ABSTRACT
This paper presents techniques to display 3D illuminations using Flying Light Specks, FLSs. Each FLS is a miniature (hundreds of micrometers) sized drone with one or more light sources to generate different colors and textures with adjustable brightness. It is network enabled with a processor and local storage. Synchronized swarms of cooperating FLSs render illumination of virtual objects in a pre-specified 3D volume, an FLS display. We present techniques to display both static and motion illuminations. Our display techniques consider the limited flight time of an FLS on a fully charged battery and the duration of time to charge the FLS battery. Moreover, our techniques assume failure of FLSs is the norm rather than an exception. We present a hardware and a software architecture for an FLS-display along with a family of techniques to compute flight paths of FLSs for illuminations. With motion illuminations, one technique (ICF) minimizes the overall distance traveled by the FLSs significantly when compared with the other techniques.

KEYWORDS
3D Display, Point Cloud, Flying Light Specks, Illuminations, Failure, Battery Charging, Flight Paths, Data Replication, Parity Groups

1 INTRODUCTION
Unmanned Aerial Vehicles (UAVs) are enabling diverse applications ranging from journalism to entertainment [3, 12, 36]. A Flying Light Speck, FLS, is a miniature (hundreds of micrometer) sized UAV configured with Red, Green, and Blue light sources to render illuminations [16, 18]. It is battery powered, network enabled, with some storage, and processing to implement decentralized algorithms.

A swarm of cooperating FLSs is synchronized to render an illumination of a virtual object in a 3D FLS display. The display is a volume partitioned into a mesh of 3D cells. Each cell of the display is identified by its length, height, and depth (L, H, D) coordinates, see Figure 1b. We use the L, H, D coordinate system instead of X, Y, Z because there is no consensus on how mathematicians and the picture industry define the Y and Z axes. It is trivial to map our L, H, D coordinate system to either definition without ambiguity.

The size of a display cell is dictated by the FLS downwash, a region of instability caused by the flight of one UAV that adversely impacts other UAVs entering this region [4, 7, 15, 33, 47, 48], e.g., loss of control or unpredictable behavior. We assume the light emitted by an FLS is larger than a display cell, positioning the FLS at the center of multiple cells along each dimension.

A static illumination is a point cloud. Each point \( p_i \) identifies a 3D coordinate with a value for its color model. In this study, we assume the RGBA model that specifies the red, green, blue, and alpha color settings. Hence, a point is a \( \{ l_i, h_i, d_i, R_i, G_i, B_i, A_i \} \) where the interval \( \{ s_i, e_i \} \) specifies when the point identified by \( p_i \) should be illuminated relative to the start of the stream. Alternatively, it maybe a stream of point clouds that must be rendered at a pre-specified rate, e.g., 24 point clouds per second. Yet another possibility is a hybrid of these two by associating only one \( \{ s, e \} \) for a point cloud and individual \( \{ s_i, e_i \} \) for select points. This paper assumes the second representation. An example is the Rose illumination of Figure 1a with a falling petal.

Display of a motion illumination is continuous when FLSs render its n point clouds in a timely manner. Each point cloud has a start and an end time stamp relative to the start of the illumination. FLSs rendering a point cloud \( \Xi_i \) at time \( T_i \) must fly to positions dictated by the next point cloud \( \Xi_{i+1} \) at time \( T_{i+\Delta} \). \( \Delta \) is dictated by the rate of point clouds displayed per unit of time, e.g., \( \Delta = \frac{1 \text{ Second}}{24} \) when 24 point clouds are rendered per second. Once at their new position, FLSs must render the lighting required by the point cloud \( \Xi_{i+1} \).
process continues until all point clouds of a motion illumination are displayed.

Display of both static and motion illuminations is non-trivial for several reasons. First, a rendering may require a large number of FLSs. For example, each point cloud of the Rose illumination consists of 65K points (FLSs). The Rose illumination is simple. We anticipate more complex illuminations to consist of millions and potentially billions of points.

Second, FLSs are mechanical devices that fail. Hence, failures are the norm rather than an exception. A display requires techniques to render an illumination in the presence of FLSs failing continuously. Third, each FLS is battery powered with a fixed flight time. Its battery requires a certain amount of time to charge. A key question is what is the relationship between these factors and the extra number of FLSs required to render an illumination? Sections 4 and 5 provide an answer.

Fourth, flight of FLSs may result in collisions. Computing collision free paths is expensive with tens of UAVs [4, 6–11, 14, 15, 21, 25, 29, 30, 32, 33, 38, 43–48]. This may be prohibitively expensive with tens of thousands of FLSs. It may be impractical in the presence of FLSs with limited flight times failing constantly. Our design philosophy is to detect FLS conflicts when computing flight paths. We provide this information to the FLSs that participate in the conflict. When FLSs take flight to render an illumination, they use this information to implement a decentralized technique to avoid collisions. A simple collision avoidance technique is for the participating FLSs to take turns flying to their destination by using their unique identifier to order themselves. Such decentralized techniques are implemented using the networking, processing, and storage capabilities of FLSs.

Contributions of this paper include:

- An architecture for FLS displays to render 3D static and motion illuminations. (Section 2.)
- MinDist and QuotaBalanced algorithms to render a static illumination. Both algorithms are fast and run in tens of milliseconds. (Section 3.1.)
- Motill, a family of offline algorithms to compute flight paths of FLSs that render a motion illumination. One technique, ICF, minimizes the overall distance travelled by FLSs when compared with the other alternatives. (Section 3.2.2.)
- A technique that uses standby FLSs to render an illumination in the presence of FLS failures. With once a month as the mean time to failure of an FLS, the quality of the Rose illumination degrades once every 40 seconds due to failures. Our proposed techniques enhance this to once a month or more by using additional FLSs. (Section 4.)
- STAG as a technique that overlaps charging of some FLS batteries with other FLSs rendering an illumination. We prove optimality of this algorithm in minimizing the total number of FLSs and charging stations for an illumination. (Section 5.)
- We open source our software and data pertaining to the Rose illumination for use by the scientific community. See https://github.com/shahramg/FLS-Multimedia2022 for details.

The rest of this paper is organized as follows. Section 2 presents a hardware and software architecture for an FLS display. Section 3 presents algorithms to render static and motion illuminations. Section 4 describes FLS failure handling. Section 5 presents an optimal algorithm for continuously charging FLS batteries. Related work is presented in Section 6. Section 7 presents future research directions.

2 ARCHITECTURE

A 3D FLS display consists of a number of software and hardware components. These include hangars that protect FLSs from external factors that may either damage them or reduce their lifetime. A hanger may be accessible to one or more Dispatchers, Charging Stations, and Terminus.

Charging Stations charge the battery of FLSs, depositing those with a fully charged battery into a hanger. Dispatchers deploy FLSs to render an illumination. Dispatchers may communicate identity, flight path, and deploy time of their FLSs to one another to detect potential FLS crashes. Dispatchers implement algorithms to avoid potential crashes, see Section 3.1.

Garbage Collectors, GCs, collect failed FLSs that fall to the bottom of the display and bring them to a Terminus. A Terminus has one or more entry points in the display. In addition to GCs, an FLS that detects it may no longer function properly may fly to a Terminus, see Section 4.

Terminus and Charging stations have well defined entry points known to the FLSs.

A Hub provides the processing, storage, and networking capabilities of the FLS display. It uses an off-the-shelf operating system. The Hub executes a software component, named the Orchestrator, that manages FLSs in Hangars, Charging stations, Dispatchers, Garbage Collectors, and Terminus. It also manages the storage and network of the Hub. The Orchestrator may delegate tasks to other components. For example, it may delegate deployment of FLSs to one or more dispatchers, see Section 3.1.

The Orchestrator implements centralized algorithms to render a motion illumination. It may also implement hybrid centralized and decentralized algorithms that include the participation of the FLSs. For example, with the parity-based technique of Section 4, the Orchestrator may identify the number of FLSs in a group and their identity. However, detection of FLS failures and subsequent
substitution of a parity FLS for the failed FLS may be performed by the FLSs without the Orchestrator involvement. At the other end of the spectrum, certain tasks may be implemented in a decentralized manner independent of the Orchestrator. An example is collision avoidance implemented by FLSs.

Figure 2 depicts these components as the bottom of an FLS display that sits on a floor or a table top. The volume that renders an illumination is above the garbage collector, i.e., the conveyor belt at the top. A cylinder at each corner serves as a dispatcher. Each is accompanied by a cylinder that an FLS flies into to obtain access to the charging stations. The charging stations deposit fully charged FLSs into hangars located on the two sides and the bottom of the display. Figure 2 shows the Hub as a server blade installed below the charging stations. Above the charging stations is the garbage collector. Failed FLSs fall on the garbage collector’s conveyor belt that rotates and deposits these FLSs in a Terminus at either end.

Figure 2 is one from many possibilities. With a wall-mounted 3D display and other application use cases, the organization may be completely different.

### 3 Display of 3D Illuminations

To display an illumination, the Orchestrator constructs a 3D mesh on the display volume. A cell of this mesh is dictated by the downwash of an FLS. Assuming an FLS is a quadrotor, a cell may be an ellipsoid \([4, 7, 33]\) or a cylinders \([15, 48]\) that results in a larger separation along the height dimension. Each display cell has a unique \((L, H, D)\) coordinate. It is referenced by one point in the point cloud.

**Definition 3.1.** A display cell is occupied by one FLS. Its size is dictated by the downwash of the FLS. It is identified by a unique \((L, H, D)\) coordinate.

We use a cuboid to represent an illumination cell rendered by an FLS light source. The size of this cuboid is dictated by the characteristics of the FLS light source. It may be either smaller than, equal to, or larger than a display cell. When it is smaller, an FLS may be configured with multiple sets of RGB light sources to render different points. When equal to or greater than, an FLS may be configured with one set of RGB light sources. When greater than, the FLS’s light sources illuminate a point that corresponds to multiple display cells. Hence, the FLS is placed at the center of these display cells for illumination. This scenario is assumed in this paper, enabling an FLS to pass by FLS\(_j\) as long as it stay outside of FLS\(_j\)’s display cell. We defer the other two cases to future work.

**Definition 3.2.** A cuboid consists of six flat faces and eight vertices. All its faces are rectangles. A cuboid is a square prism when at least two faces are squares. A cuboid is a cube when all its six faces are squares. This paper refers to all as a cuboid.

**Definition 3.3.** An illumination cell is a cuboid that is larger than a display cell. An FLS is positioned such that its rendered light fills one or more faces of the cuboid. Typically, the illumination cell is occupied by one FLS. Depending on its size, two or more FLSs may occupy it with at most one rendering its light. The other FLSs may either be a standby for failure handling (see Section 4) or transitory on their route to their assigned coordinates or a charging station.

Figure 3 shows an illumination cell consisting of \(L=5, H=5,\) and \(D=5\) display cells. It consists of 125 display cells. It shows an FLS at the center of the illumination cell, at coordinates \(L=3, H=3,\) and \(D=3\) of the illumination cells. The cones show the FLS rendering its top and bottom RGB lights to illuminate the top and bottom faces of the illumination cell. A maximum of 125 FLSs may occupy this illumination cell with only one rendering its light sources to illuminate one or more faces of the illumination cell. The other 124 may be transitory on their path to a destination, e.g., another illumination cell, a charging station. The display cell considers downwash. Hence, FLSs in an illumination cell will not interfere with one another as long as they occupy a display cell.

**Definition 3.4.** A point in a point cloud identifies an illumination cell of an FLS display. A single FLS illuminates the point.

This section describes display of illuminations assuming FLSs do not fail and have unlimited flight times. These assumptions are removed in Sections 4 and 5, respectively.

---

**Table 1: Notations and their definitions.**

| Notation | Definition |
|----------|------------|
| \(\Xi_i\) | Point cloud \(i\). |
| \(\psi\) | Number of dispatchers. |
| \(\theta\) | Max number of points/FLSs assigned to a cuboid. |
| \(\beta\) | Flight time of an FLS on a fully charged battery. |
| \(\Omega\) | Time to charge an FLS battery fully. |
| \(\kappa_i\) | Number of points assigned to Dispatcher \(i\). |
| \(\epsilon_i\) | Flight paths that transition FLSs of \(\Xi_i\) to render \(\Xi_{i+1}\). |
| \(\alpha\) | Number of points/FLSs in a point cloud. |
| \(MTTF\) | Mean Time To Failure of an FLS. |
3.1 Display of Static Illumination

An algorithm may illuminate a point cloud based on different objectives. An example objective is to minimize the total distance traveled by FLSs to arrive at the coordinate of their assigned point, i.e., an illumination cell. The formal definition of this assignment problem is as follows.

**Problem 1.** An illumination consists of $\alpha$ points and a display consists of $\psi$ dispatchers. The distance from a dispatcher to a point is fixed, $\text{Distance}(\text{Dispatcher}_i, P_j)$. Assign each point to a dispatcher such that one and only one point is assigned to a dispatcher and the total distance for a dispatcher and its assigned point is minimized, i.e., minimize $\sum_{i=1}^{\psi} \sum_{j=1}^{\alpha} \text{Distance}(\text{Dispatcher}_i, P_j)$ where $\kappa_i$ is the number of points assigned to a dispatcher $i$.

MinDist is an algorithm that iterates each point, computes the distance of the point to a dispatcher, and assigns the point to the dispatcher with the shortest euclidean distance.

Subsequently, each dispatcher sorts its assigned points in descending order based on their distance from its location. It deploys FLSs to render points starting with the farthest away one first. This minimizes the possibility of dispatched FLSs from colliding with one another.

MinDist’s iteration of points is sequential and may be implemented by the Orchestrator. However, each dispatcher may perform its sorting and deployment of FLSs in parallel and independent of the Orchestrator. Due to lack of space, we refer the interested reader to [17] for a complexity analysis of MinDist.

MinDist has several limitations. First, it may result in a slow rendering of an illumination by utilizing a subset of dispatchers more often than others. This happens when most of the points in a cloud are clustered in close proximity of a few dispatchers. While these dispatchers deploy most of the FLSs sequentially, other dispatchers sit idle. See discussions of Table 2 in Section 3.1.1.

Second, MinDist assumes a dispatcher may access all hangars and their FLSs. This assumption is violated when hangars are physically partitioned across dispatchers. MinDist may not render an illumination that consists of a cluster of points in close proximity of a dispatcher with insufficient number of FLSs.

We now present Alg 1. QuotaBalanced, that considers the distance travelled by FLSs, FLS speed, the number of dispatchers, the number of FLSs accessible to a dispatcher, and the rate at which a dispatcher may deploy FLSs. It assigns a quota to each dispatcher that is reduced as a function of the travel time by its deployed FLSs. The idea is to have a dispatcher that is very far from the points of a cloud to deploy some FLSs but not as many FLSs as those dispatchers that are in close proximity to the points of the cloud.

Distance is an approximation of travel time. The time for an FLS to fly from a dispatcher to its display cell is a function of the FLS speed. A dispatcher may be far from the point cloud. However, if FLSs are extremely fast then their travel time may become insignificant to the time to deploy FLSs. This motivates an algorithm that requires each dispatcher to deploy its fair share of FLSs while considering travel time of FLSs. QuotaBalanced is one such algorithm.

---

**Algorithm 1: QuotaBalanced**

1. $f \leftarrow \text{FLS deployment rate, FLSs second}$
2. $S \leftarrow \text{FLS Speed}$
3. $\omega[1..\psi] \leftarrow \frac{\alpha}{\psi S}$; /* Quota for each dispatcher */
4. $\{\text{Active}\} \leftarrow \text{Dispatchers with FLSs and quota } > 0$;
5. for $i \leftarrow 1$ to $\alpha$ by 1 do
6. $\text{Dispatcher}_i \leftarrow \text{The active dispatcher closest to Point}_i$;
7. $\Delta \leftarrow \text{Distance}(\text{Dispatcher}_i, \text{Point}_i)$;
8. $\text{Dispatcher}_i \leftarrow \text{Point}_i$;
9. if $\text{Dispatcher}_i$ has zero FLSs then
10. $\{\text{Active}\} \leftarrow \{\text{Active}\} – \text{Dispatcher}_i$;
11. Remove $\text{Dispatcher}_i$ from further consideration;
12. $t \leftarrow \frac{\Delta}{S}$; /* FLS travel time */
13. $\omega[\text{Dispatcher}_i] = \omega[\text{Dispatcher}_i] – t$;
14. if $\omega[\text{Dispatcher}_i] < 0$ then
15. $\{\text{Active}\} \leftarrow \{\text{Active}\} – \text{Dispatcher}_i$;
16. if $\{\text{Active}\} == \emptyset$ then
17. $\omega[1..\psi] \leftarrow \frac{\alpha}{\psi S}$; /* Re-compute dispatcher quota */
18. $\{\text{Active}\} \leftarrow \text{Dispatcher}_i$ with FLS[i]>0 and $\omega[i] > 0$;
19. for $j \leftarrow 1$ to $\psi$ by 1 do
20. $\text{Dispatcher}_i$ sorts its points in descending distance order;
21. $\text{Dispatcher}_i$ deploys points starting with the farthest away one;

QuotaBalanced assumes each dispatcher may deploy $f$ FLSs per time unit (Line 1 of Alg 1) and dispatcher $i$ has access to a fixed number of FLSs. The granularity of its quota is time units required for a dispatcher to deploy its fair share of FLSs for the point cloud, $\frac{\alpha}{\psi S}$. This algorithm converts distance to time using the speed of an FLS, see Line 12.

In each iteration, QuotaBalanced reduces the quota of a dispatcher by the FLS travel time to its assigned point. Once the quota of a dispatcher is exhausted, it is removed from the list of active dispatchers. This causes QuotaBalanced to assign points to other dispatchers that are not necessarily as close. However, the quota of these dispatchers are reduced by a larger value because they are farther away, i.e., time to travel is longer. Hence, these dispatchers will be removed from the active list after a fewer point assignments.

The quota of all dispatchers may be exhausted while some points remain unassigned. QuotaBalanced re-computes the quota of each dispatcher using the remaining points, see Lines 16-18. It continues to assigns points to the dispatchers until their quotas are exhausted. This process repeats until all points are assigned to dispatchers. The number of repetitions is 1393 with the point cloud of Section 3.1.1.

A dispatcher with no FLSs is permanently removed from the active list (Lines 9-11). This causes other dispatchers to deploy FLSs to render the illumination.

Once points are assigned to dispatchers, a dispatcher may deploy FLSs similar to the discussion of MinDist, see Lines 19-21. There is one difference. With QuotaBalanced, dispatchers may deploy FLSs that cross paths and potentially crash with one another. Dispatchers may share the flight path and deployment time of their FLSs with one another to detect such potential crashes. Prior to deploying an FLS, a dispatcher may detect a conflict with other deployed FLSs traveling to their target point. It may implement
Alternatively, the dispatcher may compute a different flight path for its FLSs, eliminating a possible crash.

3.1.1 A Comparison. We use the Princeton Shape Benchmark [37] to highlight the quantitative and qualitative differences between MinDist and QuotaBalanced. Its database consists of 1,814 3D models. We present results from the race car (m1510) with 11,894 points.

We simulate an FLS display that is a cube with length=100, height=100, and depth=100 cells. This display consists of 8 dispatchers, one at each corner of the cuboid. Each dispatcher may deploy 10 FLSs per second, one FLS every 100 milliseconds. The speed of each FLS is 4 cells per second. An FLS flies along a straight line from the location of its dispatcher to the coordinates of its assigned point.

While QuotaBalanced uses all 8 dispatchers\(^2\) to deploy FLSs, MinDist uses only 2 at the bottom corner\(^3\) of the display. Hence, QuotaBalanced enhances latency four folds, see 1st row of Table 2. This comes at a cost, namely, an increase (≈2x) in the total distance travelled by the FLSs, see 2nd row of Table 2.

QuotaBalanced exhausts the list of active dispatchers and resets their quota (Lines 16-18 of Alg 1) 1393 times with this model.

Our simulated dispatchers detect when the path of their deployed FLSs intersect one another, identifying a conflict that may result in FLSs crashes. QuotaBalanced incurs 35 such conflicts. Not every detected conflict is a crash because one FLS may fly past the crash point in advance of the other conflicting FLSs. In our study, FLSs must be in 20% proximity of one another to be considered as conflicting. Table 2 shows QuotaBalanced incurs 12 such conflicts. This is because dispatchers deploy FLSs that are farther away from a point in the display coordinate system than the other dispatchers. This is a small percentage (0.1%) of the 11,894 deployed FLSs by the 8 dispatchers. See the last paragraph of the previous section on how to eliminate possible crashes.

### Display of Motion Illuminations

We assume a motion illumination consists of a sequence of point clouds displayed at a specified rate. See Figure 1a.

Assuming an FLS corresponds to a point, a display must compute both travel path of FLSs and their change of color from one point cloud \(\Xi_1\) to the next point cloud \(\Xi_{i+1}\). While these changes may be minor with point clouds that constitute a scene, they may be drastic from the last point cloud of one scene to the first point cloud of its following scene. This paper focuses on computing the intra-scene travel paths, deferring inter-scene travel paths to future work.

To render a scene, the display must assign an FLS to each point of its first point cloud \(\Xi_1\). This is identical to rendering a static illumination. Thus, either MinDist or QuotaBalanced maybe used. To render its subsequent point cloud \(\Xi_2\), the Orchestrator must compute whether:

1. \(\Xi_2\) consists of more points than \(\Xi_1\), requiring additional FLSs to render it. In general, dark FLSs from a previous point cloud (say \(\Xi_{i-1}\)) may be used to render \(\Xi_{i+1}\). If none are available then FLSs are deployed by a dispatcher.
2. \(\Xi_2\) consists of fewer points than \(\Xi_1\), requiring some FLSs illuminating \(\Xi_1\) to either go dark or fly to a charging station. Dark FLSs may be used in a subsequent point cloud, say \(\Xi_3\). This may minimize the overall distance travelled by FLSs. This is because requiring FLSs to fly back to a charging station for \(\Xi_2\) only to dispatch FLSs to illuminate \(\Xi_3\) may result in a longer total travel distance.
3. An FLS illuminating \(\Xi_1\) remains stationary and changes color in \(\Xi_2\).
4. An FLS illuminating \(\Xi_1\) flies to a new point identified by \(\Xi_2\) and displays either the same or a different color.
5. An FLS illuminating \(\Xi_1\) remains stationary and continues to display its current color in \(\Xi_2\). This is the scenario where the point in \(\Xi_1\) and \(\Xi_2\) are identical, i.e., identify the same illumination cell and render the same color.

Any and all combinations of these possibilities may apply when considering two point clouds \(\Xi_i\) and \(\Xi_{i+1}\).

This section presents two offline algorithms, Simple and Motill, to detect the alternative scenarios and compute FLS flight paths and color renderings. Similar to the MPEG encoding technique, a system may execute these algorithms once for a motion illumination, store their computed flight paths, and reuse this information for repeated rendering of the motion illumination.

3.2.1 Simple. Simple is an offline algorithm that consists of two steps. Step 1 computes flight paths that transition one point cloud to the next. It also identifies FLSs that are extras from one point cloud to next. And, points that may require FLSs to be deployed by a dispatcher. More formally, with \(n\) point clouds, \(\{\Xi_1, \Xi_2, \ldots, \Xi_n\}\), Step 1 computes four sets. Flight paths for the first \(n-1\) point clouds, denoted \(\{e_1, e_2, \ldots, e_{n-1}\}\). Change of color for the FLSs used in the first \(n-1\) point clouds, denoted \(\{y_1, y_2, \ldots, y_{n-1}\}\). Extra FLSs for the first \(n-1\) point clouds, denoted \(\{\delta_1, \delta_2, \ldots, \delta_{n-1}\}\). There are extra FLSs when \(\Xi_i\) consists of more FLSs than \(\Xi_{i+1}\). Points with no assigned FLSs for the last \(n-1\) point clouds, denoted \(\{\mu_2, \mu_3, \ldots, \mu_n\}\). \(\Xi_i\) may have points with no assigned FLSs when it consists of more points than \(\Xi_{i-1}\).

In Step 2, Simple processes \(\{\delta_i\}\) to decide whether one or more of its FLSs should stay in the display grid and go dark or fly back to a charging station. The dark FLSs are assigned to a point identified by a \(\{\mu_{j+1}\}\). Step 2 may schedule a dispatcher to deploy FLSs to illuminate points of \(\{\mu_j\}\).

Step 2 is required when either \(\{\delta_i\}\), \(\{\mu_j\}\), or both are not empty. Step 1 produces empty \(\{\delta_i\}\) and \(\{\mu_j\}\) when the different point clouds consist of the same number of points.

| Table 2: MinDist vs. QuotaBalanced |
|-----------------------------------|----------------|
| Illumination Latency (Seconds)    | 661            |
| Distance Traveled (Cells)         | 494,938        |
| Intersecting flight paths         | 0              |
| FLS Conflicts                      | 0              |
| Execution Time (Milliseconds)     | 27.71          |
Simple may optimize for total distance travelled, the amount of energy used, the time required to execute flight paths, or a hybrid of these. The hybrid may assign weights to different criteria. This paper focuses on minimizing the total distance travelled by FLSs, deferring other possibilities to future work. Due to lack of space, we refer the reader to [17] for details of Simple’s two steps.

3.2.2 Motion Illuminations, Motill, Encoding. Motill is a family of divide-and-conquer encoding techniques to implement Step 1 of Simple. They localize mapping of the points by constructing a 3D grid on the point clouds. Instead of computing the distance between a freed FLS of \(\Xi_i\), \(\{\delta_i\}\), with every vacant coordinate of \(\Xi_{i+1}\), \(\{\mu_{i+1}\}\). Motill compares those in the same cuboid or its neighboring cuboids. Other pairings are guaranteed to be farther away. By eliminating them from consideration, Motill reduces complexity of Step 1 to provide a faster execution time when computing FLS flight paths. In addition, a Motill technique named ICF provides shorter flight distances for FLSs when compared with Simple.

Motill partitions a scene consisting of \(n\) points clouds into a Group of Point Clouds (GPCs). Each GPC consisting of \(\omega\) point clouds, \(\{\Xi_1, \Xi_2, \cdots, \Xi_\omega\}\). In a GPC corresponds to an FLS. Motill constructs flight paths for different FLSs across a GPC \(\{\Xi_1, \Xi_2, \cdots, \Xi_\omega\}\). Subsequently, Motill combines flight paths for different GPCs together to compute the travel path of FLSs for the entire scene.

Motill processes a GPC by constructing a 3D grid on its first point cloud \(\Xi_1\). A maximum limit \(\theta\) is imposed on the number of points assigned to each cuboid of the grid. Every time a cuboid overflows, Motill breaks the cuboid into two by partitioning it along a dimension. In our current implementation, we use a round-robin policy to select among the dimensions across all cuboids. However, it is possible to develop more sophisticated policies to better balance points across the cuboids.

Motill constructs a copy of the \(\Xi_1\) grid with \(\rho_1\) cuboids on the remaining point clouds of a GPC, \(\Xi_2\) to \(\Xi_\omega\). It scans points of \(\Xi_i (1 < i \leq \omega)\) and assigns each to the cuboid that contains it, populating \(\Xi_i\) grid. This step does not detect overflows and has no cuboid splits. Hence, the cuboids of these point clouds may have more points than the maximum limit \(\theta\).

The purpose of the grid is to reduce the number of points considered when computing the shortest distance. Its cuboids localize how a point (FLS) changes position from one point cloud to the next. By using the same grid across all point clouds of a GPC, Motill localizes changes to a few cuboids. We refer the interested reader to [17] for details of how this is accomplished, including use of parallelism.

3.2.3 Variants of Motill. Motill is a family of techniques. In this paper, we consider two variants: Intra-Cuboid-First (ICF) and Intra-Cuboid-Last (ICL). ICF computes intra-cuboid flight paths for every cuboid pairing first. For the remaining cuboids, it computes inter-cuboid flight paths. It is motivated by the insight that computing flight paths local to a cuboid minimizes distance.

ICL reverses the order of these two steps, computing inter-cuboid flight paths first and intra-cuboid flight paths last. Its motivation is that computing inter-cuboid flight paths first has the benefit of more candidate FLSs in \(\{\delta_i\}\) and vacant destinations in \(\{\mu_{i+1}\}\).
We describe grouping of FLSs with a standby to enhance MTDI and
An FLS is a mechanical device that may fail. Its failure may degrade
Figure 5. The point clouds that constitute the Rose illumination
flight distances. This is true with both ICF and ICL as shown in
their neighbors in the display mesh to detect processor and battery
have the same number of points (65,321). Hence, Simple and Motill
do not execute their Step 2.
Due to lack of space, see [17] for a detailed analysis of results.

## 4 FLS FAILURE HANDLING
An FLS is a mechanical device that may fail. Its failure may degrade
the quality of a rendering by not illuminating one or more of its
points. There are several types of FLS failures: rotor failures, light
source failures, computing failures in the form of reboots, and
battery power failures. Assuming these failures are independent
and occur at a constant rate, one may compute the Mean Time To Failure
of an FLS (MTTF) similar to how magnetic disk manufacturers
calculate the MTTF of disk drives [31]. The Mean Time to Degraded
Illumination (MTDI) is a linear function of the number of FLSs (\( \alpha \))
that constitute an illumination: \( MTDI = \frac{MTTF \times \alpha}{\alpha} \).

Assuming an FLS fails once a month (MTTF of 720 hours), the
MTDI of the Rose illumination with \( \alpha = 65,321 \) FLSs is 40 seconds.
We use a group parity/replication technique to enhance MTDI of
an illumination in the presence of frequent FLS failures. We start
by describing how FLSs cooperate to detect failures. Subsequently,
we describe grouping of FLSs with a standby to enhance MTDI and
conclude with an analysis of MTDI as a function of group size.

**Failure detection:** FLSs cooperate to detect failures and notify the
Orchestrator of the identity of the failed FLS. This cooperation is
in two forms. First, once an FLS detects its own failure, it uses its
networking to inform its neighbors and the Hub (Orchestrator) of
its failures. This applies to the first two forms of failures. With
light source failures, the FLS flies to a Terminus immediately as
it is no longer able to illuminate a point. With rotor failures, it
repels [23, 26, 41] FLSs in its downward descent by generating
frequent failed messages. Those FLSs that receive this message
move away to prevent the failed FLS from crashing into them. With
the architecture of Figure 2, the failed FLS falls on the conveyor
belt of the garbage collector and is deposited into a Terminus.

Second, FLSs exchange periodic heartbeat messages [34, 39] with
their neighbors in the display mesh to detect processor and battery
failures. An FLS that encounters these forms of failures may not
be able to notify other FLSs of its failure. Hence, FLSs cooperate to
detect these failed FLSs. If an FLS does not receive an anticipated
heart beat message from one of its neighbors then it polls the
neighbor. After a few failed attempts, it identifies the neighbor as
having failed and communicates the identity of this failed FLS to
its neighbors and the Hub (Orchestrator).

To maintain the quality of an illumination, a failed FLS must be
replaced with a new one quickly. The system must restore both the
physical FLS and the data that describes its flight path and lighting
responsibilities. This data is stored on the local storage of each FLS.

We use standby FLSs to recover from FLS failures. In normal
mode, standby FLSs are dark. After the discovery of a failed FLS,
the standby assumes the lighting responsibilities and flight paths
of the failed FLS. This occurs concurrently with the Orchestrator
deploying a replacement FLS to substitute for the standby.

We use parity and replication techniques to maintain the data
of a failed FLS available. These techniques assign \( G \) FLSs in close
proximity of one another to a group and assigns one or more stand-
bys to each group. In [17], we provide additional details including
construction of groups and analytical models [31] to compute the
Mean Time to Degraded Illumination, MTDI.

Table 3 shows MTDI of the Rose illumination assuming an FLS
fails once a month and the system’s MTTR is 1 second. Reliability
groups enhance MTDI from 40 seconds to almost two months with
20 FLSs in a group, \( G=20 \). This is enhanced almost two folds (to
111 days) with 10 FLSs per group, \( G=10 \). With \( G=10 \), the Rose
illumination requires approximately 6,400 additional FLSs.

## 5 STAGGERED BATTERY CHARGING, STAG
STAG is a novel algorithm that staggering charging of FLS batteries as
a function of time to minimize both the number of charging stations
and the overall number of FLSs required to render an illumination.

STAG assumes each FLS has a battery with a finite flight time \( \beta 
when fully charged. And, \( \Omega \) time units are required to fully charge
a depleted battery with minimal or no remaining flight time left.

An FLS computes the amount of battery flight time required for
it to fly to a charging station using its distance to the charging
station. Once its battery flight time reaches this threshold, the FLS
will go dark and fly to the charging station.

A standby FLS (see Section 4) will substitute for this FLS to
perform its lighting responsibility while a dispatcher will deploy
an FLS with \( \beta \) flight time to substitute for the standby.

It is important to minimize the window of time \( \Delta \) for a battery
deprecated FLS to switch places with a fully charged FLS. Should
an FLS belonging to the parity group of FLS fail during \( \Delta \), this
may result in loss of information and a degraded illumination.

**STAG** staggers FLSs as a function of time to prevent them from
exhausting their finite flight time at the same time. It overlaps charg-
ings of some FLSs with others that are rendering an illumination.
In its steady state, STAG switches a fully charged FLS with a fully
deprecated FLS continuously. Details of STAG are as follows.

STAG constructs \( b \) flocks of FLSs. A flock \( i \) consists of \( \alpha_i \) FLSs.
Within a flock \( i \), STAG staggers its \( \alpha_i \) FLSs such that their remaining
battery flight time ranges from \( \beta \) down to the staggering interval \( S \),
\( S = \frac{\beta}{\alpha_i} \). Assuming the FLSs in a flock are numbered from 1 to \( \alpha_i \),
the remaining flight time of FLS \( j \) is \( \beta(j) = \frac{j \beta}{\alpha_i} \). Thus, FLS \( j = \alpha_i \) has
a fully charged battery with \( \beta \) flight time.

The number of FLSs charging (or staged in a hangar) to substitute
for an FLS of Flock \( i \) is \( \left\lceil \frac{\alpha_i}{T} \right\rceil \). This is the extra number of FLSs
required by a flock to render an illumination. The total number of
FLSs that are charging (or staged in a hangar) is \( h \times \left\lceil \frac{\alpha}{T} \right\rceil \).

| Table 3: Reliability Groups enhance MTDI. |
|------------------------------------------|
| Total Number of FLSs | G=10 | G=20 |
| Overhead Cost | 71,853 (1.1%) | 68,588 (1.05%) |
| MTDI Hours (Days) | 2670 (111) | 1399 (58) |
Table 4: STAG with the Rose illumination.

|                   | $\beta=5$ min | $\beta=10$ min | $\beta=20$ min |
|-------------------|---------------|----------------|---------------|
| $\Omega=10$ min   | 218           | 109            | 55            |
| $\Omega=5$ min    | 300           | 600            | 1200          |
| $\Omega=2.5$ min  | 600           | 300            | 150           |
| FLSs for the last flock | 221           | 521            | 521           |
| S for the last flock (Millisec) | 1358         | 1152           | 2303          |
| Extra FLSs for the last flock | 442          | 261            | 65.125        |
| Extra FLSs for the illumination | 130642 | 32661 | 8166 |
| Overhead Cost     | 200%          | 50%            | 12.5%         |
| Total Number of FLSs | 195,963      | 97,982         | 73,487        |

STAG minimizes the number of additional FLSs required to render an illumination. See [17] for a formal theorem and its proof.

The number of FLSs in transit from a charging station to an illumination is $2h$. Given a swarm, reducing its number of flocks to one, $h=1$, minimizes the number of FLSs in transit. This reduces the likelihood of a dark FLS from obstructing the user’s field of view. However, the number of flocks is dictated by the maximum time required for an FLS to fly back to the charging station. $S_{\text{threshold}}$ and the number of FLSs required by an illumination $\alpha$. With an illumination that consists of a large number of FLSs, maintaining $h=1$ results in a small staggering interval $S$ (because $S$ is a function of the number of FLSs in a flock). $S_{\text{threshold}}$ dictates the number of FLSs in a flock ($\alpha = \frac{S}{S_{\text{threshold}}}$) which in turns dictates the number of flocks, $h = \lceil \frac{2\beta}{\alpha} \rceil$. Flocks that constitute an illumination may use different staggering intervals $S$ and consists of a different number of FLSs $\alpha$. This is highlighted by the analysis below. In [17], we describe how STAG forms flocks and initiates an illumination to realize staggering of FLSs by $S$ time units.

An Analysis. Table 4 quantifies the behavior of STAG with different $\beta$ and $\Omega$ values. The first column pertains to the flight time and battery charge time of today’s Sky Viper Dash Nano Drone. The other two columns correspond to future generations of such a device with flight time on a fully charged battery ($\beta$) doubling and the time to charge ($\Omega$) its battery is halved.

We set the lower bound on $S$ to 1 second. It limits the number of FLSs in a flock, $\alpha$, shown in the second row of Table 4. The number of FLSs that constitute the Rose illumination ($\alpha=65,321$) is not an even multiple of $\alpha$. Hence, the middle 3 rows show the characteristics of the last flock that has the remaining FLSs. Note that its value of $S$ is higher than 1 second (1000 millisecond) because it has fewer FLSs.

The characteristics of today’s Sky Viper battery increases the number of FLSs to render the Rose illumination to 130,642. This is a 200% overhead. This overhead decreases linearly as we enhance battery characteristics, see the 2nd to last row of Table 4. With an 8 fold overall improvement in battery characteristics (4 fold enhancement of $\beta$ and 4 fold reduction of $\Omega$), STAG’s overhead decreases to 12.5%.

---

6 RELATED WORK

FLS displays render virtual objects in a physical volume. They are in the same class of systems as physical artifacts [42] using programmable matter consisting of catoms [19], roboxels as cellular robots that dynamically configure themselves into the desired shape and size [27], fast 3D printing [13], and BitDrones as interactive nano-quadcopter [20]. These studies describe 3D displays. However, they do not present algorithms to render 3D illuminations.

FLS displays are inspired by today’s indoor and outdoor drone shows that use illuminated, synchronized, and choreographed groups of drones arranged into various aerial formations. A centralized, offline algorithm to compute a flight-lighting plan for outdoor light show performances is presented in [40]. This algorithm requires drones to be placed in a field and in a specific arrangement. It computes collision free paths for 500 drones to display different images in sequence, e.g., a ballerina followed by a guitar. It may be modified for use by the orchestrator to compute flight paths for point clouds that transition one scene to the next. Motill is different because it is designed for a scene where changes from one point cloud to the next is not anticipated to be drastic.

There are many path planning algorithms for robots and UAVs [4, 6–11, 14, 15, 21, 23–25, 29, 30, 32, 33, 38, 41, 43–48]. They address the challenge of moving from a given initial position to a set of predefined targets while avoiding collisions with obstacles as well as other UAVs. Most relevant are studies that avoid UAV (robot) collision using an artificial potential field (APF) that defines a safety radius around the drone [23, 24, 41]. With APF, the UAV (robot) moves to its target point guided by attractive force and repulsive forces. These techniques are applicable to FLSs that execute the flight paths computed by Motill to render a motion illumination. Our techniques have the added advantage that an FLS is informed of a potential conflict and FLSs may communicate to avoid a collision. In [23], a modified version of APF is used to identify a failing (rogue) drone. An FLS display may use this technique to detect a failed FLS.

FLSs are network enabled and the failure detection techniques described in Section 4 are inspired by those used in peer-to-peer networks, e.g., CAN [34], Chord [39]. Formation of FLS groups and use of standbys to tolerate failures is similar to disk striping techniques [31, 35] in-use by the disk manufacturers. An interesting dimension introduced by FLSs is their flight paths that may change their memberships in groups.

7 FUTURE WORK

FLS displays are relatively new with many exciting future extensions. Due to lack of space, we describe only one. A swarm of FLSs may implement encounter-type haptic interactions [28] by generating force back against a user touch [1, 2, 5, 18, 22]. This will enable a user to touch virtual objects as illuminations without wearing gloves [16]. User safety and trust are paramount and we intend to address them from the start [18].

8 ACKNOWLEDGMENTS

We thank Hamed Alimohammadzadeh for generating the Rose Clip and his drawings. We are grateful to the ACM Multimedia 2022’s anonymous reviewers for their valuable comments.
[44] Jur van den Berg, Stephen Guy, Ming Lin, and Dinesh Manocha. 2011. *Reciprocal n-Body Collision Avoidance*. Vol. 70. Berlin, Heidelberg, 3–19. https://doi.org/10.1007/978-3-642-19457-3_1

[45] Jur van den Berg, Ming Lin, and Dinesh Manocha. 2008. Reciprocal Velocity Obstacles for Real-Time Multi-agent Navigation. In *IEEE International Conference on Robotics and Automation*. IEEE, USA, 1928–1935. https://doi.org/10.1109/ROBOT.2008.4543489

[46] Jur van den Berg, Jamie Snape, Stephen Guy, and Dinesh Manocha. 2011. Reciprocal Collision Avoidance with Acceleration-Velocity Obstacles. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, USA, 3475–3482. https://doi.org/10.1109/ICRA.2011.5980408

[47] Jur van den Berg, David Wilkie, Stephen Guy, Marc Niethammer, and Dinesh Manocha. 2012. LQG-obstacles: Feedback Control with Collision Avoidance for Mobile Robots with Motion and Sensing Uncertainty. In *International Conference on Robotics and Automation*. IEEE, 346–353. https://doi.org/10.1109/ICRA.2012.6224648

[48] Yang Xu, Shupeng Lai, Jiaxin Li, Delin Luo, and Yancheng You. 2019. Concurrent Optimal Trajectory Planning for Indoor Quadrotor Formation Switching. *Journal of Intelligent & Robotic Systems* 94 (05 2019). https://doi.org/10.1007/s10846-018-0813-9