testing of image-based process control. In *2022 American Control Conference (ACC)*, 2479–2484. IEEE.

Oyama, H., Messina, D., O'Neill, R., Cherney, S., Rahman, M., Rangan, K.K., Gjonaj, G., and Durand, H. (2022b). Test methods for image-based information in next-generation manufacturing. *IFAC-PapersOnLine*, 55(7), 73–78.

Scorsoglio, A., D'Ambrosio, A., Ghilardi, L., Gaudet, B., Curti, F., and Furfaro, R. (2022). Image-based deep reinforcement meta-learning for autonomous lunar landing. *Journal of Spacecraft and Rockets*, 59(1), 153–165.

Shirley, P., Ashikhmin, M., and Marschner, S. (2009). *Fundamentals of computer graphics*. AK Peters/CRC Press.

Susorova, I., Angulo, M., Bahrami, P., and Stephens, B. (2013). A model of vegetated exterior facades for evaluation of wall thermal performance. *Building and Environment*, 67, 1–13.

Trivedi, M.M., Gandhi, T., and McCall, J. (2007). Looking-in and looking-out of a vehicle: Computer-vision-based enhanced vehicle safety. *IEEE Transactions on Intelligent Transportation Systems*, 8(1), 108–120.

Yan, H., Paynabar, K., and Shi, J. (2014). Image-based process monitoring using low-rank tensor decomposition. *IEEE Transactions on Automation Science and Engineering*, 12(1), 216–227.